RL-RTX

The RTX kernel is a real time operating system (RTOS) that enables you to create applications that simultaneously perform multiple functions or **tasks**. This is very useful in embedded applications. While it is certainly possible to create real-time programs without an RTOS (by executing one or more tasks in a loop), there are numerous scheduling, maintenance, and timing issues that an RTOS like RTX can solve for you.

An RTOS enables flexible **scheduling** of system resources like CPU and memory, and offers ways to communicate between tasks. The RTX kernel is a powerful RTOS that is easy to use and works with microcontrollers that are based on the ARM7™TDMI, ARM9™, or Cortex™-M3 CPU core.

RTX programs are written using standard C constructs and compiled with the RealView® Compiler. The **RTX.H** header file defines the RTX functions and macros that allow you to easily declare tasks and access all RTOS features.

The topic **Create New RTX Application** provides a step-by-step introduction that explains you how to create RTX applications.

Overview

The RTX kernel is an easy to use **R**eal **T**ime e**X**ecutive for microcontrollers that are based on ARM CPU cores. It provides a set of C functions and C macros that allow you to build real time applications using tasks that run quasi-parallel on the CPU.

This section provides basic information about the RTX kernel, lists timing information, and explains the concepts of quasi-parallel task exection.

Product Description

The RTX kernel provides basic functionality to start and stop concurrent tasks (processes). It provides additional functions for interprocess communication. You can use the communication functions to synchronize different tasks, manage common resources (like peripherals or memory regions), and pass complete messages between tasks.

Basic Functions

You can use the basic functions to start up the Real Time Executive, to start and stop tasks, and to pass control from one task to another (round-robin scheduling). You can assign execution priorities to tasks. When there is more than one task in the ready list, the RTX kernel uses the execution priorities to select the next task to run (preemptive scheduling).

Interprocess Communications

The RTX kernel provides several ways for interprocess communication. These are:

- **Event flags**
  The primary means of implementing a task synchronization are the event flags. Each task has 16 event flags assigned to it. Hence, you can make a task selectively wait for 16 different events. You can make a task wait for more than one flag at the same time. In this case, you can either make the task wait for all the selected flags (AND-connection), or make the task wait for any one of the selected flags (OR-connection).

  Event flags can be set either by tasks or by ARM **interrupt functions**. You can thus synchronize an external asynchronous event to an RTX kernel task by making the ARM interrupt function set a flag that the task is waiting for.
- **Semaphores**
  If more than one task needs to access a common resource, special means are required in a real time multitasking system. Otherwise, accesses by different tasks might interfere and lead to inconsistent data, or a peripheral element might function incorrectly.

  Semaphores are the primary means of avoiding such access problems. Semaphores (binary semaphores) are software objects containing a **virtual token**. The kernel gives the token to the first task that requests it. No other task can obtain the token until it is released back into the semaphore. Since only the task that has the token can access the common resource, it prevents other tasks from accessing and interfering with the common resource.

  The RTX kernel puts a task to sleep if the requested token is not available in the semaphore. The kernel wakes the task up and puts it in the ready list as soon as the token is returned to the semaphore. You can also use a time out to ensure that the task does not sleep indefinitely.
- **Mutexes**
  An alternative way to avoid synchronization and memory access problems is to use mutual exclusion locks (mutexes). These are software objects that a task can use to lock the common resource, so that only the task that locks it can access the common resource. The kernel blocks all other tasks that request the mutex lock until the task that locked the mutex unlocks it.
- **Mailboxes**
  Tasks can pass **messages** between each other using mailboxes. This is usually the case when implementing various high level protocols like TCP-IP, UDP, and ISDN.

  The message is simply a **pointer** to the block of memory containing a protocol message or frame. It is the programmer's responsibility to dynamically allocate and free the memory block to prevent memory leaks.

  The RTX kernel puts the waiting task to sleep if the message is not available . The kernel wakes the task up as soon as another task sends a message to the mailbox.

Product Specification

RTX Kernel Library is implemented in several versions:

- **ARM7™/ARM9™** version
- **Cortex™-M0/M1** version
- **Cortex™-M3** version
- **Cortex™-M4** version
- **Cortex™-R4** version

A different implementation for Cortex™-M devices is mainly because this core has extended RTOS features, which allows more robust and fail-proof RTX Kernel implementation.

Main concept differences are:

- ARM7™/ARM9™ version uses the system task manager to control task switches of all user tasks. It is executed in System Mode.
- Cortex™-M version uses system calls that are all executed as **SVC** System Supervisor Calls.

Technical Data

Description	ARM7™/ARM9™	Cortex™-M
Defined Tasks	Unlimited	Unlimited
Active Tasks	250 max	250 max
Mailboxes	Unlimited	Unlimited
Semaphores	Unlimited	Unlimited
Mutexes	Unlimited	Unlimited
Signals / Events	16 per task	16 per task
User Timers	Unlimited	Unlimited
Code Space	<4.2 Kbytes	<4.0 Kbytes
RAM Space for Kernel	300 bytes + 80 bytes User Stack	300 bytes + 128 bytes Main Stack
RAM Space for a Task	TaskStackSize + 52 bytes	TaskStackSize + 52 bytes
RAM Space for a Mailbox	MaxMessages * 4 + 16 bytes	MaxMessages * 4 + 16 bytes
RAM Space for a Semaphore	8 bytes	8 bytes
RAM Space for a Mutex	12 bytes	12 bytes
RAM Space for a User Timer	8 bytes	8 bytes
Hardware Requirements	One on-chip timer	SysTick timer
User task priorities	1 - 254	1 - 254
Context switch time	<5.3 µsec @ 60 MHz	<2.6 µsec @ 72 MHz
Interrupt lockout time	<2.7 µsec @ 60 MHz	Not disabled by RTX

Note

- **Unlimited** means that the RTX kernel does not impose any limitations on the number. However, the available system memory resources limit the number of items you can create.
- The default configuration of the RTX kernel allows 10 tasks and 10 user timers. It also disables stack checking by default.
- In the RTX kernel, **Event** is simply another name for signal.
- RAM requirements depend on the number of concurrently running tasks.
- The code and RAM size was calculated for **MicoLib** runtime library.

Timing Specifications

Function	ARM7™/ARM9™ (cycles)	Cortex™-M (cycles)
Initialize system (os_sys_init), start task	1721	1147
Create task (no task switch)	679	403
Create task (switch task)	787	461
Delete task (os_tsk_delete)	402	218
Task switch (by os_tsk_delete_self)	458	230
Task switch (by os_tsk_pass)	321	192
Set event (no task switch)	128	89
Set event (switch task)	363	215
Send semaphore (no task switch)	106	72
Send semaphore (switch task)	364	217
Send message (no task switch)	218	117
Send message (switch task)	404	241
Get own task identifier (os_tsk_self)	23	65
Interrupt latency	<160	0

Note

- The table for **ARM7™/ARM9™** RTX Kernel library is measured on **LPC2138** (ARM7), code execution from internal flash with zero-cycle latency.
- The table for **Cortex™-M** RTX Kernel library is measured on **LPC1768** (Cortex-M3), code execution from internal flash with zero-cycle latency.
- The RTX Kernel for the test is configured for 10 tasks, 10 user timers and stack checking disabled.
- Interrupt latency in **ARM7™/ARM9™** includes the ISR prolog generated by the compiler.
- The RTX Kernel library for **Cortex™-M3/M4** does not disable interrupts. Interrupt latency is the same as without the RTX kernel. Interrupt latency for **Cortex™-M0/M1** is <20 cycles.

How To Use

The Real-Time eXecutive (RTX) kernel is based on the idea of parallel tasks (processes). In the RTX kernel, the task that a system must fulfill is split up into several smaller tasks that run concurrently. There are many advantages to using the RTX kernel:

- Real world processes are usually made up of several concurrent activities. This pattern can be represented in software by using the RTX kernel.
- You can make different activities occur at different times, for example just at the moment when they are needed. This is possible because each activity is packed into a separate task, which can be executed on its own.
- You can prioritize the tasks.
- It is easier to understand and manage smaller pieces of code than one large piece of software.
- Splitting up the software into independent parts reduces the system complexity, reduces errors, and even facilitates testing.
- The RTX kernel is scalable, and additional tasks can easily be added at a later time.
- The RTX kernel offers services needed in many real-time applications, for example good handling of interrupts, periodical activation of tasks, and time-limits on wait functions.

Your First RTX Application

This section demonstrates an example of using the RTX kernel for a simple application. The example is located in the folder **\Keil\ARM\RL\RTX\Examples\RTX_ex1**. The application must perform two activities. The first activity must continuously repeat 50 ms after the second activity completes. The second activity must repeat 20 ms after the first activity completes.

Hence, you can implement these activities as two separate tasks, called task1 and task2:

1. Place the code for the two activities into two separate functions (task1 and task2). Declare the two functions as tasks using the keyword **__task** (defined in RTL.H) which indicates a RTX task.

```

__task void task1 (void) {
  .... place code of task 1 here ....
}

__task void task2 (void) {
  .... place code of task 2 here ....
}
```

10. When the system starts up, the RTK kernel must start before running any task. To do this, call the **os_sys_init** function in the C **main** function. Pass the function name of the first task as the parameter to the **os_sys_init** function. This ensures that after the RTX kernel initializes, the task starts executing rather than continuing program execution in the **main** function.

    In this example, task1 starts first. Hence, task1 must create task2. You can do this using the **os_tsk_create** function.

```

__task void task1 (void) {
  os_tsk_create (task2, 0);
  .... place code of task 1 here ....
}

__task void task2 (void) {
  .... place code of task 2 here ....
}

void main (void) {
  os_sys_init (task1);
}
```

11. Now implement the timing requirements. Since both activities must repeat indefinitely, place the code in an endless loop in each task. After the task1 activity finishes, it must send a signal to task2, and it must wait for task2 to complete. Then it must wait for 50 ms before it can perform the activity again. You can use the **os_dly_wait** function to wait for a number of system intervals. The RTX kernel starts a system timer by programming one of the on-chip hardware timers of the ARM processors. By default, the system interval is 10 ms and timer 0 is used (this is configurable).

    You can use the **os_evt_wait_or** function to make task1 wait for completion of task2, and you can use the **os_evt_set** function to send the signal to task2. This examples uses bit 2 (position 3) of the event flags to inform the other task when it completes.

    task2 must start 20 ms after task1 completes. You can use the same functions in task2 to wait and send signals to task1. The listing below shows all the statements required to run the example:

```

/* Include type and function declarations for RTX */
#include "rtl.h"

```

```
/* id1, id2 will contain task identifications at run-time */
OS_TID id1, id2;

/* Forward declaration of tasks. */
__task void task1 (void);
__task void task2 (void);

__task void task1 (void){
  /* Obtain own system task identification number */
  id1 = os_tsk_self();

  /* Create task2 and obtain its task identification number */
  id2 = os_tsk_create (task2, 0);

  for (;;) {
    /* ... place code for task1 activity here ... */

    /* Signal to task2 that task1 has compelted */
    os_evt_set(0x0004, id2);

    /* Wait for completion of task2 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 50 ms before restarting task1 activity. */
    os_dly_wait(5);
  }
}

__task void task2 (void) {
  for (;;) {
    /* Wait for completion of task1 activity. */
    /*  0xFFFF makes it wait without timeout. */
    /*  0x0004 represents bit 2. */
    os_evt_wait_or(0x0004, 0xFFFF);

    /* Wait for 20 ms before starting task2 activity. */
    os_dly_wait(2);

    /* ... place code for task2 activity here ... */

    /* Signal to task1 that task1 has compelted */
    os_evt_set(0x0004, id1);
  }
}

void main (void) {
  /* Start the RTX kernel, and then create and execute task1. */
  os_sys_init(task1);
}
```

67. Finally, to compile the code and link it with the RTX library, you must select the RTX operating system for the project. From the main menu, select **Project —> Options for Target**. Select the **Target** tab. Select RTX Kernel for the **Operating system**. Build the project to generate the absolute file. You can run the object file output from the linker either on your target or on the µVision Simulator.

Theory of Operation

The RTX kernel uses and manages your target system's resources. This section describes the resources and how they are managed by the RTX kernel.

Many aspects of the RTX kernel can be configured on a project by project basis. This is mentioned where applicable.

Timer Tick Interrupt

The RTX kernel for ARM7™ and ARM9™ uses one of the standard ARM timers to generate a periodic interrupt. RTX kernel for Cortex™-M uses common SysTick timer. This interrupt is called the RTX kernel timer tick. For some of the RTX library functions, you must specify timeouts and delay intervals in number of RTX kernel timer ticks.

The parameters for the RTX kernel timer are selected in the **RTX_Config.c** configuration file. Each ARM microcontroller family provides a different peripherals that are supported with different **RTX_Config.c** files.

For example, the **RTX_Config.c** file for NXP LPC2100/LPC2200 allows to use **Timer 0** or **Timer 1** for the RTX kernel timer.

The **timer clock value** specifies the input clock frequency and depends on CPU clock and APB clock. For a device with CPU clock 60 MHz and VPB divider 4 the peripheral clock is 15MHz and therefore the value is set to 15000000.

The **time tick value** specifies the interval of the periodic RTX interrupt. The value 10000 us configures timer tick period of 0.01 seconds.

System Task Manager

The task manager is a system task that is executed on every **timer tick interrupt**. It has the highest assigned priority and does not get preempted. This task is basically used to switch between user tasks.

The RTX tasks are not really executed concurrently. They are **time-sliced**. The available CPU time is divided into time slices and the RTX kernel assigns a time slice to each task. Since the time slice is short (default time slice is set to 10 ms) it appears as though tasks execute simultaneously.

Tasks execute for the duration of their time-slice unless the task's time-slice is given up explicitly by calling the **os_tsk_pass** or one of the **wait** library functions. Then the RTX Kernel switches to the next task that is ready to run. You can set the duration of the time-slice in the **RTX_Config.c** configuration file.

The **task manager** is the system tick timer task that manages all other tasks. It handles the task's delay timeouts and puts waiting tasks to sleep. When the required event occurs, it puts the waiting tasks back again into the ready state. This is why the tick timer task must have the highest priority.

The task manager runs not only when the timer tick interrupt occurs, but also when an interrupt calls one of the **isr_** functions. This is because interrupts cannot make the current task wait, and therefore interrupts cannot perform task switching. However, interrupts can generate the event, semaphore or mailbox message (using an **isr_** library function) that a higher priority task is waiting for. The higher priority task must preempt the current task, but can do so only after the interrupt function completes. The interrupt therefore forces the timer tick interrupt, which runs when the current interrupt finishes. The forced tick timer interrupt starts the **task manager** (clock task) scheduler. The task scheduler process all the tasks and then puts the highest ready task into the running state. The highest priority task can then continue with its execution.

Note

- The tick timer task is an RTX system task and is therefore created by the system.
- The RTX library for **Cortex™-M** uses extended RTOS features of Cortex™-M devices. All RTX system functions are executed in **svc mode**.

Task Management

Each RTX task is always in exactly one **state**, which tells the disposition of the task.

State	Description
RUNNING	The task that is currently running is in the **RUNNING** state. Only one task at a time can be in this state. The os_tsk_self() returns the Task ID (TID) of the currently executing task.
READY	Tasks which are ready to run are in the **READY** state. Once the running task has completed processing, RTX selects the next ready task with the **highest priority** and starts it.
WAIT_DLY	Tasks which are waiting for a delay to expire are in the **WAIT_DLY** State. Once the delay has expired, the task is switched to the **READY** state. The os_dly_wait() function is used to place a task in the **WAIT_DLY** state.
WAIT_ITV	Tasks which are waiting for an interval to expire are in the **WAIT_ITV** State. Once the interval delay has expired, the task is switched back to the **READY** State. The os_itv_wait() function is used to place a task in the **WAIT_IVL** State.
WAIT_OR	Tasks which are waiting for at least one event flag are in the **WAIT_OR** State. When the event occurs, the task is switched to the **READY** state. The os_evt_wait_or() function is used to place a task in the **WAIT_OR** state.
WAIT_AND	Tasks which are waiting for all the set events to occur are in the **WAIT_AND** state. When all event flags are set, the task is switched to the **READY** state. The os_evt_wait_and() function is used to place a task in the **WAIT_AND** state.
WAIT_SEM	Tasks which are waiting for a semaphore are in the **WAIT_SEM** state. When the token is obtained from the semaphore, the task is switched to the **READY** state. The os_sem_wait() function is used to place a task in the **WAIT_SEM** state.
WAIT_MUT	Tasks which are waiting for a free mutex are in the **WAIT_MUT** state. When a mutex is released, the task acquire the mutex and switch to the **READY** state. The os_mut_wait() function is used to place a task in the **WAIT_MUT** state.
WAIT_MBX	Tasks which are waiting for a mailbox message are in the **WAIT_MBX** state. Once the message has arrived, the task is switched to the **READY** state. The os_mbx_wait() function is used to place a task in the **WAIT_MBX** state. Tasks waiting to send a message when the mailbox is full are also put into the **WAIT_MBX** state. When the message is read out from the mailbox, the task is switched to the **READY** state. In this case the os_mbx_send() function is used to place a task in the **WAIT_MBX** state.
INACTIVE	Tasks which have not been started or tasks which have been deleted are in the **INACTIVE** state. The os_tsk_delete() function places a task that has been started (with os_tsk_create()) into the **INACTIVE** state.

Idle Task

When no task is ready to run, the RTX kernel executes the idle task **os_idle_demon**. The idle task is simply an endless loop. For example:

```
for (;;);
```

ARM devices provide an **idle mode** that reduces power consumption by halting program execution until an interrupt occurs. In this mode, all peripherals, including the interrupt system, still continue to work.

The RTX kernel initiates idle mode in the os_idle_demon task (when no other task is ready for execution). When the RTX kernel **timer tick interrupt** (or any other interrupt) occurs, the microcontroller resumes program execution.

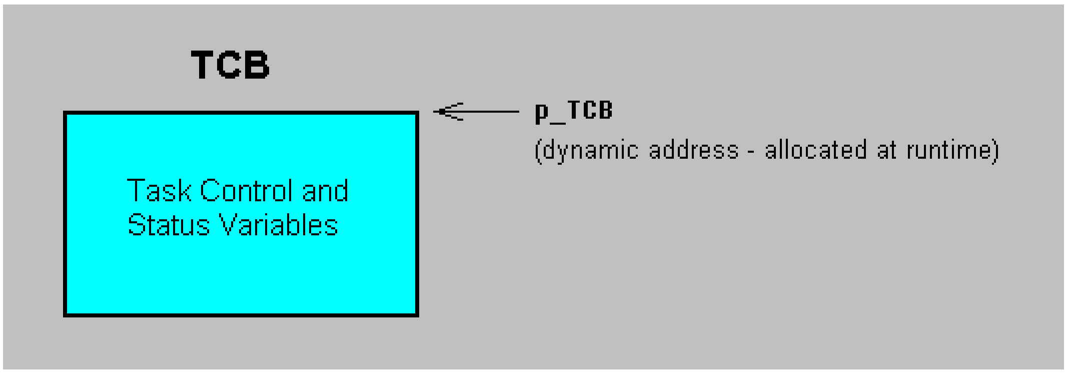
You can add your own code to the os_idle_demon task. The code executed by the idle task can be configured in the **RTX_Config.c** configuration file.

Note

- The idle task **os_idle_demon** is an RTX kernel system task and is therefore created by the system.
- Do not use **idle mode** if you are using the JTAG interface for debugging. Some ARM devices may stop communicating over the JTAG interface when idle.

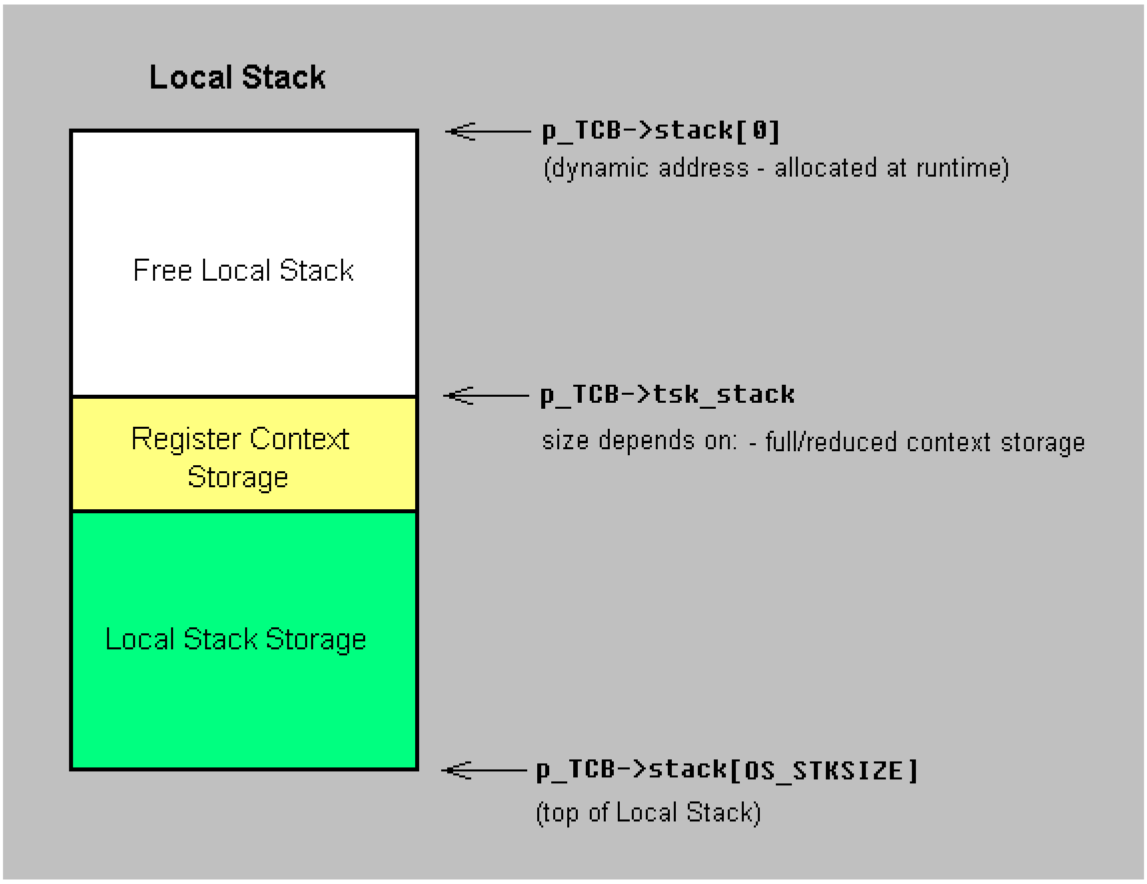
System Resources

RTX kernel tasks are identified by their Task Control Block (TCB). This is a dynamically allocated block of memory where all task control and status variables are located. TCB is allocated at runtime when the task is created with the **os_tsk_create** or **os_tsk_create_user** function call.


TCB
Task Control and Status Variables
p_TCB
(dynamic address - allocated at runtime)

The size of the TCB memory pool is defined in the **RTX_Config.c** configuration file, and it depends on the number of concurrent running tasks. This is not necessarily the number of defined tasks since **multiple instances** of a task are supported by the RTX kernel.

The RTX kernel also allocates the task its own **stack**. The stack is allocated at runtime after the TCB has been allocated. The pointer to the stack memory block is then written into the TCB.



Scheduling Options

RTX allows you to build an application with three different kernel-scheduling options. These are:

- **Pre-emptive scheduling**
  Each task has a **different priority** and will run until it is pre-empted or has reached a blocking OS call.
- **Round-Robin scheduling**
  Each task has the **same priority** and will run for a fixed period, or time slice, or until has reached a blocking OS call.
- **Co-operative multi-tasking**
  Each task has the **same priority** and the Round-Robin is disabled. Each task will run until it reached a blocking OS call or uses the **os_tsk_pass()** call.

The default scheduling option for RTX is Round-Robin Pre-emptive. For most applications, this is the most useful option.

Pre-emptive Scheduling

RTX is a **pre-emptive** multitasking operating system. If a task with a higher priority than the currently running task becomes ready to run, RTX **suspends** the currently running task.

A preemptive task switch occurs when:

- the task scheduler is executed from the system **tick timer interrupt**. Task scheduler processes the **delays** of tasks. If the delay for a task with a higher priority has expired, then the higher priority task starts to execute instead of currently running task.
- an **event** is set for a higher priority task by the currently running task or by an interrupt service routine. The currently running task is suspended, and the higher priority task starts to run.
- a token is returned to a **semaphore**, and a higher priority task is waiting for the semaphore token. The currently running task is suspended, and the higher priority task starts to run. The token can be returned by the currently running task or by an interrupt service routine.
- a **mutex** is released and a higher priority task is waiting for the mutex. The currently running task is suspended, and the higher priority task starts to run.
- a message is posted to a **mailbox**, and a higher priority task is waiting for the mailbox message. The currently running task is suspended, and the higher priority task starts to run. The message can be posted by the currently running task or by an interrupt service routine.
- a **mailbox** is **full**, and a higher priority task is waiting to post a message to a mailbox. As soon as the currently running task or an interrupt service routine takes a message out from the mailbox, the higher priority task starts to run.
- the **priority** of the currently running task is reduced. If another task is ready to run and has a higher priority than the new priority of the currently running task, then the current task is suspended immediately, and the higher priority task resumes its execution.

The following example demonstrates one of the task switching mechanisms. Task job1 has a higher priority than task job2. When job1 starts, it creates task job2 and then enters the **os_evt_wait_or** function. The RTX kernel suspends job1 at this point, and job2 starts executing. As soon as job2 sets an event flag for job1, the RTX kernel suspends job2 and then resumes job1. Task job1 then increments counter cnt1 and calls the **os_evt_wait_or** function, which suspends it again. The kernel resumes job2, which increments counter cnt2 and sets an event flag for job1. This process of task switching continues indefinitely.

```
#include <rtl.h>

OS_TID tsk1,tsk2;
int    cnt1,cnt2;

__task void job1 (void);
__task void job2 (void);

__task void job1 (void) {
  os_tsk_prio (2);
  os_tsk_create (job2, 1);
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    cnt1++;
  }
}

__task void job2 (void) {
  while (1) {
    os_evt_set (0x0001, job1);
    cnt2++;
  }
}
```

```
void main (void) {
  os_sys_init (job1);
  while (1);
}
```

Round-Robin Scheduling

RTX can be configured to use Round-Robin Multitasking (or task switching). Round-Robin allows quasi-parallel execution of several tasks. Tasks are not really executed concurrently but are **time-sliced** (the available CPU time is divided into time slices and RTX assigns a time slice to each task). Since the time slice is short (only a few milliseconds) it appears as though tasks execute simultaneously.

Tasks execute for the duration of their time-slice (unless the task's time slice is given up). Then, RTX switches to the next task that is **ready** to run and has the **same priority**. If no other task with the same priority is ready to run, the currently running task resumes it execution. The duration of a time slice can be defined in the **RTX_config.c** configuration file.

The following example shows a simple RTX program that uses Round-Robin Multitasking. The two tasks in this program are counter loops. RTX starts executing task 1, which is the function named **job1**. This function creates another task called **job2**. After **job1** executes for its time slice, RTX switches to **job2**. After **job2** executes for its time slice, RTX switches back to **job1**. This process repeats indefinitely.

```
#include <rtl.h>

int counter1;
int counter2;

__task void job1 (void);
__task void job2 (void);

__task void job1 (void) {
  os_tsk_create (job2, 0);   /* Create task 2 and mark it as ready */
  while (1) {                /* loop forever */
    counter1++;              /* update the counter */
  }
}

__task void job2 (void) {
  while (1) {                /* loop forever */
    counter2++;              /* update the counter */
  }
}

void main (void) {
  os_sys_init (job1);        /* Initialize RTX Kernel and start task 1 */
  for (;;);
}
```

Note

- Rather than wait for a task's time slice to expire, you can use one of the **system wait** functions or the **os_tsk_pass** function to signal to the RTX kernel that it can switch to another task. The system wait function suspends the current task (changes it to the **WAIT_xxx** state) until the specified event occurs. The task is then changed to the **READY** state. During this time, any number of other tasks can run.

Cooperative Multitasking

If you disable Round-Robin Multitasking you must design and implement your tasks so that they work **cooperatively**. Specifically, you must call the system wait function like the **os_dly_wait()** function or the **os_tsk_pass()** function somewhere in each task. These functions signal the RTX kernel to switch to another task.

The following example shows a simple RTX program that uses Cooperative Multitasking. The RTX kernel starts executing task 1. This function creates task 2. After counter1 is incremented once, the kernel switches to task 2. After counter2 is incremented once, the kernel switches back to task 1. This process repeats indefinitely.

```
#include <rtl.h>

int counter1;
int counter2;

__task void task1 (void);
__task void task2 (void);

__task void task1 (void) {
  os_tsk_create (task2, 0);  /* Create task 2 and mark it as ready */
  for (;;) {                 /* loop forever */
    counter1++;              /* update the counter */
    os_tsk_pass ();          /* switch to 'task2' */
  }
}

__task void task2 (void) {
  for (;;) {                 /* loop forever */
    counter2++;              /* update the counter */
    os_tsk_pass ();          /* switch to 'task1' */
  }
}

void main (void) {
  os_sys_init(task1);        /* Initialize RTX Kernel and start task 1 */
  for (;;);
}
```

The difference between the system wait function and os_tsk_pass is that the system wait function allows your task to wait for an event, while os_tsk_pass switches to another ready task immediately.

Note

- If the next ready task has a lower priority than the currently running task, then calling **os_tsk_pass** does not cause a task switch.

Priority Inversion

The RTX Real Time Operating system employs a priority-based preemptive scheduler. The RTX scheduler assings each task a unique priority level. The scheduler ensures that *of those tasks that are ready to run*, the one with the highest priority is always the task that is actually running.

Because tasks share resourcs, events outside the scheduler's control can prevent the highest priority ready task from running when it should. If this happens, a critical deadline could be missed, causing the system to fail. **Priority inversion** is the term of a scenario in which the highest-priority ready task fails to run when it should.

Resource sharing

Tasks need to share resources to communicate and process data. Any time two or more tasks share a recource, such as a memory buffer or a serial port, one of them will usually have a higher priority. The higher-priority task expects to be run as soon as it is ready. However, if the lower-priority task is using their shared resource when the higher-priority task becomes ready to run, the higher-priority task must wait for the lower-priority task to finish with it.

Priority inheritance

To prevent priority inversions, the RTX Real Time OS employs a **Priority inheritance** method. The lower-priority task **inherit** the priority of any higher-priority task pending on a resource they share. For a short time, the lower-priority task runs at a priority of a higher-priority pending task. The priority change takes place as soon as the high-priority task begins to pend. When the lower-priority task stops using a shared resource, it's priority level returns to normal.

The RTX **mutex** objects (Mutual Exclusive Lock objects) employ the Priority inheritance.

Stack Management

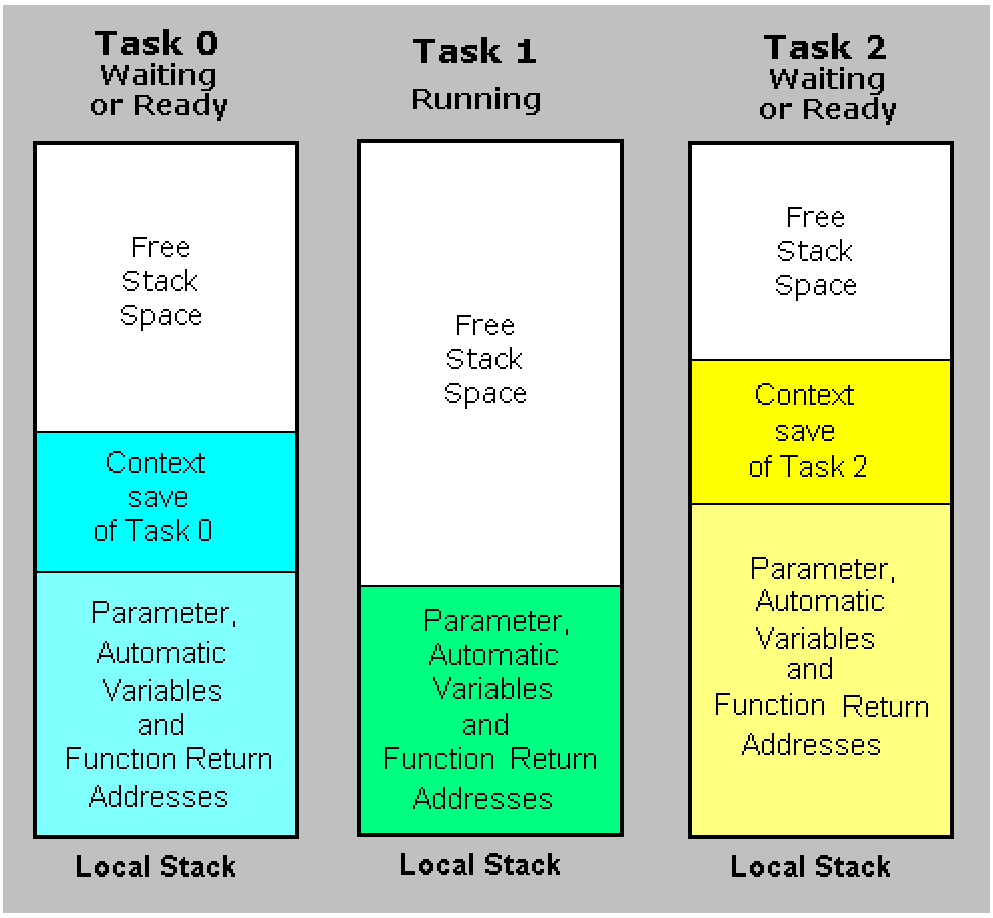
The Stack Management of the RTX kernel is designed for optimal memory usage. The RTX kernel system needs one stack space for the task that is currently in the **RUNNING** state:

- **Local Stack**: stores parameters, automatic variables, and function return addresses. On the ARM device, this stack can be anywhere. However, for performance reasons, it is better to use the on-chip RAM for the local stack.

When a task switch occurs:

- the context of the currently running task is stored on the local stack of the current task
- the stack is switched to that of the next task
- the context of the new task is restored
- the new task starts to run.

The Local Stack also holds the task context of waiting or ready tasks.



The other stack spaces need to be configured from the ARM **startup** file. All tasks run in **user mode**. The task scheduler switches the user/system mode stack between tasks. For this reason, the **default user/system** mode **stack** (which is defined in the startup file) is used until the first task is created and started. The default stack requirements are very small, so it is optimal to set the user/system stack in the startup file to **64 bytes**.

User Timers

User Timers are simple timer blocks that count down on every system timer tick. They are implemented as single shot timers. This means you cannot pause and restart these timers. However, you can create and kill the user timers dynamically at runtime. If you do not kill a user timer before it expires, the RTX kernel calls the user provided callback function, **os_tmr_call()**, and then deletes the timer when it expires.

A timeout value is defined when the timer is created by the **os_tmr_create()** function.

The RTX kernel calls the callback function with the argument **info**. The user provides this argument when creating the user timer. The RTX kernel stores the argument in the timer control block. When the timer expires, this argument is passed back to the user in the **os_tmr_call()** function. If the user kills the timer before the timeout value expires, the RTX kernel does not call the callback function.

You can customize the callback function **os_tmr_call()** in the **RTX_Config.c** configuration file.

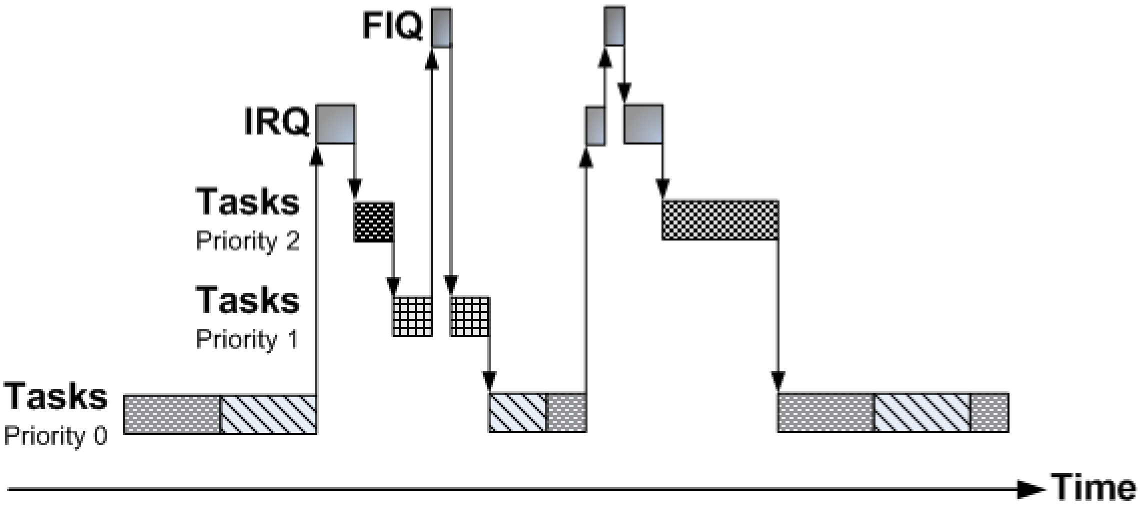
Note

- The callback function, **os_tmr_call**, is called from the system task scheduler. It is recommended to make your **os_tmr_call()** function as small and fast as possible because the callback function **blocks** the RTX task scheduler for the length of time it executes.
- The function **os_tmr_call** behaves the same way as standard interrupt functions. It is allowed to call the **isr_** functions to set an event, send a semaphore, or send a message to other tasks. You cannot call the **os_** library functions from os_tmr_call().

Interrupt Functions

RTX can work with interrupt functions in **parallel**. However, it is better to avoid IRQ nesting. Good programming techniques use short interrupt functions that send signals or messages to RTOS tasks. With this practice, interrupt nesting becomes unimportant. This avoids common problems with nested interrupts where the user mode stack usage becomes unpredictable.

The following figure shows how interrupts should be handled with tasks in the RTX kernel system. An IRQ function can send a signal or message to start a high priority task.



Interrupt functions are added to an ARM application in the same way as in any other non-RTX projects.

Note

- The **FIQ** interrupts are never disabled by the RTX kernel.
- You cannot call the **isr_** library functions from the **FIQ** interrupt function.

The following example shows how to use interrupts with the RTX kernel. The interrupt function, **ext0_int**, sends an **event** to *process_task* and exits. The task *process_task* processes the external interrupt event. In this example, *process_task* is simple and only counts the number of interrupt events.

```
#define EVT_KEY 0x0001

OS_TID pr_task;
int    num_ints;

/*----------------------------------------------------------------------------
 *   External 0 Interrupt Service Routine

*---------------------------------------------------------------------------*/
void ext0_int (void) __irq  {
  isr_evt_set (EVT_KEY, pr_task);      /* Send event to 'process_task'
*/
  EXTINT      = 0x01;                  /* Acknowledge Interrupt
*/
  VICVectAddr = 0;
}

/*----------------------------------------------------------------------------
```

```
 *   Task 'process_task'

*---------------------------------------------------------------------------*/
__task void process_task (void) {
  num_ints = 0;
  while (1) {
    os_evt_wait_or (EVT_KEY, 0xffff);
    num_ints++;
  }
}

/*----------------------------------------------------------------------------
 *   Task 'init_task'

*---------------------------------------------------------------------------*/
__task void init_task (void) {
  PINSEL1 &= ~0x00000003;              /* Enable EINT0
*/
  PINSEL1 |=  0x00000001;
  EXTMODE  = 0x03;                     /* Edge triggered lo->hi transition
*/
  EXTPOLAR = 0x03;

  pr_task = os_tsk_create (process_task, 100);

  VICVectAddr14  = (U32)eint0_int;     /* Task started, Enable interrupts
*/
  VICVectCntl14  = 0x20 | 14;

  os_tsk_delete_self ();               /* Terminate this task
*/
}
```

Configuring RTX Kernel

The RTX kernel is easy to customize for each application you create. This section describes how you can configure the RTX kernel's features for your applications. It contains:

- Basic RTX Configuration
- Advanced RTX Configuration.

Basic RTX Configuration

The RTX kernel must be configured for the embedded applications you create. All configuration settings are found in the **RTX_Config.c** file, which is located in the **\Keil\ARM\Startup** directory. **RTX_Config.c** is configured differently for the different ARM devices. Configuration options in **RTX_Config.c** allow you to:

- Specify the number of concurrent running tasks
- Specify the number of tasks with user-provided stack
- Specify the stack size for each task
- Enable or disable the stack checking
- Enable or disable running tasks in privileged mode
- Specify the CPU timer number used as the system tick timer
- Specify the input clock frequency for the selected timer
- Specify the timer tick interval
- Enable or disable the round-robin task switching
- Specify the time slice for the round-robin task switching
- Define idle task operations
- Specify the number of user timers
- Specify code for the user timer callback function
- Specify the FIFO Queue size
- Specify code for the runtime error function

There is no default configuration in the RL-RTX library. Hence, you must add the **RTX_Config.c** configuration file to each project you create.

To customize the RTX kernel's features, you must change the configurable settings in **RTX_Config.c** .

Tasks

The following **#define**(s) specify how the RTX kernel tasks are configured:

- **OS_TASKCNT** specifies the maximum number of tasks that can be active at the same time. This includes tasks in any state (running, waiting, or ready) other than the INACTIVE state.

  This information is used by the RTX kernel to reserve the memory pool for the task control variables. This number can be higher than the number of defined tasks in your application (one task can run in **multiple instances**) or lower if it is guaranteed that the number of created and running tasks will never exceed OS_TASKCNT.

  ```
  #define OS_TASKCNT     6
  ```
- **OS_PRIVCNT** specifies the number of tasks with **user-provided** stack.

  By default, the RTX kernel allocates a fixed size stack to each task. However, the stack requirement can vary widely between tasks. For example, if a task's local variables include large buffers, arrays, or complex structures, then the task requires a lot more stack. If such a task tries to use more stack than the allocated stack, it might overwrite the stack of neighboring tasks. This is because the fixed size stacks of the tasks are part of the common system stack and are contiguous. This leads to malfunctioning of the RTX kernel and is likely to cause a **system crash**. An intuitive solution to this problem is to increase the fixed stack size. However, this increases the stack size of every other task that might not need the extra stack. To avoid this wastage of valuable resource, a better solution is to allocate a separate user-provided stack for tasks that require a lot more stack.

  The term **user-provided**, in this case, means that the memory space for the task's stack is provided by the user when the **task is created**. It is not automatically assigned by the kernel. The RTX kernel uses OS_PRIVCNT to **optimize** the memory usage. The kernel will not reserve stack space for the tasks with a user-provided stack.

  ```
  #define OS_PRIVCNT     0
  ```

Note

- In addition to OS_TASKCNT user tasks, the system creates one system task **os_idle_demon** **.** This task is always required by the RTX kernel. Total number of concurrent running tasks is **OS_TASKCNT+1** (number of user tasks plus one system task).

Stack Size

The **stack usage** of a particular task depends on its amount of local automatic variables and the number of subroutine levels. Interrupt functions do not use the stack of the interrupted task.

- **OS_STKSIZE** specifies the amount of RAM allocated for the stack of each task. Stack size is defined in U32 (unsigned int). However, Configuration Wizard converts the specified size and displays it in bytes. Stack size with the following define is 400 bytes.

```
#define OS_STKSIZE    100
```

  On the full context task switch, the RTX kernel stores all ARM registers on the stack. Full task context storing requires **64 bytes** of stack.

- The **Cortex-M4** with Hardware Floating Point, needs additional **136 bytes** on stack for storing VFP registers. This means the total size of the full context store for Cortex-M4 with FP is **200 bytes**.
- The **Cortex-M4** tasks, where the Floating Point arithmetics is not used, do not store the additional VFP registers on context save. This means, they do not need additional 136 bytes on the stack. The full context store for tasks with no Floating Point calculations is still **64 bytes**.

Stack Checking

It is possible that the stack is exhausted because of many nested subroutine calls or an extensive use of large automatic variables.

If Stack Checking is enabled, the kernel can detect the stack exhaustion problem and execute the **system error** function. The application will hang in an endless loop inside the error function with parameter *err_code* set to *OS_ERR_STK_OVF*. You can identify the task id with **isr_tsk_get()** function. Check the **Active Tasks** debug dialog for the task name.

The solution to this problem is to increase the stack size for all tasks in the **configuration** file. If only one task requires a big stack and RAM is limited, you can **create** this task with a user-provided stack space.

- **OS_STKCHECK** enables the Stack Checking algorithm. It must be set to **1** to enable it or **0** to disable it. It is enabled by default.

```
#define OS_STKCHECK    1
```

- Enabled Stack Checking slightly **decreases** the kernel performance because on every task switch the kernel needs to execute additional code for stack checking.
- On stack overflow a runtime **system error** function is called.

Run in Privileged Mode

RTX Library version for **Cortex™-M** devices allows to select the running mode of all user tasks. User tasks may run in two modes:

- **Unprivileged** - Protected mode or
- **Privileged** - Unprotected mode.

In **privileged** mode user may access and configure the system and control registers like NVIC interrupt controller etc. This is however not allowed from **unprivileged** mode. An access to NVIC registers from unprivileged mode will result in Hard Fault.

- **OS_RUNPRIV** enables running of all tasks in Privileged mode. It must be set to **1** to enable it or **0** to disable it. It is disabled by default.
-

```
#define OS_RUNPRIV   1
```

You can enable the **privileged mode** for old projects. The existing code will run without any modifications when RTX_Config.c configuration file is replaced with a new one and a project is recompiled for a new Cortex™-M RTX Kernel library. Tasks are not protected in privileged mode and you may configure the system for example the interrupts from any task.

Privileged mode is disabled by default. This allows all tasks to run in protected mode. The tasks are not allowed to change system settings, change interrupts etc. The user has two options:

- run the configuration code in privileged mode as **__svc** function from the task
- run the configuration code before the kernel is started when the device is still running in privileged mode.

- The RTX Kernel library for **ARM7™/ARM9™** does not allow this option because of a different concept.

Hardware Timer

The following **#defines** specify how the RTX kernel's hardware timer is configured:

- **OS_TIMER** specifies the on-chip timer used as a time-base for the real-time system. It delivers a periodic interrupt that wakes up a time-keeping system task. The user can choose which timer serves this purpose. Use 0 for Timer 0, or 1 for Timer 1.

```
#define OS_TIMER        1
```

- **OS_CLOCK** specifies the input clock frequency for the selected timer. This value is calculated as: f(xtal) / VPBDIV. Example is for 15 MHz at @ 60 MHz CPU clock and VPBDIV = 4

```
#define OS_CLOCK        15000000
```

- **OS_TICK** specifies the timer tick interval in µsec. Recommended values are 1000 to 100000. The resulting interval is from 1 ms to 100 ms. Default configuration is for 10 ms.

```
#define OS_TICK         10000
```

Note

- Hardware Timer configuration is required only for **ARM7™** and **ARM9™** RTX Library version. The **Cortex™-M** version uses a common **SysTick** timer for all Cortex™-M device variants.

Round-Robin Multitasking

The following **#define** specify how the RTX kernel **Round-Robin Multitasking** is configured:

- **OS_ROBIN** enables the Round-Robin Multitasking. It must be set to **1** to enable it or **0** to disable it. It is enabled by default.

```
#define OS_ROBIN        1
```

- **OS_ROBINTOUT** specifies the Round-Robin Timeout. This is the time-slice assigned to the currently running task. After this time-slice expires, the currently running task is suspended and the next task ready to run is resumed. It is specified in number of system timer ticks.

```
#define OS_ROBINTOUT    5
```

User Timers

You can create and kill **user timers** at runtime. You must specify the maximum number of running user timers and also the code for the **os_tmr_call()** function.

- **OS_TIMERCNT** specifies the number of user timers that can be created and started. If user timers are not used, set this value to 0. This information is used by RTX to reserve the memory resources for timer control blocks.
-

-
  ```
  #define OS_TIMERCNT    5
  ```
- The **callback** function **os_tmr_call()** is called when the user timer expires. It is provided in the **RTX_Config.c** configuration file as an empty function. You must modify it to suit your needs.

  Parameter **info** is the parameter passed to the **os_tmr_create()** function when the timer was created.

  ```
  /*--------------------------- os_tmr_call
  -----------------------------------*/

  void os_tmr_call (U16 info) {
    /* This function is called when the user timer has expired.
    */
    /* Parameter "info" is the parameter defined when the timer was
  created.  */
    /* HERE: include here optional user code to be executed on timeout.
    */
    info = info;
  }
  ```

FIFO Queue Buffer

The **isr_** library function, when called from the interrupt handler, stores the request type and optional parameter to the **ISR FIFO Queue buffer** to be processed later, after the interrupt handler exits.

The **task manager** is activated immediately after the IRQ handler has finished its execution to process the requests stored to the FIFO Queue buffer. The size of this buffer needed, depends on the number of **isr_** functions, that are called within the interrupt handler.

For example, if there is only one interrupt handler in your project and calls one **isr_evt_set()**, the FIFO Queue buffer size of 4 entries is sufficient. If there are more interrupts used in the project that use the **isr_** communication with RTX kernel or, one interrupt handler that calls several **isr_** library functions, the FIFO Queue buffer size needs to be increased. The interrupts, that do not use **isr_** library functions are not counted here.

Default FIFO Queue buffer size is 16 entries. This should be enough for a typical RTX project.

- **OS_FIFOSZ** specifies the number of entries that can be stored to the FIFO Queue buffer. Default size is 16 entries.

```
#define OS_FIFOSZ    16
```

- On FIFO Queue buffer overflow a runtime **system error** function is called.
- See the **Rtx_Config.c** configuration file for the possible configuration settings.

Error Function

Some system error conditions can be detected during runtime. If RTX kernel detects a **runtime error**, it calls the **os_error()** runtime error function.

```
void os_error (U32 err_code) {
  /* This function is called when a runtime error is detected. */
  OS_TID err_task;

  switch (err_code) {
    case OS_ERR_STK_OVF:
      /* Identify the task with stack overflow. */
      err_task = isr_tsk_get();
      break;
    case OS_ERR_FIFO_OVF:
      break;
    case OS_ERR_MBX_OVF:
      break;
  }
  for (;;);
}
```

The **error code** is passed to this function as a parameter *err_code*:

Error Code	Description
OS_ERR_STK_OVF	The stack checking has detected a stack overflow for the currently running task.
OS_ERR_FIFO_OVF	The ISR FIFO Queue buffer overflow is detected.
OS_ERR_MBX_OVF	The mailbox overflow is detected for **isr_mbx_send()** function.

The runtime error function must contain an **infinite loop** to prevent further program execution. You can use an emulator to step over infinite loop and trace into the code introducing a runtime error. For the overflow errors this means you need to increase the size of the object causing an overflow.

Idle Task

When no tasks are ready to run, the RTX kernel executes the **idle task** with the name **os_idle_demon()**. By default this task is an empty end-less loop that does nothing. It only waits until another task becomes ready to run.

You may change the code of **os_idle_demon()** to put the CPU into a power-saving or idle mode. Most **RTX_Config.c** files define the macro _idle_() that contains the code to put the CPU into a power-saving mode.

Example:

```
/*--------------------------- os_idle_demon
---------------------------------*/

__task void os_idle_demon (void) {
  /* The idle demon is a system task. It is running when no other task is   */
  /* ready to run (idle situation). It must not terminate. Therefore it      */
  /* should contain at least an endless loop.                               */

  for (;;) {
    _idle_();    /* enter low-power mode */
  }
}
```

Note

- On some devices, the IDLE blocks debugging via the JTAG interface. Therefore JTAG debuggers such as ULINK may not work when you are using CPU power-saving modes.
- For using power-saving modes, some devices may require additional configuration (such as clock configuration settings).

Advanced RTX Configuration

RL-ARM provides several versions of the **RTX_Config.c** file for ARM7™/ARM9™ RTX Kernel library. Each one configures the RTX kernel for a specific ARM device variant that RL-ARM supports. However the ARM family of devices is growing quickly, and it is possible that RL-ARM does not contain the configuration file for the device you use. In this case, you can take the **RTX_Config.c** file for the NXP device as a template and modify it for your particular ARM device. This file is located in the **\Keil\ARM\Startup\Philips** directory.

All hardware dependent definitions are extracted from the code and defined with macros. This makes it possible to customize the configuration without modifying the code.

- RTX Kernel library for Cortex™-M has only one configuration file which is common for all Cortex™-M device variants.

HW Resources Required

In order to run the RTX kernel, the following hardware resources are required from an ARM device:

- **Peripheral Timer** for generating periodic ticks. It is better to use a peripheral timer with an auto-reload function. RTX also supports timers with manual timer (or counter) reload. However, this can generate jitter and inaccuracy in the long run. The RTX kernel needs a **count-up** timer. If the timer used is a count-down timer, you need to convert the timer value.
- **Timer Interrupts** to interrupt the execution of a task and to start the system task scheduler.
- **Forced Interrupts** to force a timer interrupt when **isr_** functions are used. If an **isr_** function is called, the kernel forces the timer interrupt immediately after the interrupt ends. The forced timer interrupt activates the task scheduler. It is possible that a task has become ready. If this task has a higher priority than the currently running task, a task switch must occur.

Configuration Macros

All hardware related configuration options are described with configuration macros. Some of the macros (for example **OS_TID_** and **OS_TIM_**) are used only to simplify the code. They are not used in all of the configuration files. Using configuration macros allows easy customization of the configuration for the different peripheral timers supported by an ARM device.

The following configuration macros are introduced:
(examples are for Philips LPC21xx devices - Timer 0)

- **OS_TRV** macro specifies the timer reload value for the peripheral timer. Peripheral timer counts up to a reload value, then then overflows to 0, and then generates a tick interrupt. The reload value should be calculated to generate the desired interval length (for example 10 ms).

  ```
  #define OS_TRV      ((U32)(((double)OS_CLOCK*(double)OS_TICK)/1E6)-1)
  ```

- **OS_TVAL** macro is used to read the current timer value for a **count-up** timer. The RTX kernel uses it to check whether a timer interrupt is a periodic timer interrupt or a software forced interrupt.

  ```
  #define OS_TVAL     T0TC                /* Timer Value */
  ```

  For a countdown timer, you must convert the return value. This is an example for a 16-bit count-down timer:

  ```
  #define OS_TVAL     (0xFFFF - T0VAL)   /* Timer Value */
  ```

- **OS_TOVF** specifies a timer overflow flag. The RTX kernel uses it together with the OS_TVAL macro to differentiate between the periodic timer interrupts and forced timer interrupts.

  ```
  #define OS_TOVF     (T0IR & 1)          /* Overflow Flag */
  ```

- **OS_TREL()** macro specifies a code sequence to reload a peripheral timer on overflow. When a peripheral timer has an auto-reload functionality, this macro is left empty.

  ```
  #define OS_TREL()   ;                   /* Timer Reload */
  ```

- **OS_TFIRQ()** specifies a code sequence to force a timer interrupt. This must be a software triggered interrupt if the peripheral timer does not allow manual setting of an overflow flag. If manual setting is possible, this macro should set a peripheral timer overflow flag, which will cause a timer interrupt.

  ```
  #define OS_TFIRQ()  VICSoftInt = OS_TIM_;  /* Force Interrupt */
  ```

- **OS_TIACK()** is used to acknowledge an interrupt from the timer interrupt function to release the timer interrupt logic.

  ```
  #define OS_TIACK()  T0IR = 1;            /* Interrupt Ack */  \
                      VICSoftIntClr = OS_TIM_;                  \
                      VICVectAddr   = 0;
  ```

- **OS_TINIT()** macro is used to initialize the peripheral timer/counter, setup a timer mode, and set a timer reload. Timer interrupts are also activated here by enabling a peripheral timer interrupt. This code is executed from the **os_sys_init()** function.

  ```
  #define OS_TINIT()  T0MR0 = OS_TRV;     /* Initialization */  \
                      T0MCR = 3;                                \
                      T0TCR = 1;                                \
                      VICDefVectAddr = (U32)os_def_interrupt;   \
                      VICVectAddr15  = (U32)os_clock_interrupt; \
                      VICVectCntl15  = 0x20 | OS_TID_;
  ```

- **OS_LOCK()** macro disables timer tick interrupts. It is used to avoid interrupting the system task scheduler. This macro should disable both the periodic timer interrupts and the forced interrupts. This code is executed from the **tsk_lock()** function.

  ```
  #define OS_LOCK()   VICIntEnClr = OS_TIM_;    /* Task Lock */
  ```

- **OS_UNLOCK()** macro enables the timer tick interrupts. The code sequence specified here should enable the periodic timer interrupts and the forced interrupts. This code is executed from **tsk_unlock()** function.

```

#define OS_UNLOCK() VICIntEnable = OS_TIM_;  /* Task Unlock */
```

Library Files

RL-RTX includes four library files:

- **RTX_ARM_L.LIB** for microcontrollers based on ARM7TDMI™ and ARM9™ - Little Endian.
- **RTX_ARM_B.LIB** for microcontrollers based on ARM7TDMI™ and ARM9™ - Big Endian.
- **RTX_CM1.LIB** for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Little Endian.
- **RTX_CM1_B.LIB** for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Big Endian.
- **RTX_CM3.LIB** for microcontrollers based on Cortex™-M3 - Little Endian.
- **RTX_CM3_B.LIB** for microcontrollers based on Cortex™-M3 - Big Endian.
- **RTX_CM4.LIB** for microcontrollers based on Cortex™-M4 - Little Endian.
- **RTX_CM4_B.LIB** for microcontrollers based on Cortex™-M4 - Big Endian.
- **RTX_CR4.LIB** for microcontrollers based on Cortex™-R4 - Little Endian.
- **RTX_CR4_B.LIB** for microcontrollers based on Cortex™-R4 - Big Endian.

All RL-ARM libraries are located in the **\Keil\ARM\RV31\LIB\** folder. Depending on the target device selected for your project, the appropriate library file is automatically included into the link process when the RTX kernel operating system is selected for your project.

The **RTX_Lib_ARM.uvproj** and **RTX_Lib_CM.uvproj** projects found in the **\Keil\ARM\RL\RTX\** folder are used to build the RL-RTX libraries.

Note

- You should not explicitly include any of the RL-RTX libraries in your application. That is done automatically when you use the **µVision® IDE**.

Using RTX Kernel

To use RTX Kernel for ARM7™/ARM9™ or Cortex™-M based applications, you must be able to successfully create RTX applications, compile and link them.

Writing Programs

When you write programs for RL-RTX, you can define RTX tasks using the **__task** keyword. You can use the RTX kernel routines whose prototypes are declared in **RTL.h**.

Include Files

The RTX kernel requires the use of only the **RTL.h** include file. All **library routines** and constants are defined in this header file. You can include it in your source files as follows:

```
#include <rtl.h>
```

Defining Tasks

Real-Time or multitasking applications are composed of one or more tasks that perform specific operations. The RTX kernel supports a maximum of 255 tasks.

Tasks are simply C functions that have a **void** return type, have a **void** argument list, and are declared using the **__task** function attribute. For example:

```
__task void func (void);
```

where

func	is the name of the task.

The following example defines the function task1 as a task. This task increments a counter indefinitely.

```
__task void task1 (void) {
  while (1) {
    counter0++;
  }
}
```

Note

- All tasks must be implemented as endless loops. A task must never return.
- The **__task** function attribute **must prefix** the task function declaration in RTX kernel version 3.40 and newer.

Multiple Instances

The RTX kernel enables you to run multiple copies of the same task at the same time. These are called **multiple instances** of the same task. You can simply create and run the same task several times. This is often required when you design a protocol based stack (like ISDN D channel stack).

The following example shows you how the function task2 can run in multiple instances.

```
#include <rtl.h>

OS_TID tsk_1, tsk2_1, tsk2_2, tsk2_3;
int cnt;

__task void task2 (void) {
  for (;;) {
    os_dly_wait (2);
    cnt++;
  }
}

__task void task1 (void) {
  /* This task will create 3 instances of task2 */
  tsk2_1 = os_tsk_create (task2, 0);
  tsk2_2 = os_tsk_create (task2, 0);
  tsk2_3 = os_tsk_create (task2, 0);
  /* The job is done, delete 'task1' */
  os_tsk_delete_self ();
}

void main (void) {
  os_sys_init(task1);
  for (;;);
}
```

Note

- Each instance of the same task must have a unique task ID.

External References

The **semaphore** and **mailbox** objects are referenced by the RTX kernel as typeless object pointers and are typecast inside the specific RTX kernel module. For semaphores and task handles, this is not a problem. The problem is when referencing the mailbox, which is declared using the macro **os_mbx_declare()**. That is why the **OS_MBX** type is defined. You have to use the OS_MBX object type identifier to reference mailboxes in external modules.

Here is an example of how the external RTX kernel objects are referenced:

```

extern OS_TID tsk1;
extern OS_SEM semaphore1;
extern OS_MUT mutex1;
extern OS_MBX mailbox1;
```

The following example shows you how to make a reference to a **mailbox** from a different C-module.

- C-Module with a **mailbox1** declaration:

```

#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
  void *msg;

  os_mbx_init (mailbox1, sizeof (mailbox1));
  msg = alloc();
  /* set message content here */
  os_mbx_send (mailbox1, msg);
   ..
}
```

- C-Module with a **mailbox1** reference:

```

#include <RTL.h>

extern OS_MBX mailbox1;

__task void task2 (void) {
  void *msg;
   ..
  os_mbx_wait (mailbox1, &msg, 0xffff);
  /* process message content here */
  free (msg);
   ..
}
```

Using a Mailbox

The RTX kernel message objects are simply pointers to a block of memory where the relevant information is stored. There is no restriction regarding the message size or content. The RTX kernel handles only the pointer to this message.

Sending 8-bit, 16-bit, and 32-bit values

Because the RTX kernel passes only the pointer from the sending task to the receiving task, we can use the pointer itself to carry simple information like passing a character from a serial receive interrupt routine. An example can be found in the **serial.c** interrupt driven serial interface module for the **Traffic example**. You must cast the char to a pointer like in the following example:

```
  os_mbx_send (send_mbx, (void *)c, 0xffff);
```

Sending fixed size messages

To send fixed size messages, you must allocate a block of memory from the dynamic memory pool, store the information in it, and pass its **pointer** to a mailbox. The receiving task receives the pointer and restores the original information from the memory block, and then releases the allocated memory block.

Fixed Memory block memory allocation functions

RTX has very powerful **fixed memory block** memory allocation routines. They are **thread safe** and fully **reentrant**. They can be used with the RTX kernel with no restriction. It is better to use the fixed memory block allocation routines for sending fixed size messages. The memory pool needs to be properly initialized to the size of message objects:

- **32-bit values**: initialize to 4-byte block size.
-
-
  ```
    _init_box (mpool, sizeof(mpool), 4);
  ```
- **any size messages**: initialize to the size of message object.
-
-
  ```
    _init_box (mpool, sizeof(mpool), sizeof(struct message));
  ```

For 8-bit and 16-bit messages, it is better to use a parameter casting and convert a message value directly to a pointer.

The following example shows you how to send fixed size messages to a mailbox (see the mailbox example for more information). The message size is 8 bytes (two unsigned ints).

```
#include <rtl.h>

os_mbx_declare (MsgBox, 16);                  /* Declare an RTX mailbox */
U32 mpool[16*(2*sizeof(U32))/4 + 3];          /* Reserve a memory for 16
messages */

__task void rec_task (void);

__task void send_task (void) {
  /* This task will send a message. */
  U32 *mptr;

  os_tsk_create (rec_task, 0);
  os_mbx_init (MsgBox, sizeof(MsgBox));
  mptr = _alloc_box (mpool);                  /* Allocate a memory for the
message */
  mptr[0] = 0x3215fedc;                       /* Set the message content. */
```

```
  mptr[1] = 0x00000015;
  os_mbx_send (MsgBox, mptr, 0xffff);        /* Send a message to a 'MsgBox' */
  os_tsk_delete_self ();
}

__task void rec_task (void) {
  /* This task will receive a message. */
  U32 *rptr, rec_val[2];

  os_mbx_wait (MsgBox, &rptr, 0xffff);       /* Wait for the message to arrive.
*/
  rec_val[0] = rptr[0];                      /* Store the content to 'rec_val'
*/
  rec_val[1] = rptr[1];
  _free_box (mpool, rptr);                   /* Release the memory block */
  os_tsk_delete_self ();
}

void main (void) {
  _init_box (mpool, sizeof(mpool), sizeof(U32));
  os_sys_init(send_task);
}
```

Sending variable size messages

To send a message object of variable size, you must use the memory allocation functions for the variable size memory blocks. The RVCT library provides these functions in **stdlib.h**

note

- The **fixed block** memory allocation functions are **fully reentrant**. The **variable length** memory allocation functions are **not reentrant**. Therefore the system timer interrupts need to be disabled during the execution of the malloc() or free() function. Function **tsk_lock()** disables timer interrupts and function **tsk_unlock()** enables timer interrupts.

SWI Functions

Software Interrupt (SWI) functions are functions that run in Supervisor Mode of **ARM7™** and **ARM9™** core and are **interrupt protected**. SWI functions can accept arguments and can return values. They are used in the same way as other functions.

The difference is hidden to the user and is handled by the RealView C-compiler. It generates different code instructions to call SWI functions. SWI functions are called by executing the SWI instruction. When executing the SWI instruction, the controller changes the running mode to a Supervisor Mode and blocks any further IRQ interrupt requests. Note that the FIQ interrupts are not disabled in this mode. When the ARM controller leaves this mode, interrupts are enabled again.

If you want to use SWI functions in your RTX kernel project, you need to:

1. Copy the **SWI_Table.s** file to your project folder and include it into your project. This file is located in the **\Keil\ARM\RL\RTX\SRC\ARM** folder.
2. Declare a function with a **__swi(*x*)** attribute. Use the first SWI number, starting from 8, that is free.

```
void __swi(8)  inc_5bit (U32 *cp);
```

5. Write a function implementation and convert the function name into a **__SWI_*x*** function name. This name is referenced later by the linker from **SWI_Table.s** module.

```
void __SWI_8            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  *cp = (*cp + 1) & 0x1F;
}
```

11. Add the function **__SWI_*x*** to the SWI function table in the **SWI_Table.s** module.

First import it from other modules:

```
; Import user SWI functions here.
                IMPORT  __SWI_8
```

then add a reference to it into the table:

```
; Insert user SWI functions here. SWI 0..7 are used by RTL Kernel.
                DCD     __SWI_8                 ; SWI 8  User Function
```

15. Your **SWI** function should now look like this:

```
void __swi(8)  inc_5bit (U32 *cp);
void __SWI_8            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  *cp = (*cp + 1) & 0x1F;
}
```

- SWI functions **0..7** are **reserved** for the RTX kernel.
- Do not leave gaps when numbering SWI functions. They must occupy a **continuous** range of numbers starting from 8.
- SWI functions can still be interrupted by **FIQ** interrupts.

SVC Functions

Software Interrupt (SVC) functions are functions that run in Privileged Handler Mode of **Cortex™-M** core. SVC functions can accept arguments and can return values. They are used in the same way as other functions.

The difference is hidden to the user and is handled by the RealView C-compiler. It generates different code instructions to call SVC functions. SVC functions are called by executing the SVC instruction. When executing the SVC instruction, the controller changes the running mode to a Privileged Handler Mode.

Interrupts are **not disabled** in this mode. In order to protect SVC function from interrupts, you need to include the disable/enable intrinsic functions **__disable_irq()** and **__enable_irq()** in your code.

You may use SVC functions to access **protected peripherals**, for example to configure NVIC and interrupts. This is required if you run tasks in **unprivileged** (protected) mode and you need to change interrupts from the task.

If you want to use SVC functions in your RTX kernel project, you need to:

1. Copy the **SVC_Table.s** file to your project folder and include it into your project. This file is located in the **\Keil\ARM\RL\RTX\SRC\CM** folder.
2. Declare a function with a **__svc(*x*)** attribute. Use the first SVC number, starting from 1, that is free.

```
void __svc(1)  inc_5bit (U32 *cp);
```

5. Write a function implementation and convert the function name into a **__SVC_*x*** function name. This name is referenced later by the linker from **SVC_Table.s** module. You need also to disable/enable interrupts.

```
void __SVC_1            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  __disable_irq();
  *cp = (*cp + 1) & 0x1F;
  __enable_irq();
}
```

13. Add the function **__SVC_*x*** to the SVC function table in the **SVC_Table.s** module.

    First import it from other modules:

```
; Import user SVC functions here.
                IMPORT  __SVC_1
```

then add a reference to it into the table:

```
; Insert user SVC functions here. SVC 0 used by RTL Kernel.
                DCD     __SVC_1                 ; user SVC function
```

17. Your **SVC** function should now look like this:

```
void __svc(1)  inc_5bit (U32 *cp);
void __SVC_1            (U32 *cp) {
  /* A protected function to increment a 5-bit counter. */
  __disable_irq();
  *cp = (*cp + 1) & 0x1F;
  __enable_irq();
}
```

- SVC function **0** is **reserved** for the RTX kernel.
- Do not leave gaps when numbering SVC functions. They must occupy a **continuous** range of numbers starting from 1.
- SVC functions can still be interrupted.

- RTX must be **initialized** before SVC functions are called.

Debugging

The µVision Simulator allows you to run and test your RTX kernel applications. RTX kernel applications load just like non-RTX programs. No special commands or options are required for debugging.

A **kernel-aware** dialog displays all aspects of the **RTX kernel** and the tasks in your program. The simulator can be used also with your target hardware, if you are using a ULINK JTAG interface on your target, to debug your application.

Note

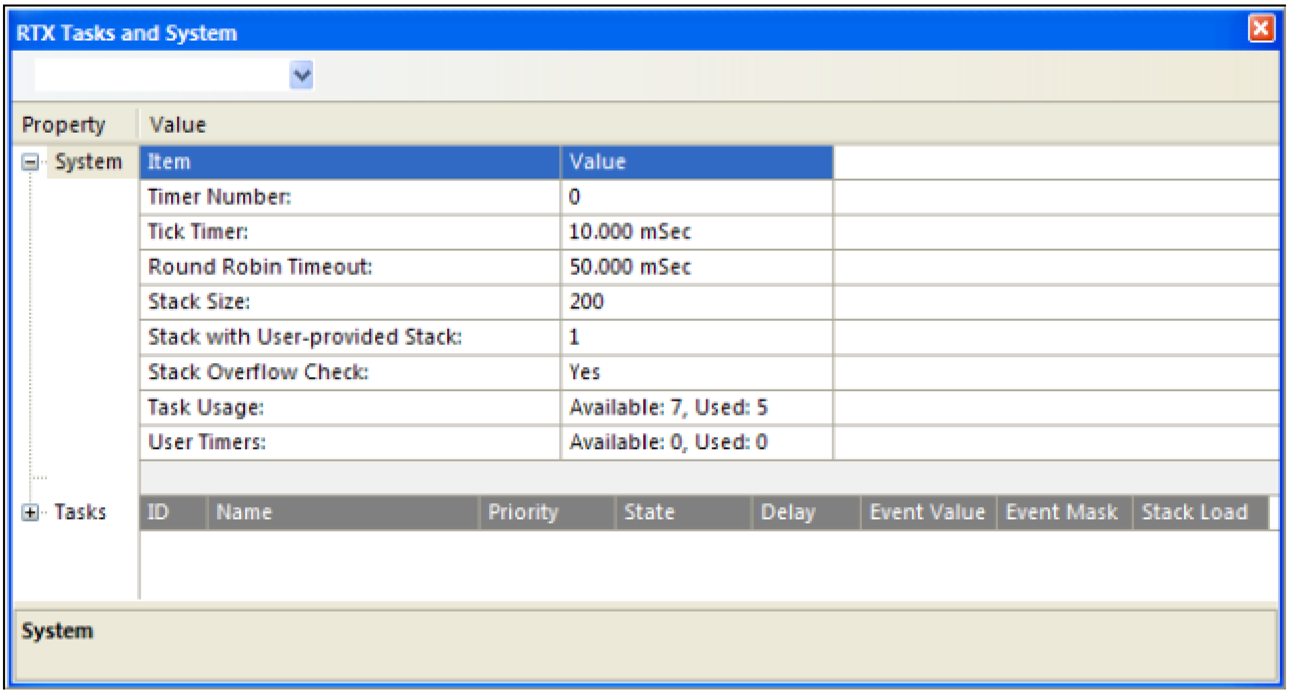
- You can have **source level debugging** if you enter the following SET variable into the debugger:

```

SET SRC=C:\Keil\ARM\RL\RTX\SRC
```

System Info

General information about the system resources and task usage is displayed by expanding the **System** property in the **RTX Tasks and System** dialog. You can use it to optimize your RTX application.

Select **RTX Tasks and System** from the **OS Support** item in the **Debug** menu to display this dialog.



Item	Value
Timer Number:	0
Tick Timer:	10.000 mSec
Round Robin Timeout:	50.000 mSec
Stack Size:	200
Stack with User-provided Stack:	1
Stack Overflow Check:	Yes
Task Usage:	Available: 7, Used: 5
User Timers:	Available: 0, Used: 0

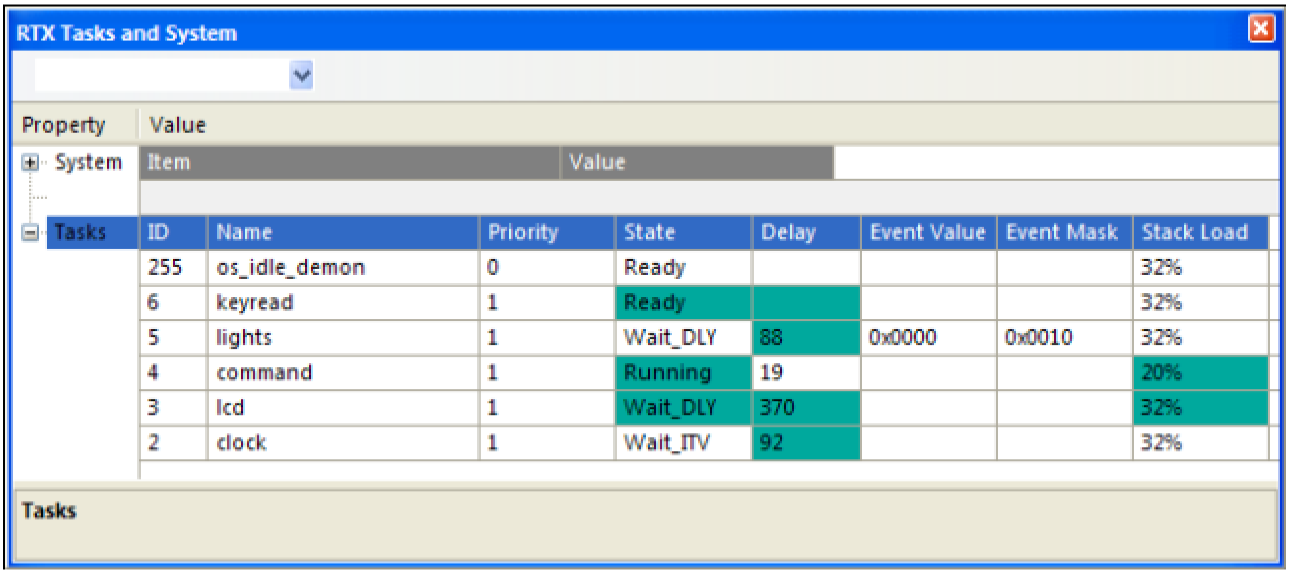
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load

System

Task Info

Detailed information about each running task is displayed when you expand the **Tasks** property in the **RTX Tasks and System** dialog. Note that one task can run in multiple instances. All active tasks are listed in this dialog.

Select **RTX Tasks and System** from the **OS Support** item in the **Debug** menu to display this dialog.



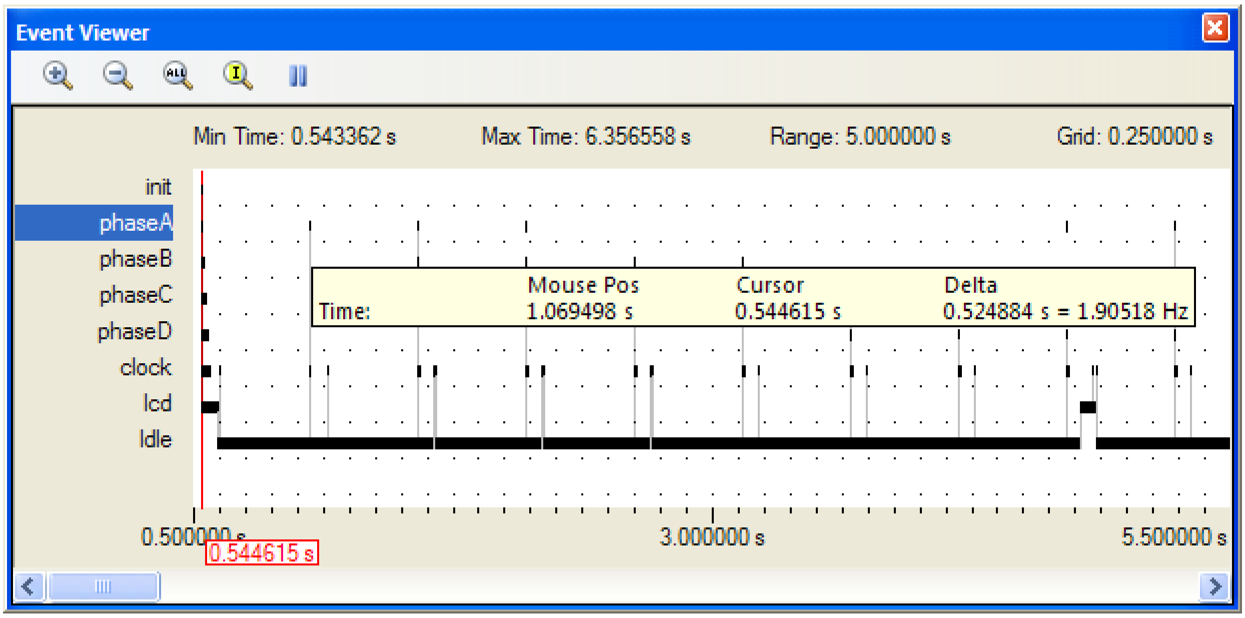
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				32%
6	keyread	1	Ready				32%
5	lights	1	Wait_DLY	88	0x0000	0x0010	32%
4	command	1	Running	19			20%
3	lcd	1	Wait_DLY	370			32%
2	clock	1	Wait_ITV	92			32%

- **ID** is the Task Identification Value assigned when the task was started.
- **Name** is the name of the task function.
- **Priority** is the current task priority.
- **State** is the current state of the task.
- **Delay** is the delay timeout value for the task.
- **Event Value** specifies the event flags set for the task.
- **Event Mask** specifies the event flags mask for the events that the task is waiting for.
- **Stack Load** specifies the usage of the task's stack.

Event Viewer

The **Event Viewer** dialog displays a chronological view of each running task allowing you to examine when tasks executed, and for how long, relative to all other tasks.

Select **Event Viewer** from the **OS Support** item in the **Debug** menu to display this dialog.

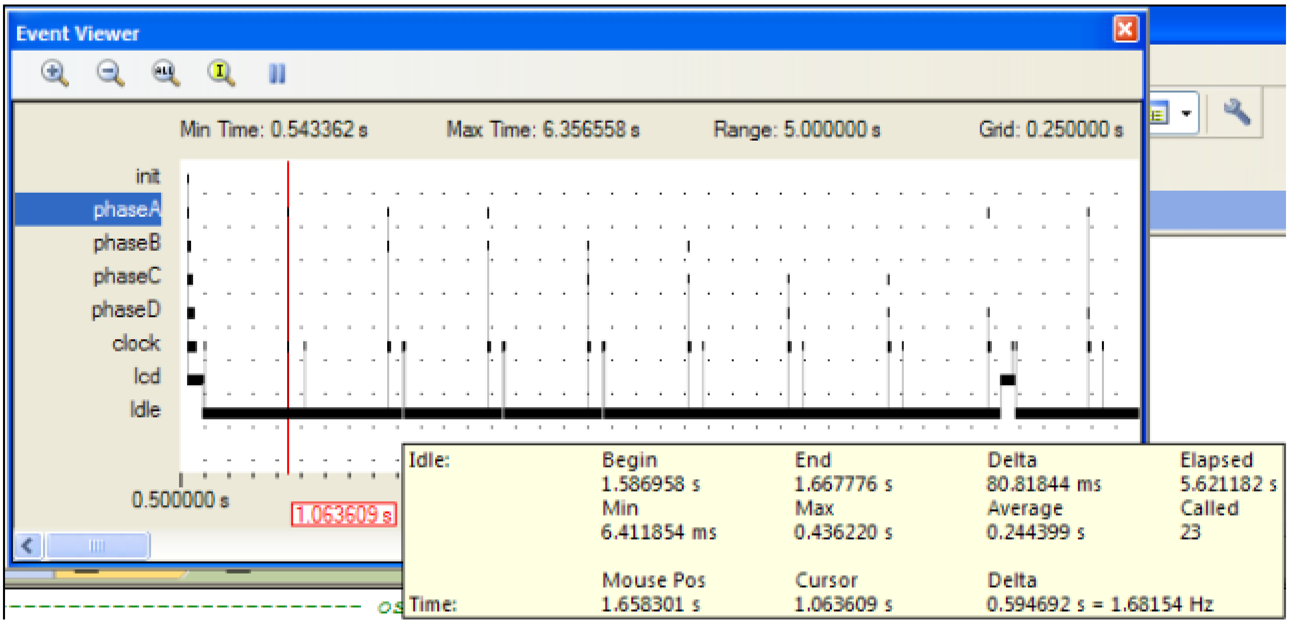


Min Time: 0.543362 s Max Time: 6.356558 s Range: 5.000000 s Grid: 0.250000 s

	Mouse Pos	Cursor	Delta
Time:	1.069498 s	0.544615 s	0.524884 s = 1.90518 Hz

Note that when a cursor (red vertical line) is placed by clicking on the display, precise timing between task events displays when the mouse cursor hovers over an event. Clicking an event name allows the left and right arrow keys to move the cursor forward and backward in time for that event.

Hovering the mouse over the **Idle** task displays minimum, maximum and average elapsed times for the Idle task and the number of time the Idle task was called.



Min Time: 0.543362 s Max Time: 6.356558 s Range: 5.000000 s Grid: 0.250000 s

	Begin	End	Delta	Elapsed
Idle:	1.586958 s	1.667776 s	80.81844 ms	5.621182 s
	Min	Max	Average	Called
	6.411854 ms	0.436220 s	0.244399 s	23
	Mouse Pos	Cursor	Delta	
Time:	1.658301 s	1.063609 s	0.594692 s = 1.68154 Hz	

Usage Hints

Here are a few hints to help you if you run into problems when using the RTX kernel.

Function usage

- Functions that begin with **os_** can be called from a task but not from an interrupt service routine.
- Functions that begin with **isr_** can be called from an **IRQ** interrupt service routine but not from a task.
- Never call **isr_** functions from **FIQ** (ARM7™, ARM9™) interrupt functions or from the task.
- Never call **tsk_lock()** or **tsk_unlock()** from an interrupt function.
- Before the kernel starts, never enable any **IRQ** interrupt that calls **isr_** functions.

Because of a two different implementations of RTX Kernel Library further hints depend on the Library version being used:

- **ARM7™/ARM9™ Version**
- **Cortex™-M Version**

ARM Version

Here are a few hints specific for ARM7™/ARM9™ library version.

Using IRQ interrupts

You can use **IRQ** interrupts with no limitation. RTX kernel uses only one timer interrupt to generate periodic **timer ticks** and activate the task scheduler.

- **IRQ** interrupts are disabled by RTX for a very short time (a few µseconds maximum).
- RTX kernel uses **Software Interrupts** to protect a critical code section from interrupts.
- Software interrupts **0-7** are used by RTX and cannot be used in your application.
- RTX uses its own **SWI Handler** which is automatically linked from the library. If you include another SWI handler (like that found in the **SWI.S** file) into your project, RTX could fail. Remove any user-created SWI handler from your project to resolve the Data Abort.
- Check the **IRQ stack size** configured from the startup file if you see sporadic crashes of your application. The IRQ stack usage depends on the complexity of your additional interrupt functions.

Using FIQ interrupts

ARM7™/ARM9™ Fast Interrupts are not used by the RTX kernel. You may freely use **FIQ** interrupts in you application in parallel with the kernel.

- **FIQ** interrupts are **never disabled** by RTX.
- You cannot call any kernel system function from the FIQ Interrupt Handler.

System Startup

RTX kernel uses a separate stack for each task it creates. The stack size is **configured** in the configuration file. However, before the kernel is started by the **os_sys_init()** function, the stack that is configured in the startup file **STARTUP.S** for the User Mode is used. When the RTX kernel is up and running, the User Mode stack is used for the task manager - an RTX task scheduler.

Minimum stack sizes for RTX kernel configured in **STARTUP.S** are:

- Supervisor Mode **32 bytes** (0x00000020)
- Interrupt Mode **64 bytes** (0x00000040)
- User Mode **80 bytes** (0x00000050)

Supervisor Mode stack is used when **SWI functions** are called. If you are using your own complex **__swi** functions, you might also need to increase the size of this stack.

Interrupt Mode stack is used on timer tick interrupts. This interrupt activates the system task scheduler. The scheduler uses the User/System Mode stack defined in **STARTUP.S** and runs in System Mode. If you are using interrupts in your application, you should increase the size of the Interrupt Mode stack. A stack size of 256 bytes is a good choice for a start. If the interrupt stack overflows, the application might crash.

User Mode stack is used until the kernel is started. It is better to initialize the user application from the first task which is created and started by the **os_sys_init()** function call.

You can initialize **simple IO**, like configure the port pins and enable AD converter, before the **os_sys_init()** function is called. The init_IO() function must be small and must not use many local variables because the User Mode stack can overflow otherwise.

```
void main (void) {
  /* Here a simple IO may be initialized. */
  init_IO ();
  os_sys_init (task1);
  for (;;);
}
```

It is better to do a **complex initialization** from the first task that starts. In this case, the stack for this task is used, which is in general much bigger than User Mode stack.

```
__task void task1 (void) {
  /* Here the interrupts and more complex IO may be initialized. */
  Init_CAN ();
   ..
}
```

Cortex Version

Here are a few hints specific for Cortex™-M library version.

Using IRQ interrupts

You can use **IRQ** interrupts with no limitation. RTX kernel uses only one timer interrupt to generate periodic **timer ticks** and activate the task scheduler.

- **IRQ** interrupts are never disabled by RTX Kernel.
- Software interrupt **0** is used by RTX and cannot be used in your application.
- RTX uses its own **SVC Handler** which is automatically linked from the library. If you include another SVC handler (like that found in the **SVC.S** file) into your project, RTX could fail. Remove any user-created SVC handler from your project to resolve the Hard Fault.
- Do not change default interrupt **priority grouping** (PRIGROUP = 0) in NVIC.
- Check the **Main Stack size** configured from the startup file if you see sporadic crashes of your application. The RTX Kernel for Cortex™-M is implemented as a System Service Calls. All SVC calls use a Main Stack.

System Startup

RTX kernel uses a separate stack for each task it creates. The stack size is **configured** in the configuration file. However, before the kernel is started by the **os_sys_init()** function, the stack that is configured in the startup file **STARTUP.S** for the Main Stack is used.

Stack size used by RTX kernel is configured in **STARTUP.S**. Minimum size is **128 bytes**, however **256 bytes** is recommended when interrupts are used.

Main stack is also used when **SVC functions** are called. If you are using your own complex **__svc** functions, you might also need to increase the size of this stack.

You can initialize **simple IO**, like configure the port pins and enable AD converter, enable interrupts, before the **os_sys_init()** function is called. The init_IO() function is executed in **privileged** mode. It is recommended to configure peripherals in this function and use **unprivileged** mode for the tasks.

```
void main (void) {
  /* Here a simple IO may be initialized. */
  init_IO ();
  os_sys_init (task1);
  for (;;);
}
```

Create New RTX Application

This section describes how to create a new application that uses the RTX kernel.

1. First, **create a new project** in the µVision IDE by selecting **Project —> New Project**.
2. In the *Create New Project* window, select a new directory for your project and enter a name for your project.
3. In the *Select Device for Target* window, select your target ARM device and click OK. Allow µVision to copy and add the device startup file to your project. This creates a basic µVision project.
4. Now **setup the project to use the RTX kernel**. To do this, select **Project —> Options for Target**. Then select RTX Kernel for the **Operating system** and click OK.
5. Copy the RTX_Config.c configuration file for your target device from the **\Keil\ARM\Startup\** directory. If the file does not exist for your specific device, then copy the file from the Philips folder and modify it to suit your device.
6. Modify the device startup file to enable SWI_Handler function:
   - Comment out the following line from the startup file:

     ```
     SWI_Handler     B     SWI_Handler
     ```
   - Add the following line to the startup file:

     ```
     IMPORT     SWI_Handler
     ```

   This change prevents the code from sitting in a loop when a SWI interrupt occurs. The change allows the right function to run when a SWI interrupt occurs.
7. Copy the **retarget.c** file to your project directory, and add it to your project. The main purpose of this file is to avoid the use of semihosting SWIs. Thus the file must contain the following:

```
8.
9.  #include <rt_misc.h>
10.
11. #pragma import(__use_no_semihosting_swi)
12.
13. void _ttywrch(int ch) {
14.   // Not used (No Output)
15. }
16.
17. void _sys_exit(int return_code) {
18. label:  goto label;  /* endless loop */
19. }
```

Depending on your application, you might have to retarget more functions. For example if you use the RL-FlashFS library, you can obtain retarget.c from the **\Keil\ARM\RL\FlashFS\SRC\** directory. Now the project is setup to use the RTX kernel.

- For **MicroLIB** runtime library you do not need a **retarget.c** in your project.

20. Now you must **configure the RTX kernel** for the needs of your application by making the required changes in the **RTX_Config.c** file.
21. **Create the application source files** if they do not already exist. Add these source files to the project. You can do this in the project workspace of µVision by right clicking on the Source Group and selecting **Add Files to Group**.
22. Build your application using **Project —> Build Target**.
23. If you project builds successfully, you can download it to your hardware or run it using the µVision Simulator. You can also debug the application using **Debug —> Start Debug Session**.

Function Overview

This section serves as a programmer's reference. It describes all system call functions in details. The functions are ordered according to the following categories:

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management

The system functions description is divided into several sections:

Summary:	Briefly describes the routine's effect, lists include file(s) containing its declaration and prototype, illustrates the syntax, and describes any arguments.
Description:	Provides a detailed description of the routine and how it is used.
Return Value:	Describes the value returned by the routine.
See Also:	Names related routines.
Example:	Gives a function or program fragment demonstrating proper use of the function.

Event Flag Management Routines

Routine	Attributes	Description
os_evt_clr		Clears one or more event flags of a task.
os_evt_get		Retrieves the event flags that caused **os_evt_wait_or** to complete.
os_evt_set		Sets one or more event flags of a task.
os_evt_wait_and		Waits for one or more event flags to be set.
os_evt_wait_or		Waits for any one event flag to be set.
isr_evt_set		Sets one or more event flags of a task.

note

- The event flag management routines enable you to send and wait for events from the other tasks.

Mailbox Management Routines

Routine	Attributes	Description
os_mbx_check		Determines the number of messages that can still be added to the mailbox.
os_mbx_declare		Creates a mailbox object.
os_mbx_init		Initializes a mailbox so that it can be used.
os_mbx_send		Sends a message to a mailbox.
os_mbx_wait		Gets the next message from a mailbox, or waits if the mailbox is empty.
isr_mbx_check		Determines the number of messages that can still be added to the mailbox.
isr_mbx_receive		Gets the next message from a mailbox.
isr_mbx_send		Sends a message to a mailbox.

note

- The mailbox management routines enable you to send and receive messages between tasks using mailboxes.
- The **os_mbx_declare** routine is implemented as a macro.

Memory Allocation Routines

Routine	Attributes	Description
_declare_box		Creates a memory pool of fixed size blocks with 4-byte alignment.
_declare_box8		Creates a memory pool of fixed size blocks with 8-byte alignment.
_init_box		Initializes a memory pool with 4-byte aligned blocks.
_init_box8		Initializes a memory pool with 8-byte aligned blocks.
_alloc_box	Reentrant	Allocates a memory block from a memory pool.
_calloc_box	Reentrant	Allocates a memory block from a memory pool, and clears the contents of the block to 0.
_free_box	Reentrant	Returns a memory block back to its memory pool.

note

- The memory allocation routines enable you to use the system memory dynamically by creating memory pools and using fixed size blocks from the memory pools.
- The **_init_box8**, **_declare_box** and **_declare_box8** routines are implemented as macros.

Mutex Management Routines

Routine	Attributes	Description
os_mut_init		Initializes a mutex object.
os_mut_release		Releases a mutex object.
os_mut_wait		Waits for a mutex object to become available.

note

- The mutex management routines enable you to use mutexes to synchronize the activities of the various tasks and to protect shared variables from corruption.
- The **priority inheritance** method is used in mutex management routines to eliminate **priority inversion** problems.

Semaphore Management Routines

Routine	Attributes	Description
os_sem_init		Initializes a semaphore object.
os_sem_send		Sends a signal (token) to the semaphore.
os_sem_wait		Waits for a signal (token) from the semaphore.
isr_sem_send		Sends a signal (token) to the semaphore.

note

- The semaphore management routines enable you to use semaphores to synchronize the activities of the various tasks and to protect shared variables from corruption.

System Functions

Routine	Attributes	Description
tsk_lock		Disables task switching.
tsk_unlock		Enables task switching.

note

- The system functions enable you to control the timer interrupt and task switching.

Task Management Routines

Routine	Attributes	Description
os_sys_init		Initializes and starts RL-RTX.
os_sys_init_prio		Initializes and starts RL-RTX assigning a priority to the starting task.
os_sys_init_user		Initializes and starts RL-RTX assigning a priority and custom stack to the starting task.
os_tsk_create		Creates and starts a new task.
os_tsk_create_ex		Creates, starts, and passes an argument pointer to a new task.
os_tsk_create_user		Creates, starts, and assigns a custom stack to a new task.
os_tsk_create_user_ex		Creates, starts, assigns a custom stack, and passes an argument pointer to a new task.
os_tsk_delete		Stops and deletes a task.
os_tsk_delete_self		Stops and deletes the currently running task.
os_tsk_pass		Passes control to the next task of the same priority.
os_tsk_prio		Changes a task's priority.
os_tsk_prio_self		Changes the currently running task's priority.
os_tsk_self		Obtains the task ID of the currently running task.
isr_tsk_get		Obtains the task ID of the interrupted task.

note

- The task management routines enable you to start the RTX kernel, create and delete various types of tasks, and control their execution priorities.

Time Management Routines

Routine	Attributes	Description
os_dly_wait		Pauses the calling task for a specified interval.
os_itv_set		Enables the calling task for periodic wake up.
os_itv_wait		Pauses the calling task until the periodic wake up interval expires.

note

- The time management routines enable you to pause and restart tasks using a timer.

User Timer Management Routines

Routine	Attributes	Description
os_tmr_create		Starts a countdown timer to call the **os_tmr_call** function.
os_tmr_kill		Aborts a user defined timer.
os_tmr_call		User customizable function that gets called when the user defined timer expires.

note

- The **user timer** management routines enable you to use timers to control when a user customizable function runs.

RL-FlashFS

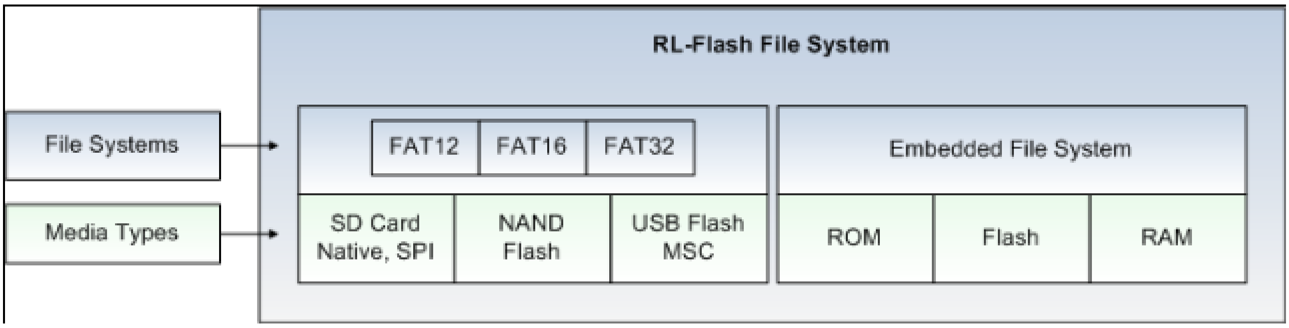
RL-Flash File System (RL-FlashFS) is a software library that provides a common API to create, save, read, and modify files on a Flash device. The library offers interface functions and handles the low level file input and output operations. Developer can focus on the application needs rather than concerning about the implemented file system. **RL-FlashFS** works with several ARM- and Cortex-M processor-based devices, and can be used standalone or with the RTX-RTOS.

RL-FlashFS supports several media types, such as standard Secure Digital (SD), Secure Digital High Capacity (SDHC), Multi Media Card (MMC), and Flash Memory Cards. The media are used in **SD Native mode** or in **SPI mode**. Standard device sizes ranging from a few MBytes up to 32 GBytes are supported. Files on Memory Cards are stored in the standard **FAT12**, **FAT16**, or **FAT32** file format. **RL-FlashFS** supports huge Flash sizes where Flash ROMs typically have several 64KB pages.

The file system depends on the memory device type used in the application. The following file systems are supported:

- **FAT File System**
  Supports the FAT12, FAT16, and FAT32 file system. This file system is used for memory card devices (SD Cards), USB Memory sticks, and NAND Flash devices.
- **Embedded File System**
  Is optimized for low density Flash devices (NOR Flash), SPI Flash, and RAM devices. This file system is not FAT-compatible and cannot be used as USB mass storage device.

The picture relates the media type to the file system.



RL-FlashFS applications are written using standard C constructs and are compiled with the ARM RealView® Compiler. To write applications using RL-FlashFS, link the source files to the project.

The following topics are included:

Embedded File System

Provides information about the memory organization, file allocation, and file data block usage in the Embedded File System.

Technical Data

Provides an overview about the RL-FlashFS performance on various boards.

Configuring RL-FlashFS

Explains the configuration options for the supported media types.

Using RL-FlashFS

Explains how to create applications using RL-FlashFS.

Function Overview

Describes the functions and routines provided by RL-FlashFS.

Embedded File System

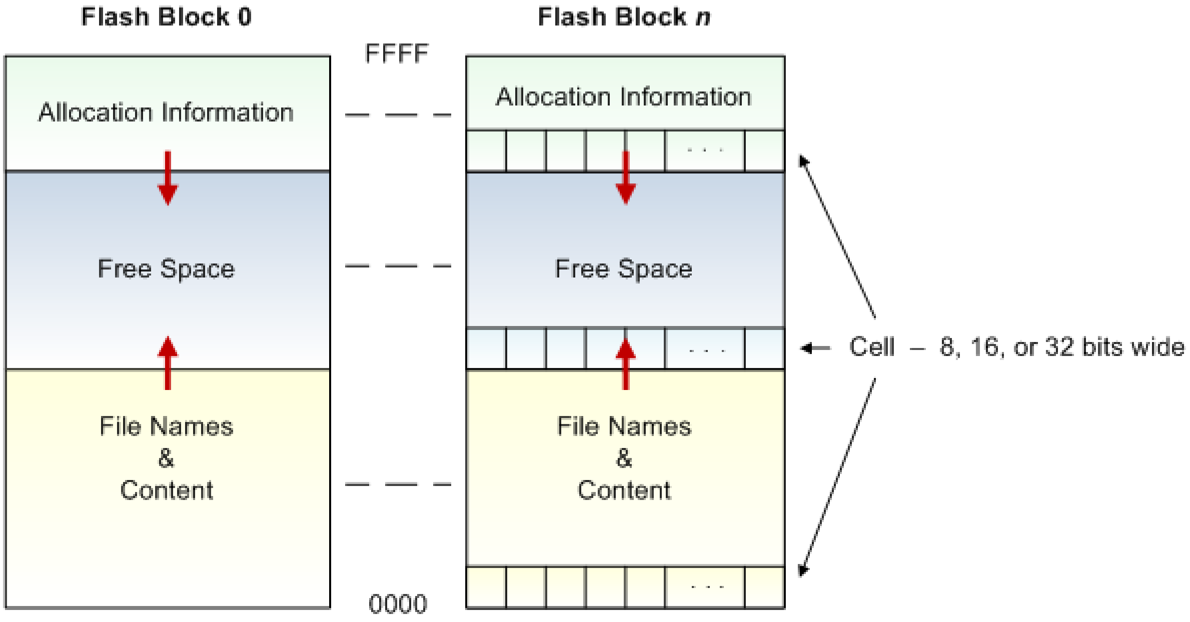
Embedded File System (EFS) describes the proprietary file system used in **RL-FlashFS**.

- Memory Organization of the Flash Device is optimized for maximum performance.
- Allocation Information are reduced to a minimum allowing small data overhead.
- File Data Fragments are of variable size and provide optimal file access times.

Memory Organization

The **Memory Organization** of a Flash Device is divided into Flash sectors. Flash sectors are named **blocks** in **RL-FlashFS**. Typically, a blocks is a 64 KB memory page. Blocks can be devided into memory cells, which are written sequencially. The memory cell size depends on the device architecture and is 8- (byte), 16- (half word) or 32-bit wide (word).

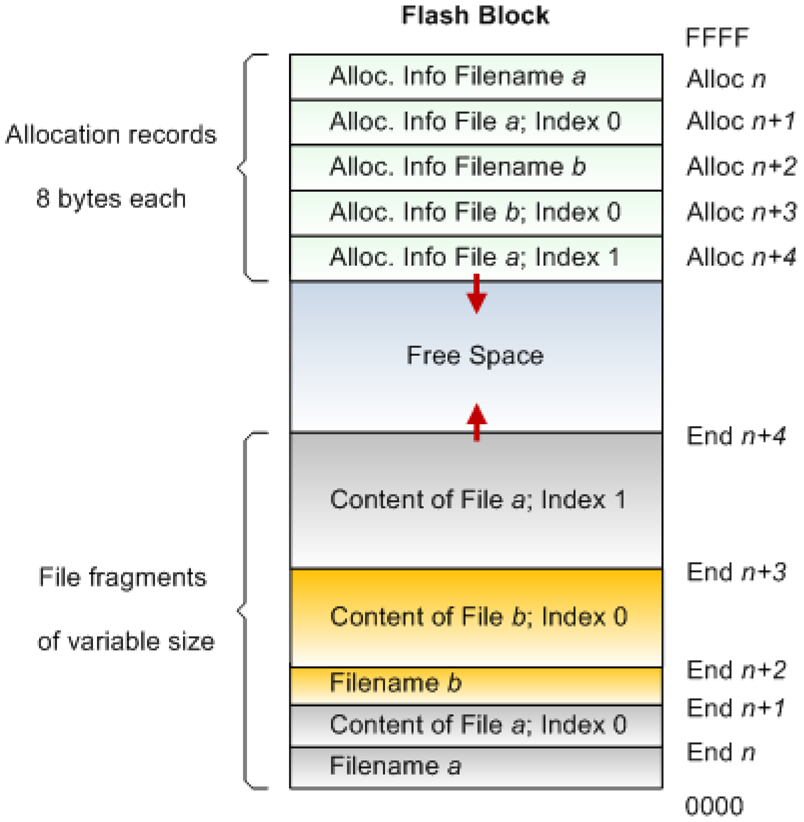
Each Block contains its own allocation information written to the file allocation table located on top of memory. The file name and file content are stored in lower memory regions. If the file size exceeds a single block, then the file is stored across several blocks. Several smaller files are stored into a single block.



When the file content is modified, the old file content is **invalidated** and a new memory block is allocated. The Flash Block is erased when all the data stored in the Flash Block have been invalidated.

Allocation Information

Allocation Information is stored at the top of a Flash Block and is written in **descending** order. Each allocation information record consists of 8 bytes. Each file fragment has its own allocation record. The first file fragment starts in Flash Block at offset 0. It is always assumed that the first file block starts at the beginning of a Flash Block.



The file allocation information record has the following components:

- **end** is the end address of the file fragment.
- **fileID** is the file identification number and is associated with the file name.
- **index** is the file fragment ordering number, which starts at 0 for each file.

```
struct falloc {
  U32 end;
  U16 fileID;
  U16 index;
};
```

The file allocation information is written when:

- The file is **opened** for writing and RL-FlashFS creates a Filename information record.
- The file is **closed** and the file handle released.
- The file is **flushed** and the number of bytes from a file buffer is not a multiple of 4.
- The Flash Block is **full** and there is no more free space.

File Data Fragments

File Data Fragments are of **variable size** and are fully defined through the file allocation information record.

To make optimum use of the Flash Block, create **big file fragments** to reduce the total number of file fragments. It is not optimal to open a file for appending or writing a byte to it and close the file. This approach creates huge file allocation information records, which consumes 12 bytes of Flash in total; 8 bytes for the file allocation information and 4 bytes for the information. In addition, such an approach creates slow access times to files.

Technical Data

The following table gives an overview about the RTX Flash File System performance on different Evaluation Boards.

Board	Device	CPU Core	CPU Speed [MHz]	Card Interface	Write [KB/s]	Read [KB/s]
AT91SAM9260-EK	Atmel AT91SAM9160	ARM9	96.1	SD4 at 25MHz	4785.0	5044.0
AT91SAM9261-EK	Atmel AT91SAM9161	ARM9	96.1	SD4 at 25MHz	4790.6	5069.3
AT91SAM9G20-EK	Atmel AT91SAM9G20	ARM9	99.1	SD4 at 25MHz	4899.5	5418.0
AT91SAM9RL-EK	Atmel AT91SAM9RL64	ARM9	100.0	SD4 at 25MHz	4096.0	5211.2
MCB2400	NXP LPC2468	ARM7	48.0	SD4 at 24MHz	4084.3	5525.9
MCB2300	NXP LPC2368	ARM7	48.0	SD4 at 24MHz	3946.3	5330.6
MCB2140	NXP LPC2148	ARM7	60.0	SPI at 7.5MHz	299.4	313.4
MCBSTR9	ST STR912	ARM9	48.0	SPI at 12MHz	355.2	357.1
MCBSTR750	ST STR750	ARM7	60.0	SPI at 15MHz	402.2	416.1
MCBSTM32	ST STM32	Cortex-M3	72.0	SPI at 18MHz	711.1	758.1
LM3S8962	Luminary LM3S8962	Cortex-M3	50.0	SPI at 12.5MHz	537.8	607.6
LM3S6965	Luminary LM3S6965	Cortex-M3	50.0	SPI at 12.5MHz	539.2	603.6
LM3S3768	Luminary LM3S3768	Cortex-M3	50.0	SPI at 12.5MHz	539.5	603.8

Performance test procedure

- **Initialize the Card for testing**:
  - Format the Card
  - Open the file "Test.txt"
  - Prewrite 4MB of file data with 0x55
  - Close and Delete the file
- **Write performance test**:
  - Open the file "Test.txt"
  Start the measurement timer
  - Write 4MB of data in 4KB blocks
  Stop the measurement timer
  - Close the file
- **Read performance test**:
  - Open the file "Test.txt"
  Start the measurement timer
  - Read 4MB of data in 4KB blocks
  Stop the measurement timer
  - Close the file
- **Test validity check**:
  - Open the file "Test.txt"
  - Verify 4MB of data in 4KB blocks
  - Close and Delete the file

Note

- SD Cards: SanDisk Extreme III (1GB) and Kingston (1GB) were used for testing.

Configuring RL-FlashFS

Configuring RL-FlashFS explains the configuration options and lists the library files needed to create an application for Flash devices.

Configuration

Explains the configuration options available for several media types.

Source Files

Lists the library files, include files, and locations.

Configuration

Configuration options are set in the file **File_Config.c**. Depending on the device type, include one of the following files to the project:

- Memory card drives, USB Flash drives, and NAND Flash drives need the ***system driver*** file, which is located in the folder **\ARM\RL\FlashFS\Driver**.
- Embedded Flash drives and SPI Flash drives need the ***Flash Programming Algorithm***, which is located in the folder **\ARM\RL\FlashFS\Flash**.

RL-FlashFS supports multiple volums. A drive letter can be assigned to each device type.

Device Type	Drive Letter
Flash	**F:**
Memory Card	**M0:** and **M1:**
NAND Flash	**N:**
RAM Device	**R:**
SPI Flash	**S:**
USB Flash	**U0:** and **U1:**

Configuration options in the file **File_Config.c** allow the developer to:

File System

- Specify **Number of open files**
- Specify **CPU Clock Frequency [Hz]**

The options are explained in File System.

Flash Drive

- Enable the Flash Drive
- Specify **Base Address** of the Flash Device
- Specify **Device Size**
- Specify **Content of Erased Memory**
- Specify **Device Description file**
- Enable **Default Drive [F:]**

The options are explained in Flash Drive.

SPI Flash Drive

- Enable the SPI Flash Drive
- Specify **Device Size**
- Specify **Content of Erased Memory**
- Specify **Device Description file**
- Enable **Default Drive [S:]**

The options are explained in SPI Flash Drive.

RAM Drive

- Enable Ram Drive
- Specify **Device Size**
- Specify **Number of virtual Sectors**
- Enable RAM Buffer Relocation
- Specify **Base address** of the RAM Buffer
- Enable **Default Drive [R:]**

The options are explained in RAM Drive.

Memory Card Drive 0

- Enable Memory Card Drive 0
- Specify **Bus Mode**
- Specify **File System Cache** size
- Enable Cache Buffer Relocation
- Specify **Base address** of the Cache Buffer
- Enable **Default Drive [M0:]**

The options are explained in Memory Card Drive.

Memory Card Drive 1

- Enable Memory Card Drive 1
- Specify **Bus Mode**
- Specify **File System Cache** size
- Enable Cache Buffer Relocation
- Specify **Base address** of the Cache Buffer
- Enable **Default Drive [M1:]**

The options are explained in Memory Card Drive.

USB Flash Drive 0

- Enable USB Flash Drive 0
- Specify **File System Cache** size
- Enable **Default Drive [U0:]**

The options are explained in USB Flash Drive.

USB Flash Drive 1

- Enable USB Flash Drive 1
- Specify **File System Cache** size
- Enable **Default Drive [U1:]**

The options are explained in USB Flash Drive.

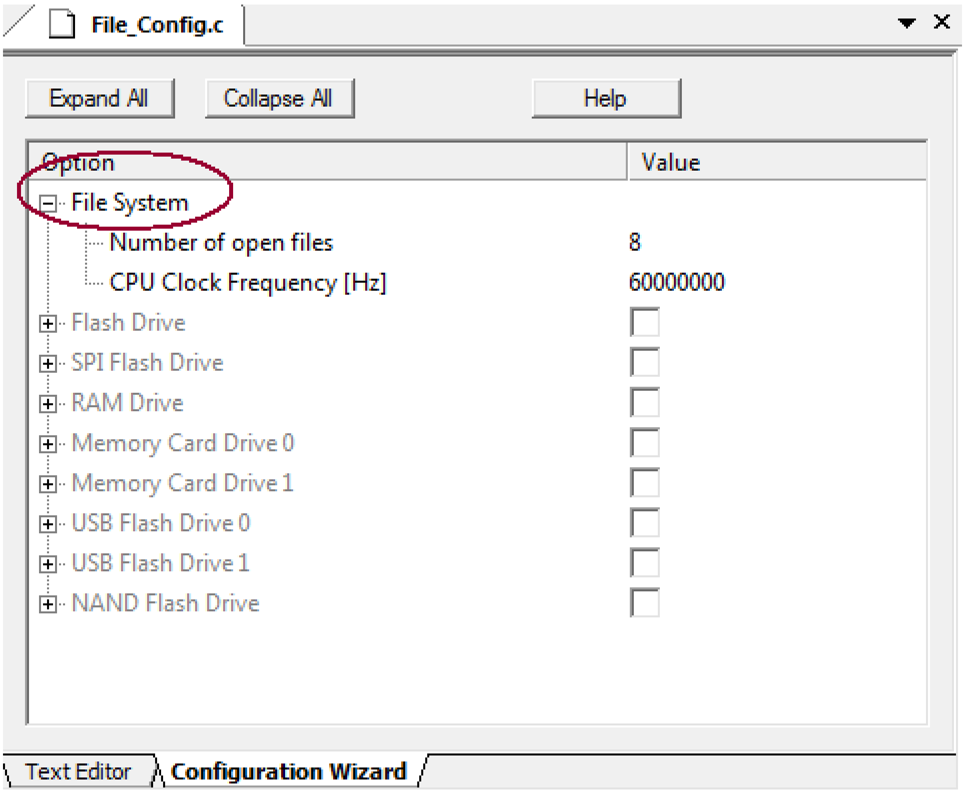
NAND Flash Drive

- Enable NAND Flash Drive
- Specify **Page size**
- Specify **Block Size**
- Specify **Device Size [blocks]**
- Specify **Page Caching**
- Specify **Block Indexing**
- Enable **Software ECC** (error correction code)
- Specify **File System Cache** size
- Enable Cache Buffer Relocation
- Specify **Base address** of the Cache Buffer
- Enable **Default Drive [N:]**

The options are explained in NAND Flash Drive.

File System

File System options allow configuring RL-FlashFS to open multiple files at the same time. Each opened file requires some memory resources for file buffering. Set the options manually or use the Configuration Wizard.



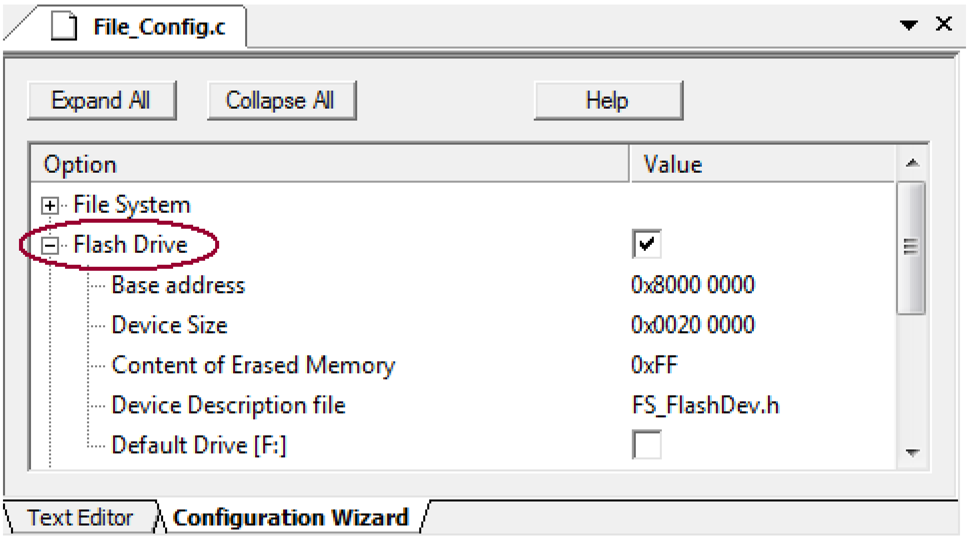
Option	Value
⊟ File System	
Number of open files	8
CPU Clock Frequency [Hz]	60000000
⊞ Flash Drive	□
⊞ SPI Flash Drive	□
⊞ RAM Drive	□
⊞ Memory Card Drive 0	□
⊞ Memory Card Drive 1	□
⊞ USB Flash Drive 0	□
⊞ USB Flash Drive 1	□
⊞ NAND Flash Drive	□

The following options can be configured:

- **Number of open files** specifies the number of files, that can be open at the same time.
-
- `#define N_FILES      8`
- **CPU Clock Frequency [Hz]** determines the CPU clock frequency value in Hz. Set this value equal to the Core Clock value.
-
- `#define CPU_CLK      60000000`

Flash Drive

Flash Drive options allow configuring RL-FlashFS to use Flash devices. Set the options manually or use the Configuration Wizard. Add also the Flash Driver and Sector Layout description files to the project.



Option	Value
⊞ File System	
⊟ Flash Drive	☑
Base address	0x8000 0000
Device Size	0x0020 0000
Content of Erased Memory	0xFF
Device Description file	FS_FlashDev.h
Default Drive [F:]	□

The following options can be configured:

- **Flash Drive** enables support for a Flash Drive. The Flash drive is not used when this option is disabled. It must be set to **1** to enable it and **0** to disable it.

```
#define FL0_EN       1
```

- **Base address** specifies the device base address in the memory space of the processor.

```
#define FL0_BADR     0x80000000
```

- **Device Size** specifies the size of the flash device to be used for storing files. Typically, this is the whole size of the flash device, specified in bytes. It is allowed to specify only a part of the device to be used for the Flash File System. The rest of the device might be used for the application code. In this case, the function **EraseChip** must not be provided in the driver control block. The value for the EraseChip function must be set to NULL.

```
#define FL0_SIZE     0x0200000
```

- **Content of Erased Memory** specifies the initial content of the erased Flash Device. In most cases, this value is set to **0xFF**. Accepted values are 0xFF or 0x00.

```
#define FL0_INITV    0xFF
```

- **Device Description file** specifies a file containing the Flash device sector layout description. The file is named FS_FlashDev.h and is tailored to a specific Flash device. Several description files are available in the folder **\ARM\RL\FlashFS\Flash**.

```
#define FL0_HFILE    "FS_FlashDev.h"
```

- **Default Drive [F:]** enables the Flash Drive as a default system drive. This drive is used, when a drive letter is not specified in a filename.

```
#define FL0_DEF      1
```

Sector Layout

The **Sector Layout** description file, **FS_FlashDev.h**, specifies the memory map of the device. Every device has its own description file, which are located in the folder **\ARM\RL\FlashFS\Flash**.

To generate a new description file, copy the flash sector layout information from the Flash Device datasheet. Specify a sector **size** in bytes and a sector **base address** relative to a Flash Device Start address. The macro **DSB** converts this information into the RL-FlashFS compatible sector description.

To improve the RL-FlashFS performance, the sector information is stored as a table in the code. The RL-FlashFS scans this table when accessing files from the Flash Device.

The following example shows a Flash Sector layout configuration description for the **Am29x800BT** Flash Device:

```
#define FLASH_DEVICE                                    \
  DFB(0x10000, 0x000000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x010000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x020000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x030000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x040000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x050000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x060000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x070000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x080000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x090000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0A0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0B0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0C0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0D0000),    /* Sector Size 64kB */    \
  DFB(0x04000, 0x0E0000),    /* Sector Size 16kB */    \
  DFB(0x08000, 0x0E4000),    /* Sector Size 32kB */    \
  DFB(0x02000, 0x0EC000),    /* Sector Size  8kB */    \
  DFB(0x02000, 0x0EE000),    /* Sector Size  8kB */    \
  DFB(0x02000, 0x0F0000),    /* Sector Size  8kB */    \
  DFB(0x02000, 0x0E2000),    /* Sector Size  8kB */    \
  DFB(0x08000, 0x0F4000),    /* Sector Size 32kB */    \
  DFB(0x04000, 0x0FC000),    /* Sector Size 16kB */    \

#define FL_NSECT    22
```

Note

- The RL-FlashFS does not require that specified Flash sectors be continuous. Gaps are allowed in the Device Memory space. The developer can reserve some Flash sectors for the application code. Reserved sectors are not used for storing files. Simply **do not include** such Flash sectors in the **FLASH_DEVICE** description table.
- It is not allowed to assign parts of a sector to the RL-FlashFS and the rest of that sector for some other usage. When such a sector is erased by the RL-FlashFS, the whole sector is erased, and not just a part of it.
- When a Flash Device usage is split between storing files and some other usage, do not provide the function **EraseChip** in the Flash driver.

Flash Driver

The **Flash driver** implements low level flash programming routines to interface the Embedded parallel Flash or SPI Data Flash memory. An interface to the RL-FlashFS is the flash driver control block.

Driver control block

The structure **EFS_DRV** is defined in File_Config.h as follows:

```
/* Embedded Flash Device driver */
typedef struct {
  BOOL (*Init)         (U32 adr, U32 clk);
  BOOL (*UnInit)       (void);
  BOOL (*ReadData)     (U32 adr, U32 sz, U8 *buf);   /* Optional, NULL for
memory-mapped Flash */
  BOOL (*ProgramPage)  (U32 adr, U32 sz, U8 *buf);
  BOOL (*EraseSector)  (U32 adr);
  BOOL (*EraseChip)    (void);                       /* Optional, NULL if not
existing    */
} const EFS_DRV;
```

The Flash driver uses six low-level user-provided functions to control the Flash device:

- **Init()**
  This function is called when the RL-FlashFS is initialized by the **finit** function.
- **UnInit()**
  This function is called to uninitialize the embedded flash drive by the **funinit** function.
- **ReadData()**
  This function is used to read data from a Flash Device.
  Not provided (NULL) for parallel memory-mapped flash device.
- **ProgramPage()**
  This function is used to program data into a Flash Device.
- **EraseSector()**
  This function is called by the RL-FlashFS to erase a flash sector and from the **fformat** function to format the device.
- **EraseChip()**
  This function is used to global erase a flash device. It is called from the **fformat** function.

Every device has its own implementation of the Flash programming algorithms. These functions are located in the file **FS_FlashPrg.c** in the folder **\ARM\RL\FlashFS\Flash**.

To generate a new flash algorithm module, check the Flash Device datasheet, write the code for the functions above, and provide the Sector Layout description file.

As an alternative, existing ULINK Flash programming algorithms can be converted to be used with the RL-FlashFS. Flash programming algorithms are located in the folder **\ARM\Flash\*device family***. The source files are named **FlashDev.c** and **FlashPrg.c**. Refer to Converting FlashDev.c and Converting FlashPrg.c for detailed instructions.

Converting FlashDev.c

Here are instructions to convert existing device description files for various Flash Devices, which are used by ULINK flash programming with RL-FlashFS. There are two macros which must be specified in this file:

- the **FLASH_DEVICE** macro defines sector layout,
- the **FL_NSECT** macro defines the number of flash sectors.

The following example describes conversion for the **Am29x800BT** Flash Device:

1. Copy the **FlashDev.c** file from **\Keil\ARM\Flash** to a subfolder under **\Keil\ARM\RL\FlashFS\Flash** with the same subfolder name. This is a device name.
2. Rename this file to **FS_FlashDev.h**
3. Delete following lines:

```

#include "..\FlashOS.H"        // FlashOS Structures

struct FlashDevice const FlashDevice  =  {
  FLASH_DRV_VERS,              // Driver Version, do not modify!
  "AM29x800BT Flash",          // Device Name
  EXT16BIT,                    // Device Type
  0x000000,                    // Device Start Address
  0x100000,                    // Device Size in Bytes (1MB)
  1024,                        // Programming Page Size
  0,                           // Reserved, must be 0
  0xFF,                        // Initial Content of Erased Memory
  100,                         // Program Page Timeout 100 mSec
  3000,                        // Erase Sector Timeout 3000 mSec

// Specify Size and Address of Sectors
```

and lines at the bottom:

```

  SECTOR_END
};
```

20. Expand and Convert the Sector description and convert it to a macro **FLASH_DEVICE**.

```

  0x10000, 0x000000,           // Sector Size 64kB (14 Sectors)
```

This line needs to be expanded to 14 lines for 14 sectors using a **DFB** macro. Do not forget a macro continuation sign \.

```

  DFB(0x10000, 0x000000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x010000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x020000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x030000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x040000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x050000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x060000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x070000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x080000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x090000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0A0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0B0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0C0000),    /* Sector Size 64kB */    \
  DFB(0x10000, 0x0D0000),    /* Sector Size 64kB */    \
```

23. Repeat the expansion for each line from the file.

24. Add an **FL_NSECT** macro with defined number of sectors.

If you have done everything correctly, your **FS_FlashDev.h** file should look like this.

Converting FlashPrg.c

Here are instructions to convert existing programming algorithm modules for various Flash Devices, which are used by ULINK flash programming with RL-FlashFS. The functions from the original ULINK programming algorithms are similar. Only some small modifications are required to convert it for use with RL-FlashFS.

The following example describes the conversion for the **Am29x800BT** Flash Device:

1. Copy the **FlashPrg.c** file from **\Keil\ARM\Flash** to a subfolder under **\Keil\ARM\RL\FlashFS\Flash** with the same subfolder name. This is a device name.
2. Rename this file to **FS_FlashPrg.c**
3. Change the include header file from:

```
#include "..\FlashOS.H"        // FlashOS Structures
```

   to the RL-FlashFS definition header.

```
#include <File_Config.h>
```

6. Add the Flash driver control block definition:

```
/* Embedded Flash Driver Interface functions */
static BOOL Init        (U32 adr, U32 clk);
static BOOL UnInit      (void);
static BOOL ProgramPage (U32 adr, U32 sz, U8 *buf);
static BOOL EraseSector (U32 adr);
static BOOL EraseChip   (void);         /* Optional function if
supported    */

/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  NULL,                                 /* =NULL, use FFS internal
ReadData  */
  ProgramPage,
  EraseSector,
  EraseChip
};
```

24. Replace the function headers and rename the interface functions to conform with Flash driver conventions:
   - **Init()** from original

```
int Init (unsigned long adr, unsigned long clk, unsigned long fnc) {
```

     to

```
static BOOL Init (U32 adr, U32 clk)  {
```

     The *fnc* parameter is not used by the Flash driver.
   - **UnInit()** from original

```
int UnInit (unsigned long fnc)  {
```

     to

```
static BOOL UnInit (void)  {
```

     The *fnc* parameter is not used by the Flash driver.
   - **EraseChip()** from original

```
int EraseChip (void) {
```

to

```
static BOOL EraseChip (void) {
```

- **EraseSector()** from original
-

```
int EraseSector (unsigned long adr) {
```

to

```
static BOOL EraseSector (U32 adr) {
```

- **ProgramPage()** from original
-

```
int ProgramPage (unsigned long adr, unsigned long sz, unsigned char
*buf) {
```

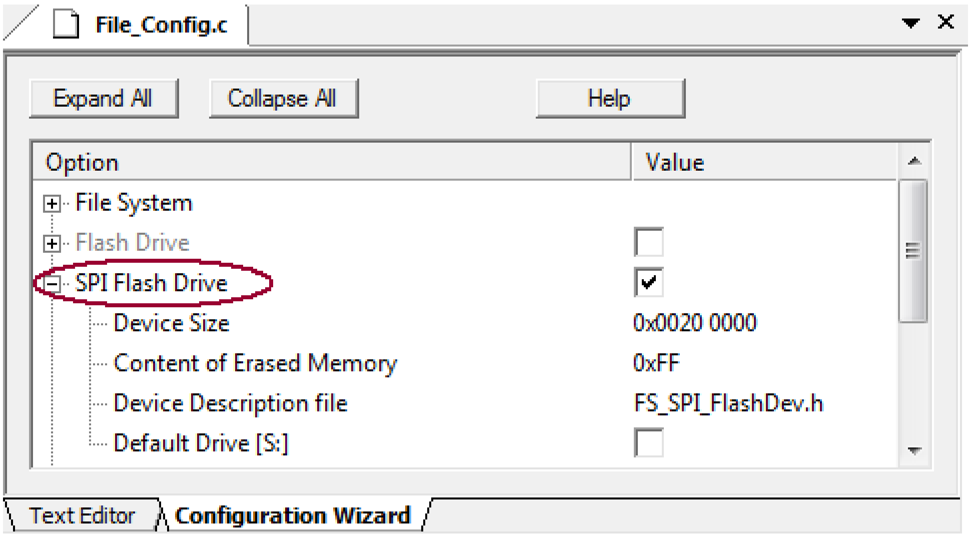
to

```
static BOOL ProgramPage (U32 adr, U32 sz, U8 *buf) {
```

25. Modify the function **return values** for all Flash driver functions. Return __TRUE on success, and __FALSE on failure.
26. Update the **ProgramPage()** function to allow unaligned buffer access. This in general means to add **__packed** attribute to buffer access if it is not a byte access.
27.
28. `/* 'buf' might be unaligned. */`
29. `M16(adr) = *(__packed U16 *)buf;`
30. Keep the optional local functions (ie. Polling), which Flash driver functions still need.

SPI Flash Drive

SPI Flash Drive options allow configuring RL-FlashFS to use SPI Flash devices. Set the options manually or use the Configuration Wizard. An SPI Flash Drive is accessed over the SPI bus.



Option	Value
⊞ File System	
⊞ Flash Drive	☐
⊟ SPI Flash Drive	☑
Device Size	0x0020 0000
Content of Erased Memory	0xFF
Device Description file	FS_SPI_FlashDev.h
Default Drive [S:]	☐

The following options can be set:

- **SPI Flash Drive** enables support for an SPI Flash Drive. It is not used when this option is disabled. It must be set to **1** to enable it and **0** to disable it.

```
#define SF0_EN       1
```

- **Device Size** specifies the size of the SPI flash device to be used for storing files. Typically, this is the whole size of the SPI Flash device, specified in bytes. It is allowed to specify only a part of the device for the Flash File System. The rest of the device might be used for other purposes. In this case, do not provide the function **EraseChip** in the driver control block. The value for the **EraseChip** function must be set to NULL.

```
#define SF0_SIZE     0x0200000
```

- **Content of Erased Memory** specifies initial content for erased SPI Flash Devices. In most cases, this value is set to 0xFF. Accepted values are 0xFF or 0x00.

```
#define SF0_INITV    0xFF
```

- **Device Description file** specifies a file containing the description of the SPI Sector Layout. The file is named **FS_SPI_FlashDev.h** and is tailored to a specific SPI Flash Device. Several description files are available in the folder **\ARM\RL\FlashFS\Flash**.

```
#define SF0_HFILE    "FS_SPI_FlashDev.h"
```

- **Default Drive [S:]** enables the SPI Flash Drive as a default system drive. This drive is used, when a drive letter is not specified in a filename.

```
#define SF0_DEF      1
```

Note

- For enabled SPI Flash Drives, add the SPI Flash Driver, the SPI Sector Layout description file, and the low-level routine SPI Driver to the project.

SPI Sector Layout

The **SPI Sector Layout** description file, **FS_SPI_FlashDev.h**, specifies the memory map of a SPI Flash device. Every device has its own description file, which is located in the folder **\ARM\RL\FlashFS\Flash**.

To generate a description file, copy the flash sector layout information from the Flash Device datasheet, specify a sector **size** in bytes, and a sector **base address** relative to an SPI Flash Device. The Macro **DSB** converts this information into the RL-FlashFS compatible sector description.

To improve the RL-FlashFS performance, the sector information is stored as a table in the code. The RL-FlashFS scans this table when accessing files from the SPI Flash device.

The following example shows a Flash sector layout configuration for the Intel SPI Flash device **25F160S33** with 2MByte memory:

```
#define SPI_FLASH_DEVICE                                 \
  DSB(0x10000, 0x000000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x010000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x020000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x030000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x040000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x050000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x060000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x070000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x080000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x090000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0A0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0B0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0C0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0D0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0E0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x0F0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x100000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x110000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x120000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x130000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x140000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x150000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x160000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x170000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x180000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x190000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1A0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1B0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1C0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1D0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1E0000),     /* Sector Size 64kB */ \
  DSB(0x10000, 0x1F0000),     /* Sector Size 64kB */ \

#define SF_NSECT    32
```

It is not optimal to define lots of small sectors (256 bytes or smaller). A more optimal solution for the RL-FlashFS is to join several physical sectors into bigger **virtual sectors**. In this case, the function **EraseSector** must be modified to erase a virtual sector, not a single physical sector.

SPI Flash Driver

The **SPI Flash Driver** implements flash programming interface functions. The interface to the RL-FlashFS is the flash driver control block.

Driver control block

The structure **EFS_DRV** is defined in the file **File_Config.h** as follows:

```
/* Embedded Flash Device driver */
typedef struct {
  BOOL (*Init)        (U32 adr, U32 clk);
  BOOL (*UnInit)      (void);
  BOOL (*ReadData)    (U32 adr, U32 sz, U8 *buf);
  BOOL (*ProgramPage) (U32 adr, U32 sz, U8 *buf);
  BOOL (*EraseSector) (U32 adr);
  BOOL (*EraseChip)   (void);                       /* Optional, NULL if not
existing    */
} const EFS_DRV;
```

The Flash driver uses six low-level user-provided functions to control the Flash device. The functions are located in the file **FS_SPI_FlashPrg.c** of the folder **\ARM\RL\FlashFS\Flash**.

- **Init()**
  This function is called when the RL-FlashFS is initialized by the function **finit**.
- **UnInit()**
  This function is called to uninitialize the embedded flash drive by the function **funinit**.
- **ReadData()**
  This function is used to read data from a Flash Device.
- **ProgramPage()**
  This function is used to program data into a Flash Device.
- **EraseSector()**
  This function is called by the RL-FlashFS to erase a flash sector and from the **fformat** function to format the device.
- **EraseChip()**
  This function is used to global erase a flash device. It is called from the function **fformat**.

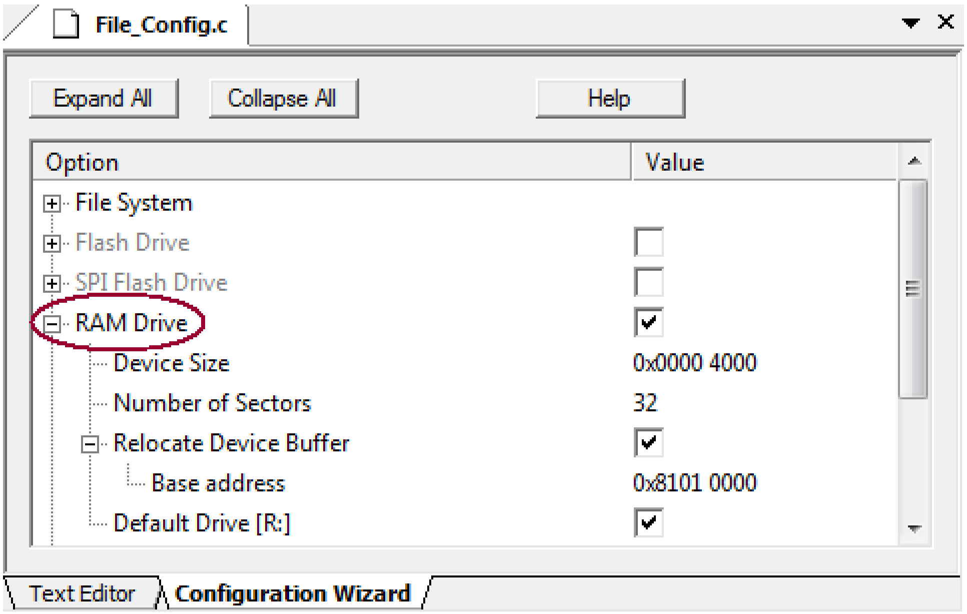
To generate a new Flash algorithm module (FS_SPI_FlashPrg.c), check the Flash device datasheet and implement the interface functions listed above.

Note

- In addition to this driver, the low-level SPI Driver is needed, which implements SPI serial communication routines.
- The SPI Driver **instance number** might need changing, if the SD Card drive is running in SPI mode. In this case, change the SPI driver index to **spi1_drv** or **spi2_drv** respectively.

RAM Drive

RAM Drive options allow configuring the RL-FlashFS to use RAM devices. Set the options manually or use the Configuration Wizard.



Option	Value
⊞ File System	
⊞ Flash Drive	□
⊞ SPI Flash Drive	□
⊟ RAM Drive	☑
Device Size	0x0000 4000
Number of Sectors	32
⊟ Relocate Device Buffer	☑
Base address	0x8101 0000
Default Drive [R:]	☑

The following options can be set:

- **RAM Drive** enables support for a RAM device. Set the option to **1** to enable it and **0** to disable it.

```
#define RAM0_EN      1
```

- **Device Size** specifies the size, in bytes, of the RAM device.

```
#define RAM0_SIZE    0x040000
```

- **Number of Sectors** specifies the number of logical sectors for a RAM drive. The following values are available: 8, 16, 32, 64, and 128. Select **smaller values** when storing a few bigger files on the RAM drive. For example only one or two big log files. **Bigger values** should be selected when storing lots of small files on the RAM drive. The default setting is 32.

```
#define RAM0_NSECT   32
```

- **Relocate Device Buffer** allows to allocate RAM buffer at a specific address in the memory space. It must be set to **1** to enable it and **0** to disable it. When disabled, the linker assigns the address of the RAM buffer.

```
#define RAM0_RELOC   1
```

- **Base address** specifies the starting address of the RAM buffer. This option is used when the **Relocate Device Buffer** is enabled.

```
#define RAM0_BADR    0x81010000
```

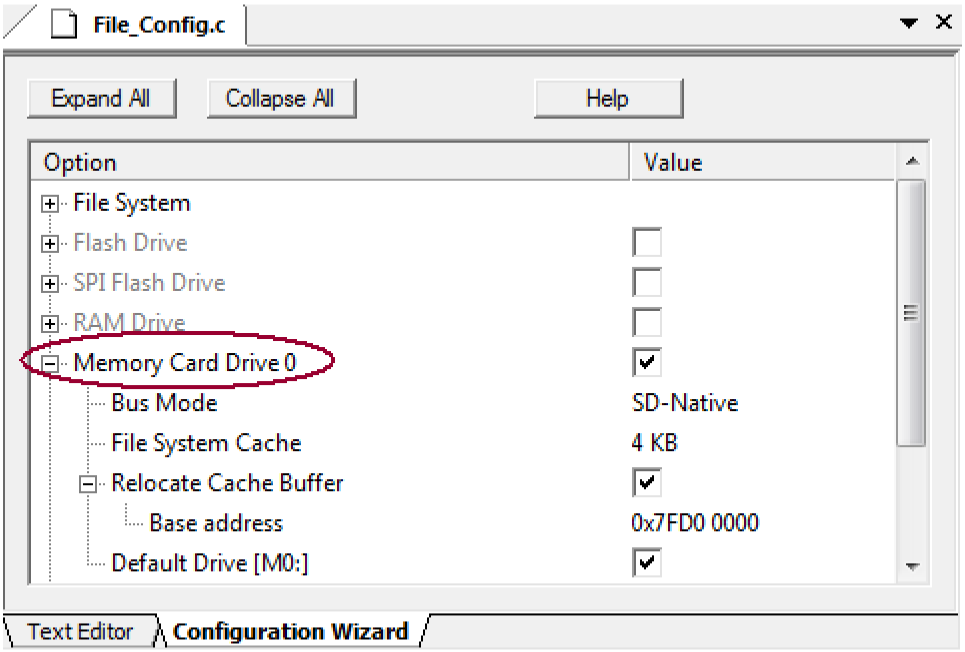
- **Default Drive [R:]** enables the RAM Drive as a default system drive. This drive is used, when a drive letter is not specified in a filename.

```
#define RAM0_DEF     1
```

When a RAM device is used with RL-FlashFS, the device is split automatically into **logical sectors** by the configuration. As a consequence, RL-FlashFS does not need an extra description table containing the sector layout.

Memory Card Drive

Memory Card Drive options allow configuring RL-FlashFS to use memory cards. Set the options manually or use the Configuration Wizard.



Option	Value
File System	
Flash Drive	□
SPI Flash Drive	□
RAM Drive	□
Memory Card Drive 0	☑
Bus Mode	SD-Native
File System Cache	4 KB
Relocate Cache Buffer	☑
Base address	0x7FD0 0000
Default Drive [M0:]	☑

The following options can be set:

- **Memory Card Drive 0** enables support for SD/MMC Flash memory card device. Set this option 1 to enable the device or 0 to disable the device.

```
#define MC0_EN       1
```

- **Bus Mode** specifies the access mode for SD/MMC Flash memory card device. This option should be set to 0 for **SD-Native** mode or to 1 for **SPI** mode.

  RL-FlashFS can use SPI mode or SD/MMC native mode to initialize and control the memory card drive.

  - **SPI mode** - the required routines are in the low level SPI driver. It handles data transfer on the SPI interface. In SPI mode the memory card control is handled in software.
  - **Native mode** - can be used if a device has integrated a Multimedia Card Interface peripheral. The required routines are in the low level MCI driver module. Native mode is faster then SPI mode because the memory card control is handled in hardware.

  The SPI driver or Native mode driver is not included in the RL-FlashFS library because it is device dependent. Hence, copy the driver to the project folder and include it into the project.

```
#define MC0_SPI      0                     // 0=SD-Native; 1=SPI
```

- **File System Cache** defines the data caching and specifies the **Cache Buffer size**. When SD/MMC Memory Card is controlled in **SD-Native** mode, data caching might increase the file r/w speed. When caching is enabled, Multiple Sector Read and Multiple Sector Write commands are used to control the SD/MMC memory card data read and write. Turn off the data cache if the application is low on memory and the file read/write speed is not important. The cache buffer size is specified in KBytes.

```
#define MC0_CASZ     4
```

- **Relocate Cache Buffer** allows to allocate the RAM buffer at a specific address in the memory space. It must be set to **1** to enable it and **0** to disable it. When disabled, the linker assigns the address of the RAM buffer.

```
#define MC0_RELOC    1
```

- **Base address** specifies the location address of the Cache Buffer. This option is active when

Relocate Cache Buffer is enabled.

```
#define MC0_CADR     0x7FD00000
```

- **Default Drive [M0:]** enables the Memory Card Drive 0 as a default system drive. This drive is used, when no drive letter is specified in a filename.

```
#define MC0_DEF      1
```

The RL-FlashFS supports two Memory Card drives in the system. The identical options exist also for **Memory Card Drive 1**. The drive label **M** is considered as **M0**. Both drives **M0** and **M1** can operate at the same time without limitations.

Note

- If the application needs file time information refer to File Time Support.

MCI Driver

The **MCI Driver** implements low-level routines to interface the SD/MMC flash memory cards in **SD-Native** mode. To make this possible, the host controller must have a Memory Card Interface peripheral that supports SD/MMC Memory Card interfacing in native mode. The interface to the RL-FlashFS is the MCI driver control block.

Driver control block

The structure **MCI_DRV** is defined in the file **File_Config.h** as follows:

```
typedef struct {
  BOOL (*Init)        (void);
  BOOL (*UnInit)      (void);
  void (*Delay)       (U32 us);
  BOOL (*BusMode)     (U32 mode);
  BOOL (*BusWidth)    (U32 width);
  BOOL (*BusSpeed)    (U32 kbaud);
  U32  (*Command)     (U8  cmd, U32 arg, U32 resp, U32 *rp);
  BOOL (*ReadBlock)   (U32 bl, U8 *buf, U32 cnt);
  BOOL (*WriteBlock)  (U32 bl, U8 *buf, U32 cnt);
  BOOL (*SetDma)      (U32 mode, U8 *buf, U32 cnt); /* NULL for local DMA or non
DMA  */
  U32  (*CheckMedia)  (void);                       /* Optional, NULL if not
existing */
} const MCI_DRV;
```

The MCI driver uses eleven low-level user-provided functions to control the Memory Card interface:

- **Init()**
  This function is called when the RL-FlashFS is initialized by the **finit** function.
- **UnInit()**
  This function is called to uninitialize the MCI interface by the **funinit** function.
- **Delay()**
  This function is called to delay a program execution in the driver.
- **BusMode()**
  This function is used to set the bus mode to push-pull or open-drain.
- **BusWidth()**
  This function is used to set the bus width to 1-bit or 4-bit bus.
- **BusSpeed()**
  This function is used to set the desired baud rate speed.
- **Command()**
  This function is used to send SD/MMC Command.
- **ReadBlock()**
  This function is used to read block(s) of data from SD/MMC memory card.
- **WriteBlock()**
  This function is used to write block(s) of data to SD/MMC memory card.
- **SetDma()**
  This function is used to set the DMA for data transfer.
- **CheckMedia()**
  This function is used to check the SD/MMC media status (Card Inserted, Write Protected).

Implemented drivers

RL-FLashFS includes the following MCI drivers in the folder **\ARM\RL\FlashFS\Drivers**:

- **MCI_LPC23xx.C** - for NXP LPC23xx devices.
- **MCI_LPC24xx.C** - for NXP LPC24xx devices.
- **MCI_LPC3xxx.C** - for NXP LPC3000 devices.
- **MCI_SAM3U.C** - for Atmel AT91SAM3U devices.
- **MCI_SAM9.C** - for Atmel AT91SAM9 devices.
- **MCI_SAM9G20.C** - for Atmel AT91SAM9G20 devices.

- **MCI_SAM9RL.C** - for Atmel AT91SAM9RL devices.
- **MCI_SAM9260.C** - for Atmel AT91SAM9260 devices.
- **MCI_SAM9261.C** - for Atmel AT91SAM9261 devices.
- **MCI_SAM3U.C** - for Atmel ATSAM3U devices.
- **SDIO_STM32F103.C** - for ST STM32F103 devices.

Copy one of the provided driver modules and use it as a template for a new MCI interface driver.

note

- Copy the MCI driver to the project folder if a flash memory card is used in SD-Native mode. An MCI driver is not required, if the MMC/SD card is used in SPI mode.

SPI Driver

The **SPI Driver** implements low-level routines to interface the SD/MMC Flash memory cards or SPI data Flash memory. An interface to the RL-FlashFS is the SPI driver control block.

Driver control block

The structure **SPI_DRV** is defined in File_Config.h as follows:

```
typedef struct {
  BOOL (*Init)         (void);
  BOOL (*UnInit)       (void);
  U8   (*Send)         (U8 outb);
  BOOL (*SendBuf)      (U8 *buf, U32 sz);
  BOOL (*RecBuf)       (U8 *buf, U32 sz);
  BOOL (*BusSpeed)     (U32 kbaud);
  BOOL (*SetSS)        (U32 ss);
  U32  (*CheckMedia)   (void);              /* Optional, NULL if not existing */
} const SPI_DRV;
```

The SPI driver uses eight low-level user-provided functions to control the SPI interface:

- **Init()**
  This function is called when the RL-FlashFS is initialized by the **finit** function.
- **UnInit()**
  This function is called to uninitialize the SPI interface by the **funinit** function.
- **Send()**
  This function is used to send and read a byte on the SPI interface.
- **SendBuf()**
  This function is used to send a block of data to the SPI interface.
- **RecBuf()**
  This function is used to receive a block of data from the SPI interface.
- **BusSpeed()**
  This function is used to set the desired baud rate speed.
- **SetSS()**
  This function is used to enable or disable SPI Slave Select signal (drive it high or low).
- **CheckMedia()**
  This function is used to check the SD/MMC media status (Card Inserted, Write Protected).

Implemented drivers

RL-FLashFS includes the following SPI drivers in the **\Keil\ARM\RL\FlashFS\Drivers** directory:

- **SPI_LPC17xx.C** - for NXP LPC17xx Cortex-M3 devices
- **SPI_LPC214x.C** - for NXP LPC214x ARM7 devices
- **SPI_LPC29xx.C** - for NXP LPC29xx ARM9 devices
- **SPI_STR71x.C** - for ST Microelectronics STR71x ARM7 devices
- **SPI_STR75x.C** - for ST Microelectronics STR75x ARM7 devices
- **SPI_STR91x.C** - for ST Microelectronics STR91x ARM9 devices
- **SPI_STM32F103.C** - for ST Microelectronics STM32F103 Cortex-M3 devices
- **SPI_STM32F107.C** - for ST Microelectronics STM32F107 Cortex-M3 devices
- **SPI_LM3S37x8.C** - for Luminary LM3S37x8 Cortex-M3 devices
- **SPI_LM3S6965.C** - for Luminary LM3S6965 Cortex-M3 devices
- **SPI_LM3S8962.C** - for Luminary LM3S8962 Cortex-M3 devices
- **SPI_LM3S37x8.C** - for Luminary LM3S37x8 Cortex-M3 devices
- **SPI_SAM7X.C** - for Atmel AT91SAM7X ARM7 devices
- **SPI_EFM32_Gxxx.C** - for EnergyMicro EFM32Gxxx Cortex-M3 devices

If RL-FlashFS does not contain the SPI driver for the device selected, create new SPI driver routines and locate them in a single module. It is good practice to name the module according to the SPI controller type.

Copy one of the provided driver modules and use it as a template for the new SPI interface driver.

Note

- Some of the SD/MMC flash memory cards do not work without pullups on all communication lines. For this reason you might need to connect **47K** pullups to **SSEL**, **MOSI**, **MISO** and **SCK**.

File Time Support

File Time Support is possible with RL-FlashFS. RL-FlashFS uses the standard FAT file system to store data in Flash memory cards. The FAT file system stores the time when a file was created, modified, or last accessed. RL-FlashFS is able to retrieve the current time and date.

If the system has a Real Time Clock (RTC) functionality, modify the provided RTC interface module to allow RL-FlashFS to read the current time and date. Copy the Real Time Interface module **fs_time.c** from the folder **\ARM\RL\FlashFS\SRC** to the project folder and customize the RTC functions:

- **fs_get_time()** - read the current Time
- **fs_get_date()** - read the current Date.

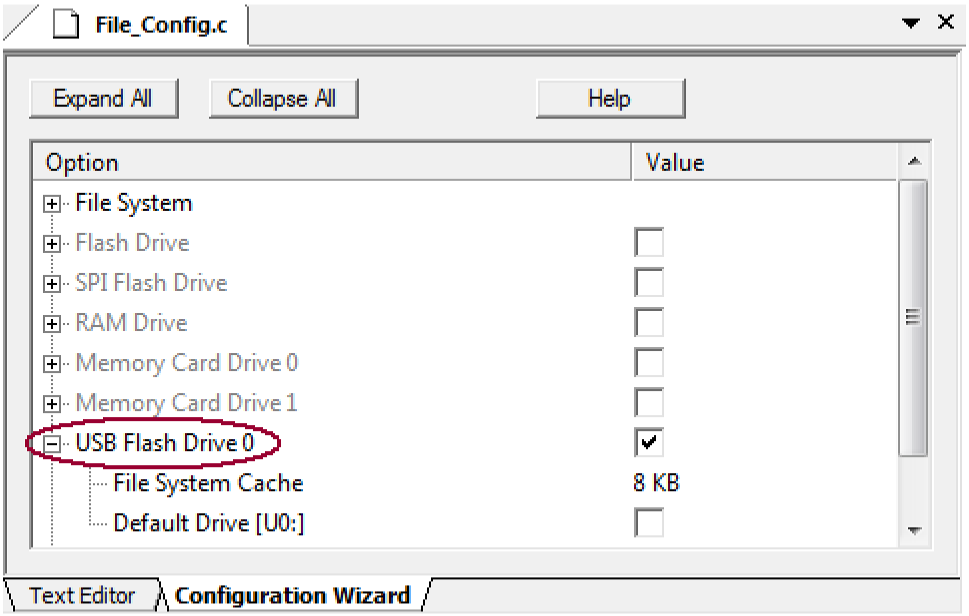
The customized functions can be used with the RL-FlashFS library. The linker will overwrite the library RTC functions with the customized RTC functions. However, **both** RTC interface functions must be provided in the project to replace the default library RTC functions.

Note

- A default RTC interface module is included in the RL-FlashFS library. This default implementation returns a fixed time and a fixed date that has been coded into the library.

USB Flash Drive

USB Flash Drive options allow configuring RL-FlashFS to use USB Flash devices. Set the options manually or use the Configuration Wizard.



Option	Value
⊞ File System	
⊞ Flash Drive	□
⊞ SPI Flash Drive	□
⊞ RAM Drive	□
⊞ Memory Card Drive 0	□
⊞ Memory Card Drive 1	□
⊟ USB Flash Drive 0	☑
File System Cache	8 KB
Default Drive [U0:]	□

To following options can be set:

- **USB Flash Drive 0** enables support for Mass Storage device. Set this option to 1 to enable the device or 0 to disable the device.

```
#define USB0_EN      1
```

- **File System Cache** enables or disables data caching and specifies the cache buffer size in KBytes. Data caching might increase the file r/w speed several times. Turn off data caching if the application is low on memory and the file read/write speed is not important.

```
#define USB0_CASZ    8
```

- **Default Drive [U0:]** enables the USB Flash Drive 0 as a default system drive. This drive is used, when a drive letter is not specified in a filename.

```
#define USB0_DEF     1
```

RL-FlashFS supports two USB Flash drives in the system. Identical options exist also for **USB Flash Drive 1**. The drive label **U** is considered as **U0**. Both drives **U0** and **U1** can operate at the same time without limitations.

Note

- Add also the FAT Driver to the project.

FAT Driver

The **FAT Driver** implements low-level routines to interface the USB Flash drive. To make this possible, the host controller must have a USB Host Interface peripheral and USB Host stack running. An interface to the RL-FlashFS is the FAT driver control block.

Driver control block

The structure **FAT_DRV** is defined in the file **File_Config.h** as follows:

```
typedef struct {
  BOOL (*Init)         (U32 mode);
  BOOL (*UnInit)       (U32 mode);
  BOOL (*ReadSect)     (U32 sect, U8 *buf, U32 cnt);
  BOOL (*WriteSect)    (U32 sect, U8 *buf, U32 cnt);
  BOOL (*ReadInfo)     (Media_INFO *cfg);
  U32  (*CheckMedia)   (void);           /* Optional, NULL if not existing */
} const FAT_DRV;
```

The MSD driver uses six low-level user-provided functions to control the USB Flash drive:

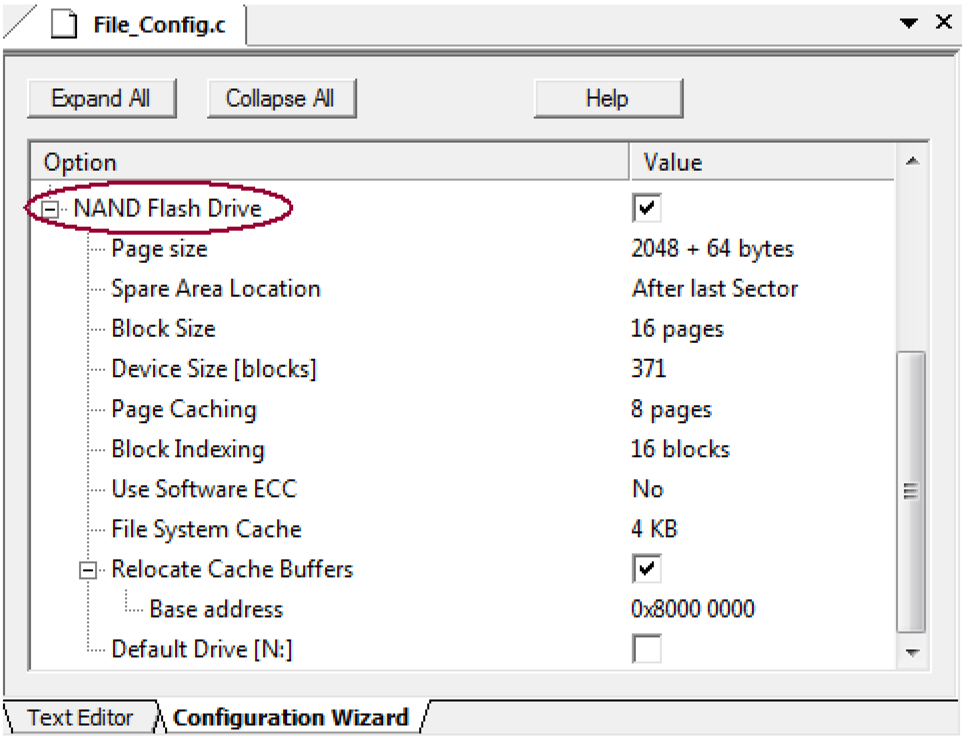
- **Init()**
  This function is called when the RL-FlashFS is initialized by the **finit** function.
- **UnInit()**
  This function is called to uninitialize the USB Flash Drive by the **funinit** function.
- **ReadSect()**
  This function is used to read sectors from the FAT Drive.
- **WriteSect()**
  This function is used to write sectors to the FAT Drive.
- **ReadInfo()**
  This function is used to read configuration info from the FAT Drive.
- **CheckMedia()**
  This function is used to check the FAT Drive media status.

Note

- The interface to the FAT module in RL-FlashFS library is a FAT Driver. Also the NAND Drive and Memory Card Drive are interfaced to FAT layer via the FAT Driver. However, the FAT Driver configuration of these drives is made internally in File_Config.c configuration.

NAND Flash Drive

NAND Flash Drive options allow configuring RL-FlashFS to use NAND Flash devices. Set the options manually or use the Configuration Wizard.



Option	Value
NAND Flash Drive	☑
Page size	2048 + 64 bytes
Spare Area Location	After last Sector
Block Size	16 pages
Device Size [blocks]	371
Page Caching	8 pages
Block Indexing	16 blocks
Use Software ECC	No
File System Cache	4 KB
Relocate Cache Buffers	☑
Base address	0x8000 0000
Default Drive [N:]	□

To following options can be set:

- **NAND Flash Drive** enables support for NAND Flash device. Set the option to 1 to enable NAND drive support. 0 disables the NAND drive.

```
#define NAND0_EN     1
```

- **Page size** specifies the NAND Flash read/write page size. The page size is defined as the sum of **user** plus **spare** area. Select the page size supported by the NAND Flash device. The following standard page sizes are available:

Page Size	Config Wizard Option
528	512 + 16 bytes
2112	2048 + 64 bytes
4224	4096 + 128 bytes
8448	8192 + 256 bytes

```
#define NAND0_PGSZ  2112
```

- **Block Size** is a size of NAND Flash block. It is specified in number of flash pages. The following block sizes are available:

Block Size	Config Wizard Option
8	8 pages
16	16 pages
32	32 pages
64	64 pages
128	128 pages
256	256 pages

```
#define NAND0_PGCNT 16
```

- **Device Size** specifies number of blocks that are available in NAND Flash device.

```
#define NAND0_BLCNT 371
```

- **Page Caching** enables or disables the page data caching. The NAND page data caching might increase the NAND Flash r/w speed a lot. When Page caching is enabled, the Flash

File System keeps recently accessed NAND flash pages in cache memory for faster access.

```
#define NAND0_CAPG   8
```

- **Block Indexing** enables or disables the NAND Flash block indexing. When Block indexing is enabled, the Flash File System keeps the Flash block index table in memory and does not scan the NAND flash blocks.

```
#define NAND0_CABL   16
```

- **Use Software ECC** enables or disables using of the software Error Correction Algorithms from RL-FlashFS library. This option should be enabled, when the NAND Flash Driver does not implement this functionality in hardware. Set the option to 1 to enable it. 0 disables the option.

```
#define NAND0_SWECC 0
```

- **File System Cache** enables or disables the data caching and specifies the **Cache Buffer size**. The data caching might increase the file r/w speed several times. When Caching is enabled, Flash File System uses Multiple Sector Read and Multiple Sector Write commands to control the USB Flash data read and write. Cache buffer size is specified in KBytes. Turn off the data cache if the application is low on memory and the file read/write speed is not important.

```
#define NAND0_CASZ   8
```

- **Relocate Cache Buffer** allows to allocate RAM buffer at a specific address in the memory space. Set to the option to 1 to enable it. 0 disables the option. When disabled, the linker assigns the address of the RAM buffer.

```
#define NAND0_RELOC 1
```

- **Base address** specifies the base address of the RAM buffers. This option is used when the **Relocate Device Buffer** is enabled.

```
#define NAND0_CADR   0x80000000
```

- **Default Drive [N:]** enables the NAND Flash Drive as a default system drive. This drive is used, when a drive letter is not specified in a filename.

```
#define NAND0_DEF    1
```

Note

- For enabled NAND Flash Drive, add the NAND Driver to the project.

NAND Driver

The **NAND Driver** implements low-level routines to interface the NAND Flash drive. An interface to the RL-FlashFS is the NAND driver control block.

Driver control block

The structure **NAND_DRV** is defined in File_Config.h as follows:

```
typedef struct {
  U32   (*Init)       (NAND_DRV_CFG *cfg);
  U32   (*UnInit)     (NAND_DRV_CFG *cfg);
  U32   (*PageRead)   (U32 row, U8 *buf, NAND_DRV_CFG *cfg);
  U32   (*PageWrite)  (U32 row, U8 *buf, NAND_DRV_CFG *cfg);
  U32   (*BlockErase) (U32 row, NAND_DRV_CFG *cfg);
} const NAND_DRV;
```

The NAND driver uses five low-level user-provided functions to control the NAND Flash drive:

- **Init()**
  This function is called when the RL-FlashFS is initialized by the **finit** function.
- **UnInit()**
  This function is called to uninitialize the NAND Flash Drive by the **funinit** function.
- **PageRead()**
  This function is used to read a page from NAND Flash device.
- **PageWrite()**
  This function is used to write a page to NAND Flash device.
- **BlockErase()**
  This function is used to erase NAND Flash block.

Implemented drivers

RL-FLashFS includes the following NAND drivers in the folder **\ARM\RL\FlashFS\Drivers**:

- **NAND_LPC32xx.C** - for NXP LPC3200 devices.
- **NAND_SAM3U.C** - for Atmel AT91SAM3U devices.

Copy one of the provided driver modules and use it as a template for creating new NAND interface driver.

note

- Copy the NAND driver to the project directory if a NAND Flash drive is used.

Page Data Layout

Page Data Layout explains the NAND driver configuration structure, default data organization in a NAND Flash Device page and how this organization can be changed.

NAND_DRV_CFG structure

This structure is defined in File_Config.h as follows:

```
typedef struct {
  NAND_PG_LAY *PgLay;                    /* Page Layout Definitions
*/
  U16 NumBlocks;                         /* Number of blocks per device
*/
  U16 NumPages;                          /* Number of pages per block
*/
  U16 PageSize;                          /* Page size
*/
  U16 SectorsPerBlock;                   /* Number of sectors per block
*/
  U8  SectorsPerPage;                    /* Number of sectors per page
*/
  U8  AddrCycles;                        /* Device address cycles
*/
  U8  SwEccEn;                           /* Software ECC enabled
*/
  U8  DrvInst;                           /* Driver Instance definition
*/
} const NAND_DRV_CFG;
```

NAND flash device driver is provided with the device configuration structure which contains NAND flash device configuration info:

- **PgLay**
  Pointer to the page data layout definition structure. This structure is filled with default layout definitions at FlashFS initialization.
- **NumBlocks**
  Number of blocks per device, as defined in the NAND Flash Drive configuration.
- **NumPages**
  Number of pages per block, as defined in the NAND Flash Drive configuration.
- **PageSize**
  Device page size, as defined in the NAND Flash Drive configuration.
- **SectorsPerBlock**
  Number of sectors per device block is derived from the device page size and number of pages per block.
- **SectorsPerPage**
  Number of sectors per device page is derived from device page size.
- **AddrCycles**
  Number of address cycles required for NAND device addressing.
- **SwEccEn**
  This variable is greater than zero if error correction code (ECC) encoding/decoding is enabled in software:
  0 = Software ECC disabled
  1 = Hamming ECC algorithm enabled in software.
- **DrvInst**
  Provides information about NAND flash drive instance number. This variable can be used to determine NAND device chip select. If only one NAND drive is used, this value equals to zero.

NAND_PG_LAY structure

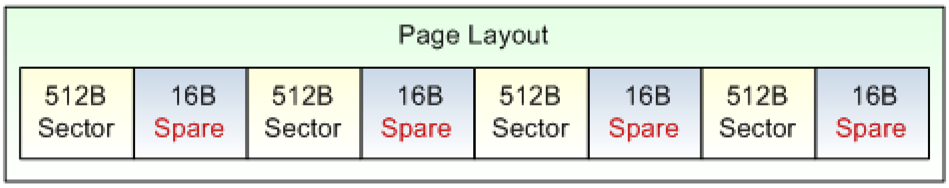
This structure is defined in File_Config.h as follows:

```
typedef struct {
  U8   Pos_LSN;                        /* LSN position
*/
  U8   Pos_COR;                        /* Data in page corrupted marker
*/
  U8   Pos_BBM;                        /* Bad Block marker position
*/
  U8   Pos_ECC;                        /* First byte of ECC
*/
  U16 SectInc;                         /* Column increment till next sector
*/
  U16 SpareOfs;                        /* Spare area offset from begining
*/
                                       /* of the page
*/
  U16 SpareInc;                        /* Column increment till next spare
*/
} NAND_PG_LAY;
```

This structure contains basic configuration info:

- **Pos_LSN**
  Position in spare area, where logical sector number (LSN) is placed. Usually, this is the first byte of spare, therefore Pos_LSN has value zero. LSN is a 32-bit value.
- **Pos_COR**
  Position of data corrupted marker in spare area. Usually, this byte is the fifth byte of spare and Pos_COR has value four.
- **Pos_BBM**
  Position of bad block marker (BBM) in spare area and is usually placed as the sixth byte of spare, Pos_BBM has value 5.
- **Pos_ECC**
  Position of the first byte of Error Correction Code (ECC) bytes in the spare area. First ECC byte is default seventh byte of spare (Pos_ECC == 6). This value is used by flash translation layer only if ECC is encoded and decoded in software.
- **SectInc**
  Provides information about user data sector locations within page. If page contains multiple sectors, first sector always starts at the begining of the page (byte zero). Second sector starts at SectInc, third sector at SectInc + SectInc and so on.
- **SpareOfs**
  Provides information about the location of the first spare area byte within page.
- **SpareInc**
  Provides information about spare area locations within page. If page contains multiple sectors, first byte of the first spare area is determined by reading SpareOfs value. Location of the first byte of the second spare, can be determined by adding SpareInc value to the SpareOfs value.

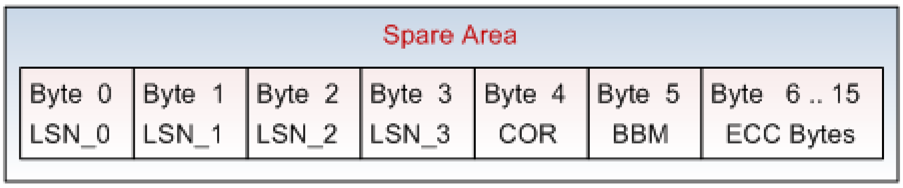
Default page data layout (defined by SectInc, SpareOfs and SpareInc values) contains spare area after each sector.

- 

Page Layout

512B Sector	16B Spare	512B Sector	16B Spare	512B Sector	16B Spare	512B Sector	16B Spare

Default 16-byte spare area data organization (defined by Pos_LSN, Pos_BBM and Pos_ECC values):

- 

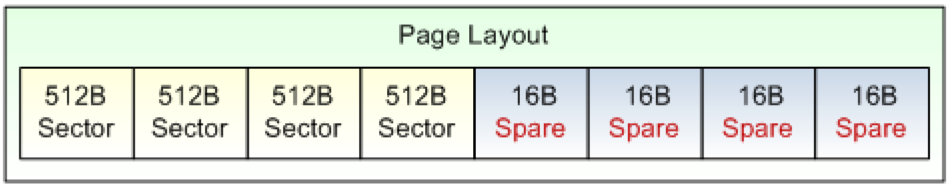
  Spare Area

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6 .. 15
LSN_0	LSN_1	LSN_2	LSN_3	COR	BBM	ECC Bytes

Changing the default page data layout

NAND Flash device or controller peripheral can demand different page data layout in order to automatically calculate and store redundant error correction information. Therefore, page data layout can be changed through configuration structure which is provided to the NAND flash device driver. To define page data layout other than default, NAND device driver can simply overwrite fields in the NAND_PG_LAY structure when the **Init** function is called.

The spare area location for OneNAND device is after last sector in a page:

- 

  Page Layout

512B Sector	512B Sector	512B Sector	512B Sector	16B Spare	16B Spare	16B Spare	16B Spare

Page data layout example to support OneNAND devices:

```
static U32 Init (NAND_DRV_CFG *cfg) {

  /* Setup OneNAND Page Layout */
  cfg->PgLay->Pos_LSN   =    2;
  cfg->PgLay->Pos_COR   =    1;
  cfg->PgLay->Pos_BBM   =    0;
  cfg->PgLay->Pos_ECC   =    8;
  cfg->PgLay->SectInc   =  512;
  cfg->PgLay->SpareOfs = 2048;
  cfg->PgLay->SpareInc =   16;

  /* Init NAND Driver Peripheral */
  /*          ...                */
  return RTV_NOERR;
}
```

Source Files

Source Files for creating applications with the RL-FlashFS library can be found in the folders:

Folder Name	Description
\ARM\RV31\INC	Contains include files, header files, and configuration files.
\ARM\RV31\LIB	Contains the library files **FS_ARM_L.LIB** and **FS_CM3.LIB**.
\ARM\RL\FlashFS\Drivers	Contains driver modules for MCI, MMC, NAND, and SPI devices.
\ARM\RL\FlashFS\Config	Contains configuration files, such as **File_Config.c** and **Retarget.c**.
\ARM\RL\FlashFS\Flash\*device family*	Contains Flash programming functions and device description files.
\ARM\Boards\*vendor*\board\RL\FlashFS	Contains example applications built with the RL-FlashFS Library. Use the projects as templates to create new applications.

RL-FlashFS include files in **\ARM\RV31\INC**:

File Name	File Type	Layer	Description
absacc.h	Header File	All layers	Header file to locating variables at absolute addresses at C level. The file is included from the file File_lib.c. Code changes are not required.
File_Config.h	Header File	All layers	Header file with common definitions. Code changes are not required.
File_lib.c	Module	All layers	System configuration file outlining library functions. Code changes are not required.
RTL.h	Header File	All layers	Common header file with type definitions and exporting library functions. Code changes are not required.
RTX_lib.c	Module	All layers	RTX Kernel configuration file exposing RTOS functions. Needed when the RTX-RTOS is used in the application. Code changes are not required.

RL-FlashFS library files in **\ARM\RV31\LIB**:

File Name	File Type	Layer	Description
FS_ARM_L.lib	Library	All layers	RL-FlashFS library for ARM7 and ARM9 devices - Little Endian.
FS_CM3.lib	Library	All layers	RL-FlashFS library for Cortex-M devices - Little Endian.

RL-FlashFS interface files in **\ARM\RL\FlashFS\Drivers**:

File Name	File Type	Layer	Description
MCI_*device family*	Module and include file	All layers	Multimedia Card Interface driver files with device specific definitions and functions. Code changes are not required.
NAND_*device family*	Module and include file	All layers	NAND Flash Interface driver files with device specific definitions and functions. Code changes are not required.
SDIO_*device family*	Module and include file	All layers	Multimedia Card Interface driver files with device specific definitions and functions. Code changes are not required.
SPI_*device family*.c	Module	All layers	Serial Peripheral Interface driver file with device specific definitions and functions. Code changes are not required.

RL-FlashFS configuration files in **\ARM\RL\FlashFS\Config**:

File Name	File Type	Layer	Description
File_Config.c	Module	All layers	Application and device configuration file. Code changes can be entered manually or using the µVision Configuration Wizard.
Retarget.c	Module	All layers	Module exposing low-level I/O functions. Code changes are not required.

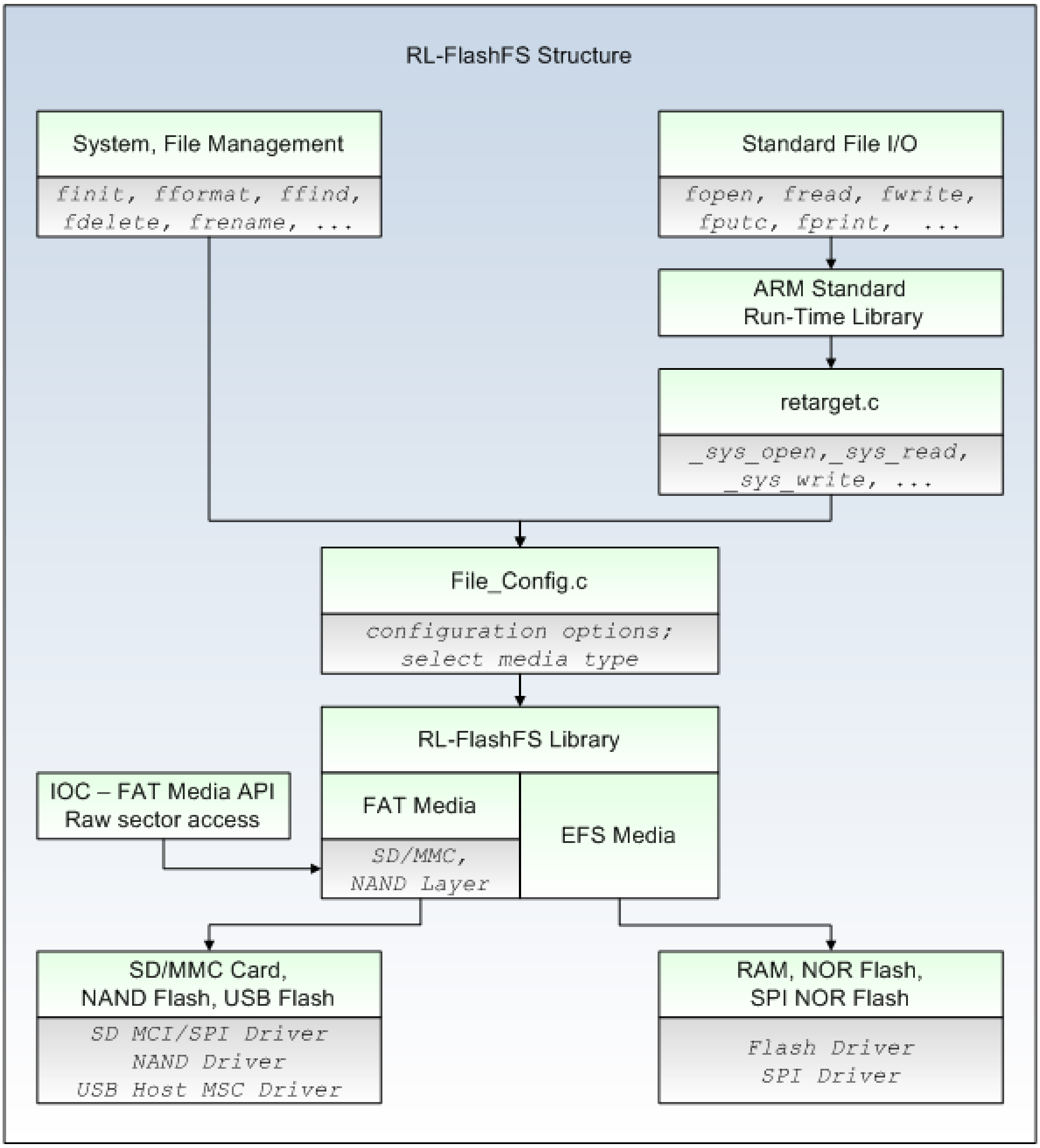
RL-FlashFS Flash programming and device description files in **\ARM\RL\FlashFS\Flash\*device family***:

File Name	File Type	Layer	Description
FS_FlashDev.h	Header file	All layers	Device description file defining the memory layout for devices using Embedded File System. Adapt the code the application needs.
FS_FlashPrg.c	Module	All layers	Module with Flash programming functions for devices using the Embedded File System. Adapt the code to the application needs.
FS_SPI_FlashDev.h	Header file	All layers	Device description file for SPI devices outlining the memory layout. Adapt the code the application needs.
FS_SPI_FlashPrg.c	Module	All layers	Module with Flash programming functions for SPI devices. Adapt the code to the application needs.
IAP.s	Module	All layers	Assembler file with IAP execution functions. Adapt the code to the application needs.

Using RL-FlashFS

Using RL-FlashFS shows how to create applications for managing files on Flash devices. RL-FlashFS can be used stand-alone or with the RTX-RTOS. RL-FlashFS cannot be used with the **MicroLIB** library.

The picture below explains the RL-FlashFS structure from a developer's perspective.


RL-FlashFS Structure
System, File Management
finit, fformat, ffind, fdelete, frename, ...
Standard File I/O
fopen, fread, fwrite, fputc, fprint, ...
ARM Standard Run-Time Library
retarget.c
_sys_open, _sys_read, _sys_write, ...
File_Config.c
configuration options; select media type
RL-FlashFS Library
IOC – FAT Media API Raw sector access
FAT Media
SD/MMC, NAND Layer
EFS Media
SD/MMC Card, NAND Flash, USB Flash
SD MCI/SPI Driver
NAND Driver
USB Host MSC Driver
RAM, NOR Flash, SPI NOR Flash
Flash Driver
SPI Driver

The logical picture block:

- **System, File Management**

  Represents functions to manage the system, such as file formatting, creating, finding, renaming, ...
- **Standard File I/O**

  Represents functions to manage data in files, such as reading, writing, printing, ...
- **ARM Standard Run-Time Library**

  Represents the library with functions to manage data in files.
- **retarget.c**

Represents the abstraction hardware layer with functions to input and output data on various interfaces, such as screens, LCD displays, keyboards, SD Cards, ...

- **File_Config.c**

  Represents the configuration file with options to define the media characteristics.
- **RL-FlashFS Library**

  Represents the library with interface functions that handle low-level input and output file operations. Through the media type selected in the **File_Config.c**, RL-FlashFS detects the appropriate file system: FAT or EFS. RL-FlashFS implements a Flash Translation Layer (FTL) for NAND Media.
- **IOC - FAT Media API**

  Represents interface functions for FAT Media that allow accessing raw sectors.
- The lower blocks

  Represent the supported media types and relates them to the file system.

This section includes the topics:

Using Flash Devices

Lists the configuration files needed for Flash devices (NOR Flash), and describes the steps for configuring dedicated-, large-, and internal Flash devices.

Using RAM Devices

Explains the configuration of RAM devices.

Using Memory Card Devices

Explains the configuration of memory card devices, file naming conventions, hot insertion, and root directory limitations.

Debugging

Explains the µVision debugger configuration and settings for debugging Flash devices.

Flash Device Applications

Flash Device Applications explains how to create applications for embedded Flash devices.

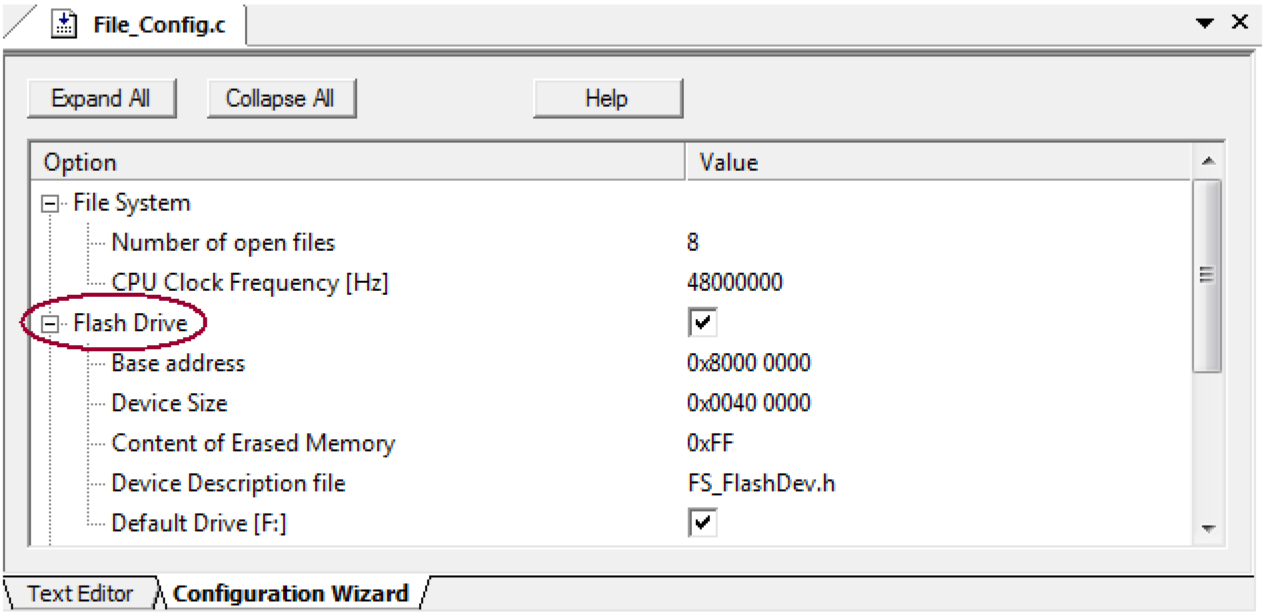
Include into the project and configure the following source files:

1. The library that matches the device core:
   FS_CM3.lib - for Cortex-M devices.
   FS_ARM_L.lib - for ARM7 or ARM9 devices.
2. The description files:
   FS_FlashDev.h for the sector layout.
   FS_FlashPrg.c with the Flash programming algorithm.
3. The configuration files:
   File_Config.c to configure the device.
   Retarget.c to configure the output.
4. The ***main*** file to initialize and connect the Flash device.

```

#include <RTL.h>
#include "Em_File.h"

int main (void)  {
  ...
  init_comm ();                               // initialize communication port
  init_file ();                               // initialize Flash File System

  while (1)  {
    ...                                       // add the code
  }
}
```

Applications can be created using existing µVision projects as templates. The projects are located in the folder **\ARM\Boards\*vendor**board name*\RL\FlashFS**.

- Copy all files from any folder **\ARM\Boards\*Vendor**BoardName*\RL\FlashFS\*em_file*** to a new folder and open the project ***.uvproj** with µVision. **RTX** projects are using the RTX-RTOS, whereas simple **Audio** projects work without an RTOS. However, the configuration does not differ.
- Open the file **File_Config.c** and configure the Flash device with the Configuration Wizard.
- Enable **Flash Device** and set the device characteristics.



Option	Value
⊟ File System	
Number of open files	8
CPU Clock Frequency [Hz]	48000000
⊟ Flash Drive	☑
Base address	0x8000 0000
Device Size	0x0040 0000
Content of Erased Memory	0xFF
Device Description file	FS_FlashDev.h
Default Drive [F:]	☑

- Adapt the source files **FS_FlashPrg.c** that contain Flash programming algorithms.
- Modify the file **usbd_user_adc.c** to adapt the code to the application needs.

When used for the first time, file storage devices need to be formatted:

- Non-volatile devices like Flash Devices, EEPROMS, or ZEROPOWER RAMs only once - when the system is started the **first time**.
- Standard RAM devices **every time** the system is started.

Note

- The configuration options are explained in Flash Drive.

Using Dedicated Flash Devices

If you use a dedicated Flash Device for storing files, then you must setup the Device configuration options, copy all three configuration files to your project directory, and include them in your project.

You should be aware of the following:

- **Writing** to Flash Memory is quite **fast**, typically around 10us/cell. It may still take some time to write big files. This also depends on the File Buffer size. When this buffer is full, RL-FlashFS writes the buffer content to a Flash memory.
- **Erasing** a Flash Sector is **slow**. Typically around 1sec/sector. During this time, the RL-FlashFS is frozen running in a loop inside **EraseSector()** function and waiting for the erase operation to finish.

Using One Large Flash Device

A special configuration is required when a single Flash Device is used for storing files and application code. The code can **not** be **executed** from a Flash device while it is **being programmed**. For this reason, programming routines are **relocated** to RAM and executed within RAM while Flash is being programmed or erased. The programming routines located in module **FS_FlashPrg.c** are relocated at runtime and copied to RAM for execution.

In addition to a standard Flash Device configuration, the programming functions must:

- **relocate** to RAM using the function attribute **__ram**,
- be **protected** from interrupts for **IRQ** and **FIQ** interrupts. This is possible by using the function attribute **__swi** or relocating your IRQ interrupt handler function to RAM also. You should be careful if you are using **FIQ** interrupts as well.

Using Internal Flash Devices

Using Internal Flash Memory of an ARM device for code execution and file storage is the same as using an external flash which stores the code and files. However, there are differences when comparing internal flash memory with external flash devices:

- In most cases, it is possible to program internal flash memory only with **IAP** function calls. IAP functions are factory preprogrammed into the boot sector of Internal Flash Memory.
- Another limitation is that flash memory may be programmed only a complete **page** at a time. A page starts at the page boundary address. A page size for Philips LPC2xxx devices is 512/1024/.. bytes. So to program only one byte requires reading a complete page from Internal Flash Memory into the RAM buffer, modifying the particular byte, and then writing back a complete page. This is handled by the **ProgramPage** function.
- IAP function calls also require an assembly interface function for C-function calls. This is why an **IAP.s** assembly module must be included for NXP (formerly Philips) Internal Flash programming algorithms.

There are preconfigured Flash Algorithms for NXP (formerly Philips) ARM devices in the **\Keil\ARM\RL\FlashFS\Flash** folder. You may use them as a reference for your own driver.

Note

- If the device uses an **Error Correction Code (ECC)** to correct single bit flash errors, then RL-FlashFS will **not work**. Such devices are, for example, NXP LPC213x and LPC214x devices.

Using RAM Devices

RL-FlashFS can be configured to store files also to a static RAM. The buffer for RAM drive is allocated by the linker. You can allocate the buffer at a specific address in the memory space, if you enable the Relocate Device Buffer in the File_Config.c configuration file.

Because the RAM content is undefined when the power is applied to the system, the Ram drive must be formatted for usage.

All the drives, that are enabled in the File_Config.c configuration, may be used at the same time, in parallel, with no limitations. Of course, all drives must be properly configured.

If the Ram drive is the only drive used in your project, you need to copy only the **File_Config.c** configuration file to your project directory.

Using Memory Card Devices

You can configure RL-FlashFS to store files in a standard SD/MMC flash memory card. All drives (**Flash drive**, **RAM drive**, and **MC drive**) can be used at the same time with no limitations, as long as each drive is properly configured. To configure the memory card drive, you must copy the configuration file **File_Config.c** and the SPI interface driver **SPI.c** to your project directory.

The RL-FlashFS supports standard **SD** and **MMC** flash memory cards with sizes ranging from a **few MBytes** up to **4 GBytes**. The system can read or write to the **Root directory** of the drive and to any subdirectories created. The root directory typically has **512 entries**. This limitation applies to FAT12 and FAT16 file system only. This means that 512 files plus root subdirectories together may be stored on the root directory of the drive. This limitation does not apply to the FAT32 file system.

RL-FlashFS fully support support **directories**, **subdirectories** and **long filenames**. The system can handle standard **12-bit**, **16-bit** and **32-bit** File Allocation Tables. When accessing files located in subdirectories a complete path must be specified in the file reference. For example:

```
fopen ("logs\\work\\temp.log","r");
```

Before a memory card is removed from the system, all files that have been opened for writing have to be closed. Flash File System uses a simple **FAT Table Caching** and **Data Caching** to speed-up memory card data access. Data Cache Buffer size and location is configured in the configuration file. If the files are not closed, modified data that is still in the FAT Cache is not written to the memory card. This makes the affected FAT table invalid and corrupts the file system.

note

- All **files** opened for writing must be **closed** before the memory card is **removed** from the socket. Otherwise, a FAT file system might be corrupted.

File Naming Convention

File system used for memory card has **FAT** implemented. When files are created the file system determines if file being created should be written to media as short file name or long file name.

Rules for determining if file name will be created as **short file name**:

- file name contains 8 or less characters
- file extension contains 3 or less characters
- no SPACE characters in file name or file extension
- all characters are in same CASE

If file name does not satisfy the rules for short file name it will be created as **long file name**.

Examples of short file names:

- test.txt
- TEST.TXT

Examples of long file names:

- Test.txt
- tesT.txt
- test file.txt

note

- Short names are encoded in FAT as all upper case.

Hot Insertion

When the **finit** function is called at startup, it tries to initialize the memory card. However, Flash Memory Cards also support hot insertion, which enables the memory card to be inserted when the system is already up and running. When the memory card is inserted, the **finit** function must be called to initialize the card before using the card. To detect the insertion of a memory card, the user application can **poll** the state of the **insertion switch**. Alternatively, the application can try to initialize the memory card periodically.

note

- If you want to detect hot insertion, your hardware must provide the insertion switch using a general purpose I/O pin.

Root Directory Limitation

The root directory in **FAT12** and **FAT16** file system has a limited storage of **512 entries**. This is a limitation of the FAT file system, not the limitation of the SD Card.

This limitation means:

1. You can store a maximum of **512 files** in the root folder, if the files are stored in **short filename** format (old DOS 8.3 file format).
2. You can store maximum of **256 files**, or **less** in the root folder, if the files are stored in **long filename** format. How many filename records are used depends on how long the filename is.

How to avoid this limitation

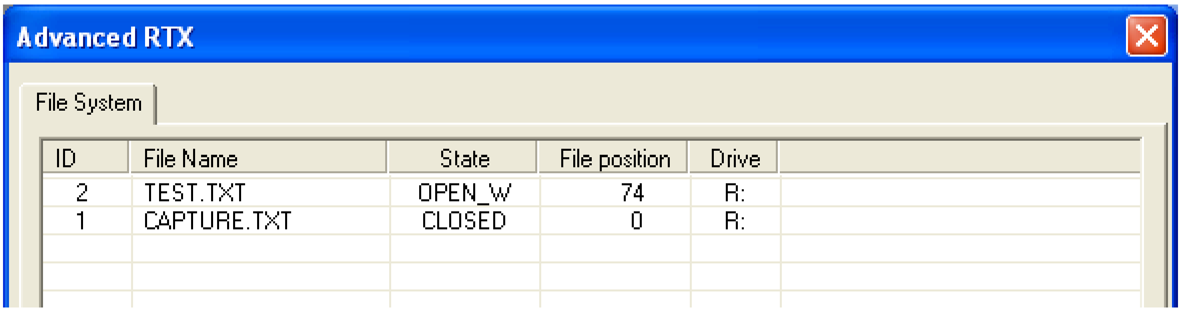
A simple workaround to avoid this limitation is to create a **subdirectory** in the root folder. Then create all the files in this subdirectory. There is no limitation how many files can be created in the subdirectory.

- The root directory limitation does not exist in **FAT32** file system.
- Long filename entry has **one short** and **several long** filename entries allocated.

Debugging

The µVision Simulator allows you to run and test your RL-FlashFS application. It must be configured to run from **RAM**. To simulate the target **flash** device, a special debugging script must be written. Detailed information about each file from the directory of the RL-FlashFS displays when you select the **File** tab. This dialog lists all stored files.

Select **RTX Kernel** from the **Peripherals** menu to display this dialog.



ID	File Name	State	File position	Drive
2	TEST.TXT	OPEN_W	74	R:
1	CAPTURE.TXT	CLOSED	0	R:

- **ID** is the File Identification Value.
- **File Name** is the **name** of this file.
- **State** is the current state of this file.
- **File position** is current file position pointer value.
- **Drive** is a drive where the file is stored.

Note

- You may have a **source level debugging** if you enter the following SET variable into the debugger:
-
- `SET SRC=C:\Keil\ARM\RL\FlashFS\SRC`
- You must select the RTX Kernel Operating System for the project under: **Options for Target — Target — Operating System — RTX Kernel** to enable the RL-FlashFS debug dialog.

Function Overview

This section summarizes all the routines in the RL-FlashFS product. The functions are ordered according to the following categories:

- File I/O Routines
- File Maintenance Routines
- File Time Support Routines
- Flash Driver Routines
- FAT Driver Routines
- MCI Driver Routines
- SPI Interface Routines
- NAND Interface Routines
- IOC Media Interface Routines
- System Routines

The function format is same as that of the RL-RTX functions.

note

- The RL-FlashFS library does not contain all the functions that are part of the RL-FlashFS product. The Library Reference section on each function mentions whether the function is in the library or not. If a function you want to use is not in the library, you must do one of the following:
  - Include one of the provided RL-FlashFS source files that contains the function in your project. You can further customize the function.
  - Provide your own function if the RL-FlashFS source files do not contain the function you require. This is usually the case when you want to use driver functions for a different hardware.

File I/O Routines

Routine	Description
fclose	Closes the file stream.
feof	Reports whether the end of stream has been reached.
ferror	Reports whether there is an error in the file stream.
fflush	Flushes the file stream.
fgetc	Reads a character from the file stream.
fgets	Reads a string from the file stream.
fopen	Opens the file stream.
fprintf	Writes a formatted string to the file stream.
fputc	Writes a character to the file stream.
fputs	Writes a string to the file stream.
fread	Reads a number of bytes from the file stream.
fscanf	Reads a formatted string from the file stream.
fseek	Moves the file stream's in-file pointer to a new location.
ftell	Gets the current location of the stream's in-file pointer.
fwrite	Writes a number of bytes to the file stream.
rewind	Moves the file stream's in-file pointer to the beginning of the file.
ungetc	Stores a character into an input file stream.

note

- The File I/O routines provide several ways of reading and writing files.
- The File I/O Flash routines are not reentrant.

File Maintenance Routines

Routine	Description
fanalyse	Checks the drive for fragmentation.
fcheck	Checks the consistency of the drive.
fdefrag	Defragments the drive.
fdelete	Deletes the specified file.
ffind	Performs a pattern-matching search for filenames.
fformat	Formats the drive.
ffree	Calculates the free space in the drive.
frename	Changes the name of the file.

note

- The File Maintenance routines enable you to perform file operations on the specified drive.
- The File Maintenance Flash routines are not reentrant.

File Time Support Routines

Routine	Description
fs_get_date	Returns the current date.
fs_get_time	Returns the current time.

note

- The file time support routines enable the RL-FlashFS system to get the current time and date when a file is created, modified, or accessed.
- The file time support routines are not reentrant.

Flash Driver Routines

Routine	Description
Init	Initializes the flash programming algorithm.
UnInit	Uninitializes the flash programming algorithm.
ReadData	Reads data from a flash memory.
ProgramPage	Writes data to a flash memory.
EraseSector	Erases a sector in flash memory.
EraseChip	Globally erases a flash memory.

note

- The low level flash routines enable the RL-FlashFS system to write to or erase the flash memory.
- The low-level flash routines are not reentrant.

FAT Driver Routines

Routine	Description
Init	Initializes the FAT Drive.
UnInit	Uninitializes the FAT Drive.
ReadSect	Reads sectors from FAT Drive.
WriteSect	Writes sectors to FAT Drive.
ReadInfo	Reads the configuration info from the FAT Drive.
CheckMedia	Checks the FAT Drive media status (Card Inserted, Write Protected).

note

- The FAT Driver routines are the functions that the RL-FlashFS system uses to communicate with a all drives that support FAT file system.
- The FAT Driver routines are not reentrant.

MCI Driver Routines

Routine	Description
Init	Initializes the MCI controller.
UnInit	Uninitializes the MCI controller.
Delay	Delays a program execution in the driver.
BusMode	Sets the bus mode to push-pull or open-drain.
BusWidth	Sets the bus width to 1-bit or 4-bit bus.
BusSpeed	Set the desired baud rate speed for the MCI interface.
Command	Sends SD/MMC command.
ReadBlock	Reads data from SD/MMC memory card.
WriteBlock	Writes data to SD/MMC memory card.
SetDma	Sets the DMA for data transfer.
CheckMedia	Checks the SD/MMC media status (Card Inserted, Write Protected).

note

- The MCI interface routines are the functions that the RL-FlashFS system uses to communicate with a memory card in SD-Native mode.
- The MCI interface routines are not reentrant.

SPI Driver Routines

Routine	Description
Init	Initializes the SPI controller.
UnInit	Uninitializes the SPI controller.
Send	Send and receive a byte over the SPI interface.
SendBuf	Sends a block of data to the SPI interface.
RecBuf	Receives a block of data from the SPI interface.
BusSpeed	Set the desired baud rate speed for the SPI interface.
SetSS	Enables or disables SPI Slave Select signal.
CheckMedia	Checks the SD/MMC media status (Card Inserted, Write Protected).

note

- The SPI interface routines are the functions that the RL-FlashFS system uses to communicate with a memory card or SPI data flash device.
- The SPI interface routines are not reentrant.

NAND Driver Routines

Routine	Description
Init	Initializes the NAND Flash Device.
UnInit	Uninitializes the NAND Flash Device.
PageRead	Reads a page from NAND Flash Device.
PageWrite	Writes a page to NAND Flash Device.
BlockErase	Erases a flash block on NAND Flash Device.

note

- The NAND Driver routines are the functions that the RL-FlashFS system uses to communicate with the NAND Flash drive.
- The NAND Driver routines are not reentrant.

IOC Interface Routines

Routine	Description
ioc_getcb	Retrieves the Media Control Block.
ioc_init	Initializes the FAT Media Device.
ioc_uninit	Uninitializes the FAT Media Device.
ioc_read_sect	Reads sectors from FAT Media Device.
ioc_write_sect	Writes sectors to FAT Media Device.
ioc_read_info	Reads the configuration info from the FAT Media Device.

note

- The IOC Media Interface routines are the functions that the RL-FlashFS system uses to communicate with a all media devices that support FAT file system.
- The FAT Media Interface routines are not reentrant.

System Functions

Routine	Description
finit	Initializes the RL-FlashFS system or drive
funinit	Uninitializes the RL-FlashFS system or drive

note

- The RL-FlashFS system routines are not reentrant.

RL-TCPnet

RL-TCPnet is an implementation of the **TCP/IP protocol** stack. The focus of the stack is to reduce memory usage and code size. This makes it suitable for use by small clients with limited resources, such as embedded systems. The RL-TCPnet library is a ground-up implementation of software routines for the ARM7™, ARM9™, Cortex™-M1 and Cortex™-M3 architectures.

Programs are written using standard C constructs and compiled with the ARM® RealView® Compiler. To create applications, you must include a special header file and link the RL-TCPnet library into your program.

Note

- **RL-TCPnet** is not included with the RealView MDK-ARM™ Microcontroller Development Kit. It is available in the stand-alone product **RL-ARM**™, which also contains the RTX kernel (with source code), Flash File System, CAN and USB drivers.

TCP Socket

The Transmission Control Protocol (TCP) runs on top of the Internet Protocol (IP). TCP is a connection-oriented and **reliable** byte stream service. The term connection-oriented means that the two applications using TCP must establish a TCP connection with each other before they can exchange data.

TCP is a **full duplex** protocol. This means that each TCP connection supports a pair of byte streams, one for each direction. It takes care of retransmitting any data which does not reach the final destination due to errors. It also takes care of retransmitting if there is data corruption of the received data.

TCP protocol delivers data (to the application) in the same sequence as they were transmitted.

Opening TCP Connection

RL-TCPnet is based on the client/server model of operation, and the TCP connection setup is based on these roles as well. Both, the client and the server, prepare for the connection by performing an **open** operation. However, there are two different kinds of **open**:

- **Active Open**: A client process using TCP takes the "active role" and initiates the connection by actually sending a TCP message to start the connection (a *SYN* message).
- **Passive Open**: A server process, designed to use TCP, takes a more relaxed approach. It performs a *passive Open* by contacting the TCP, which is like saying "I am here, and I am waiting for clients that may wish to talk to me to send me a message on the following port number". The *Open* is called *passive* because aside from indicating that the process is listening, the server process does nothing.

A *passive Open* can, in fact, specify that the server is waiting for an *Active Open* from a specific client. However, not all RL-TCPnet APIs support this capability.

TCP Active Open

Embedded applications use TCP Active Open when trying to connect to a remote server, for example to send an email. In this case, the TCPnet system is the initiator of the TCP connection.

To open an active TCP connection, the following steps must be taken:

1. **Enable** the TCP socket in the **Net_Config.c** configuration file.
2. **Allocate** a free TCP socket with the **tcp_get_socket()** function call.
3. **Initiate** the TCP connection by calling the **tcp_connect()** function.

TCP Passive Open

Passive Open is used when running a server application, for example a Web Server. The TCP socket opens in passive mode and waits for incoming connections.

Do the following steps to open a passive TCP connection:

1. **Enable** the TCP socket in the **Net_Config.c** configuration file.
2. **Allocate** a free TCP socket with the **tcp_get_socket()** function call.
3. **Activate** the TCP socket listen mode with the **tcp_listen()** function call.

RL-TCPnet can handle multiple connections on the same port. Several TCPnet applications such as: Web server, FTP server, Telnet server, etc. are using this concept.

Sending TCP Data

The TCP protocol is a byte stream service. It does not know anything about the format of the data being sent. It simply takes the data, **encapsulates** it into the TCP packet, and sends it to the remote peer. The TCP socket then keeps the last packet in memory and waits for an **acknowledge** from the remote peer.

If the packet is not acknowledged when the timeout expires, it resends the same packet. This process is repeated a couple of times before the packet is either acknowledged or the TCP socket aborts the connection.

The main goal is to keep it small and not resource hungry. For this reason, the TCP socket cannot keep a large amount of data in the buffer waiting to be acknowledged. It only keeps the **last packet sent**, in memory, until it is acknowledged. When the packet is acknowledged, it is released from memory.

Example for Sending Data

In the following example, the basic concept is to send large amounts of data using a TCP socket. This example sends 64 Kbytes to the remote IP address 192.168.0.100, which is listening on port 1000. The TCP socket is permanently allocated and is not released when the data is sent or when the connection is closed.

1. **Initialize** the RL-TCPnet system and **allocate** a free TCP socket:

```

#include <rtl.h>

U8 tcp_soc;
U8 soc_state;
BOOL wait_ack;

void main (void) {
  init_ ();
  tcp_soc = tcp_get_socket (TCP_TYPE_CLIENT, 0, 120, tcp_callback);
  soc_state = 0;
```

13. Run the **main thread** of the RL-TCPnet system and call the **send_data()** function from an endless loop:

```

  while (1) {
    timer_poll ();
    main_TcpNet ();
    send_data ();
  }
}
```

21. The **send_data()** function must be implemented as a **state machine**. It opens an Active TCP connection, sends data, and closes the TCP connection in the end. When the **soc_state** is 0, the connection is **initiated**:

```

void send_data (void) {
  static const U8 rem_IP[4] = {192,168,0,100};
  static int bcount;
  U32 max;
  U8 *sendbuf;

  switch (soc_state) {
    case 0:
      tcp_connect (tcp_soc, rem_IP, 1000, 0);
      bcount     = 0;
      wait_ack   = __FALSE;
      soc_state = 1;
      return;
```

36. Next, state 1 is waiting for the **TCP_EVT_CONNECT** event. This event is received in the **tcp_callback()** event callback function, which places the **send_data** process into state 2 (sending data state).
37. In state 2, allocate the maximum possible **size of transmit buffer**, fill it with some data, and **send** it. The maximum possible transmit buffer is allocated to reduce the number of packets and improve the transfer speed.

    After the packet is sent, wait for the **remote acknowledge** before proceeding with the next data packet.

```

    case 2:
      if (wait_ack == __TRUE) {
        return;
      }
      max = tcp_max_dsize (tcp_soc);
      sendbuf = tcp_get_buf (max);
      for (i = 0; i < max; i += 2) {
        sendbuf[i]   = bcount >> 8;
        sendbuf[i+1] = bcount & 0xFF;
        if (bcount >= 32768) {
          soc_state = 3;
          break;
        }
      }
      tcp_send (tcp_soc, sendbuf, i);
      wait_ack = __TRUE;
      return;
```

38. State 3 is achieved when the data transfer is finished. Wait for the last packet to be acknowledged and then **close** the TCP connection.

```

    case 3:
      if (wait_ack == __TRUE) {
        return;
      }
      tcp_close (tcp_soc);
      soc_state = 4;
      return;
  }
}
```

49. The embedded application waits for the TCP socket to **connect** before starting to send data. When the data packet is sent, the application waits for the **acknowledge** before creating and sending the next data packet. Use the callback **listener** function to wait for the remote acknowledge.

```

U16 tcp_callback (U8 soc, U8 event, U8 *ptr, U16 par) {
  /* This function is called on TCP event */

  switch (event) {
     ..
    case TCP_EVT_CONNECT:
      /* Socket is now connected and ready to send data. */
      soc_state = 2;
      break;
    case TCP_EVT_ACK:
      /* Our sent data has been acknowledged by remote peer */
      wait_ack = __FALSE;
      break;
       ..
  }
  return (0);
}
```

Note

- This assumes that the Network Interface Adapter is selected, enabled, and properly configured in the **Net_Config.c** configuration file.

- If the system runs out of TCP sockets, the application hangs in an endless loop in the system **error function** with the error code **ERR_TCP_ALLOC**.

Multiple TCP Connections

It is often required for the server applications to be able to accept several TCP connections from clients on the **same port**. Such applications are for example: Web server, FTP server, Telnet server etc. The multiplex handling must be implemented in the **session layer** of the user application.

Because TCP socket is a **connection-oriented** service, it accepts only one concurrent connection. The basic packet multiplexing is done in TCP Transport layer and the user application receives the **socket number** as a parameter in the callback function.

The framework of the user application shall contain the following basic functions:

1. The **user_init()** function to initialize all user application sessions at startup.

```

void user_init () {
  USER_INFO *user_s;
  int i;

  for (i = 0; i < user_num_sess; i++) {
    user_s = &user_session[i];
    user_s->Count = 0;
    user_s->Flags = 0;
    user_s->BCnt  = 0;
    user_s->Tout  = 0;
    user_s->File  = NULL;
    user_s->Script= NULL;
    /* Allocate a TCP socket for the session. */
    user_s->Socket = tcp_get_socket (TCP_TYPE_SERVER, TCP_TOS_NORMAL,
                     tcp_DefTout, user_listener);
    user_s->State = USER_STATE_ERROR;
    if (user_s->Socket != 0) {
      if (tcp_listen (user_s->Socket, USER_SERVER_PORT) == __TRUE) {
        user_s->State = USER_STATE_IDLE;
      }
    }
  }
}
```

All user sessions are now initialized and each session has allocated it's own TCP socket. A socket is listening on selected **USER_SERVER_PORT** port.

26. The **user_listener()** callback function for TCP socket. This callback function is common for all TCP sockets allocated in this user application.

```

static U16 user_listener (U8 socket, U8 event, U8 *ptr, U16 par) {
  USER_INFO *user_s;
  U8 session;
  int i;

  session = user_map_session (socket);
  if (session == 0)) {
    return (__FALSE);
  }
  user_s = &user_session[session-1];
  switch (event) {
    case TCP_EVT_CONREQ:
      if (user_s->State == USER_STATE_IDLE) {
        user_s->State = USER_STATE_RESERVED;
      }
      return (__TRUE);

    case TCP_EVT_ABORT:
      user_kill_session (user_s);
      return (__TRUE);

    case TCP_EVT_CONNECT:
      user_s->State = USER_STATE_ACTIVE;
      return (__TRUE);

    case TCP_EVT_CLOSE:
      user_kill_session (user_s);
      return (__TRUE);

    case TCP_EVT_ACK:
      user_s->Count += user_s->BCnt;
      user_s->BCnt = 0;
      return (__TRUE);

    case TCP_EVT_DATA:
       ..
      return (__TRUE);
  }
  return (__FALSE);
}
```

The **user_map_session()** function to map the socket, which has generated a callback event, to it's owner session.

```

static U8 user_map_session (U8 socket) {
  int i;

  for (i = 1; i <= user_num_sess; i++) {
    if (user_session[i-1].Socket == socket) {
      return (i);
    }
  }
  return (0);
}
```

The **user_kill_session()** function to initialize the session to a default state, close any eventually opened files and release any eventually allocated buffers.

```

static void user_kill_session (USER_INFO *user_s) {

  user_s->State =  USER_STATE_IDLE;
  if (user_s->Flags & USER_FLAG_FOPENED) {
    user_fclose (user_s->File);
    user_s->File = NULL;
  }
  if (user_s->Script != NULL) {
    free_mem (user_s->Script);
    user_s->Script = NULL;
  }
  user_s->Flags = 0;
  user_s->Count = 0;
  user_s->BCnt  = 0;
  user_s->Tout  = 0;
}
```

The **user_run_server()** function to maintain the application jobs, timeouts, etc. This function shall be frequently called from the main loop.

```

void user_run_server () {
  USER_INFO *user_s;
  int i;

  for (i = 0; i < user_num_sess; i++) {
    user_s = &user_session[i];

    switch (user_s->State) {
      case USER_STATE_IDLE:
      case USER_STATE_RESERVED:
        /* Keep TCP sockets listening. */
        if (tcp_get_state (user_s->Socket) < TCP_STATE_LISTEN) {
          tcp_listen (user_s->Socket, USER_SERVER_PORT);
        }
        break;
      case USER_STATE_WAITING:
        if (sec_tick == __TRUE) {
          if (--user_s->Tout == 0) {
            /* A timeout expired. */
            user_kill_session (user_s);
          }
        }
        break;
      case USER_STATE_ACTIVE:
         ..
        break;
    }
  }
}
```

Note

- There is one TCP socket used per user session. You need to reserve enough TCP sockets in **Net_Config.c** configuration file for all user sessions.

UDP Socket

The User Datagram Protocol (UDP) runs on top of the Internet Protocol (IP). The UDP protocol was developed for use by application protocols that do not require reliability, acknowledgment, or flow control features at the transport layer. It is simple and only provides transport layer addressing in the form of UDP ports and an optional checksum capability.

Opening UDP Connection

The User Datagram Protocol (UDP) is probably the simplest in all of TCP/IP. The UDP takes the application layer data passed to it, packages the layer in a simplified message format, and sends it to the IP for transmission. This handles the UDP sockets in a simple way. The UDP socket only needs to be opened for communication. It listens for incoming messages and sends outgoing messages on request.

Sending UDP Data

The UDP protocol is a simple byte stream service. It does not know anything about the format of the data being sent. It simply takes the data, **encapsulates** it into the UDP packet, and sends it to a remote peer.

The UDP protocol does not wait for any acknowledgement and is unable to detect any lost packets. When acknowledgement or detection is required, it must be done by the application layer. However, it is better to use a TCP socket for communication when acknowledgement is necessary.

When DHCP Enabled

A Dynamic Host Configuration Protocol (DHCP) client automatically configures the network parameters for the application. This normally takes some time after startup.

When the network traffic is low and a DHCP server is idle, automatic device configuration is finished in less than **60 msec**. But it is possible, on high traffic networks, that this configuration could take a lot longer by up to a couple of seconds. Any attempt to send an UDP data packet during that time will **fail** and the UDP data packet will be lost.

Communications must wait until the local IP address is configured. This can be done by simply monitoring the IP address in the **localm** structure, which holds all the network configuration parameters.

When the DHCP client starts, it copies a default Local IP address, which is set in the **configuration**, to a local buffer and clears the **assigned IP** address for the ethernet adapter. The DHCP client then tries to acquire the proposed IP address in the DHCP negotiation process. To see when the DHCP configuration procedure has finished, it is enough to monitor the assigned IP address of the ethernet adapter.

The whole procedure is required only when we want to send UDP data. For receiving UDP packets, this is not a problem because the application will not accept any IP packet until the ethernet adapter IP address is assigned.

Here is an example for the **send_data()** function from the **UDP example** modified for enabled DHCP:

```
void send_data (void) {
  static const U8 rem_IP[4] = {192,168,0,100};
  U8 *sendbuf;

  if (wait_ack == __TRUE) {
    return;
  }
  if (mem_test (localm[NETIF_ETH].IpAdr, 0, 4) == __TRUE) {
    /* IP address not yet assigned by DHCP. */
    return;
  }
  if (bindex < 128) {
    sendbuf = udp_get_buf (512);
    for (i = 0; i < 512; i += 2) {
      sendbuf[i]   = bcount >> 8;
      sendbuf[i+1] = bcount & 0xFF;
    }
    udp_send (udp_soc, rem_IP, 1000, sendbuf, 512);
  }
}
```

- The example assumes that the DHCP Client for Ethernet Network Interface is enabled in the **configuration**.

When ARP Cache Empty

The Address Resolution Protocol (ARP) module caches all received IP addresses to an internal cache buffer, which stores the IP addresses and Ethernet addresses (MAC).

When the application starts, the ARP Cache buffer is normally empty. The ARP module does not yet know the target MAC address for the first UDP data packet being sent from the application. It sends the ARP request to the network. The first and any subsequent UDP data packets sent from the user application are **lost** until the target MAC address is resolved. This is because the UDP does not buffer outgoing packets.

An ARP request must be sent to the network and the MAC address for the target IP address must be resolved before the first UDP data packet is sent to the network. This is only required if no packets were received from a destination IP. Every received IP or ARP packet is also processed by the ARP module, and the IP and MAC addresses are cached internally.

All cached IP addresses are by default **temporary IP** cache entries. After a timeout, which is set in the **configuration**, such entries are automatically deleted from the cache. You can use the function **arp_cache_ip()** to force an ARP request. You can also use the function to change the cache entry attribute to a **fixed IP** address rather than a temporary IP address.

Fixed IP entries are automatically refreshed by the ARP module on timeout. When timeout expires, the ARP module sends an ARP request to the target again to verify whether the target is still active and able to accept packets.

Once the function **arp_cache_ip()** returns the value of **__TRUE**, the remote IP address is resolved and cached in the ARP cache buffer. If the Cache Entry attribute is set to **ARP_FIXED_IP**, there is no need to take care of resolving the IP address when a timeout expires. The ARP module does this automatically.

Example for Sending Data

This example demonstrates the basic concept of sending a large amount of data using a UDP socket.

- The example sends 64 Kbytes to the remote IP address 192.168.0.100, which is listening on port 1000.
- The UDP socket will be allocated permanently and will not be released when the data is sent.
- The data is sent as a stream of bytes.
- The UDP socket does not wait for acknowledge from the remote peer to ascertain whether the data has been accepted. For this reason, a very simple acknowledge protocol is added to the example. The data will be sent in 512-byte blocks. Upon receiving the packet, the remote peer will send back a simple UDP packet with an acknowledgement. The acknowledgement is simply an index of the received packet starting with 0.

The steps for this example are as follows:

1. **Initialize** the system and **allocate** a free UDP Socket and **open** it for communication:

```

#include <RTL.h>

U8 udp_soc;
U16 bindex;
BOOL wait_ack;

void main (void) {
  init_TcpNet ();
  udp_soc = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS, udp_callback);
  udp_open (udp_soc, 0);
  bindex   = 0;
  wait_ack = __FALSE;
```

15. Run the **main thread** and call the **send_data()** function from an endless loop:

```

  while (1) {
    timer_poll ();
    main_TcpNet ();
    send_data ();
  }
}
```

23. The **send_data()** function **sends** UDP Data and waits for acknowledge. Note that the UDP sockets do not support any acknowledgment. The example provides its own acknowledgement.

```

void send_data (void) {
  static const U8 rem_IP[4] = {192,168,0,100};
  U8 *sendbuf;

  if (wait_ack == __TRUE) {
    return;
  }
  if (bindex < 128) {
    sendbuf = udp_get_buf (512);
    for (i = 0; i < 512; i += 2) {
      sendbuf[i]   = bcount >> 8;
      sendbuf[i+1] = bcount & 0xFF;
    }
    udp_send (udp_soc, rem_IP, 1000, sendbuf, 512);
  }
}
```

When the packet is sent, wait for the **remote acknowledge** before proceeding with the next data packet. Use the callback **listener** function to wait for the remote acknowledge.

```

U16 udp_callback (U8 socket, U8 *remip, U16 port, U8 *buf, U16 len)
  /* This function is called when UDP data has been received. */

  if ((len == 2) && (bindex == (buf[0]<<8 | buf[1]))) {
    wait_ack == __FALSE;
  }
  return (0);
}
```

Note

- The example assumes that the Network Interface Adapter is selected, enabled, and properly configured in the **Net_Config.c** configuration file.
- If the system runs out of UDP sockets, the application hangs in an endless loop in the system **error function** with the error code **ERR_UDP_ALLOC**.

IP Multicasting

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all the members of its destination host group in the same way as regular unicast datagrams.

A host can receive a multicast datagram when it is a member of a host group a datagram is destined to. To enable reception of multicast packets a host must first **join** a host group.

A membership of a host group is dynamic. Hosts may **join** and **leave** groups at any time.

Host groups are identified by class D IP adresses, those with "1110" as their high-order four bits. Host group addresses range from **224.0.0.0** to **239.255.255.255**. The address 224.0.0.0 is reserved and shall not be assigned to any group and 224.0.0.1 is assigned to the permanent group of all IP hosts, including gateways. This group shall also not be used for a dynamic host group.

UDP multicasting

Incoming multicast UDP datagrams are received by UDP socket in the same way as normal, unicast datagrams. The only difference is that the host must first **join** a host group, before the muticast packets for the selected group can be received.

Outgoing multicast UDP datagrams are **sent** in the same way as unicast datagrams, only the destination IP address is a group address instead of a host address.

Multiple UDP Connections

It is often required for the server applications to be able to accept several UDP connections from clients on the **same port**. Such application is for example a TFTP server. The multiplex handling must be implemented in the **session layer** of the user application.

Because UDP socket is a **connectionless** service, it is able to accept several concurrent connections from different remote hosts. The packet multiplexing must be done in user application.

The framework of the user application shall contain the following basic functions:

1. The **user_init()** function to initialize all user application sessions at startup.

```
void user_init () {
  USER_INFO *user_s;
  int i;

  for (i = 0; i < user_num_sess; i++) {
    user_s = &user_session[i];
    mem_set (user_s->RemIp, 0, IP_ADRLEN);
    user_s->RemPort = 0;
    user_s->Flags   = 0;
    user_s->Retries = 0;
    user_s->Tout    = 0;
    user_s->File    = NULL;
    user_s->State   = USER_STATE_IDLE;
  }
  /* Allocate one UDP socket for all sessions. */
  user_Socket = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS,
                                user_listener);
  if (user_Socket != 0) {
    udp_open (user_Socket, USER_SERVER_PORT);
  }
}
```

All user sessions are now initialized. A common UDP socket is opened for communication on selected **USER_SERVER_PORT** port.

24. The **user_listener()** callback function for UDP socket.

```
static U16 user_listener (U8 socket, U8 *remip, U16 port,
                          U8 *buf, U16 len) {
  USER_INFO *user_s;
  U8 session;
  int i;

  if (socket != user_Socket)) {
    return (__FALSE);
  }

  session = user_map_session (remip, port);
  if (session == 0) {
    return (__FALSE);
  }
  user_s = &user_session[session-1];
  switch (user_s->State) {
    case USER_STATE_IDLE:
      /* A new connection established. */
       ..
      user_s->State = USER_STATE_ACTIVE;
      break;

    case USER_STATE_ACTIVE:
      /* Process UDP data. */
       ..
      break;
  }
  return (__TRUE);
}
```

The **user_map_session()** function to map the UDP packet, which has generated a callback event, to it's owner session.

```

static U8 user_map_session (U8 *remip, U16 port) {
  USER_INFO *user_s;
  int i;

  /* Check if this is an existing connection. */
  for (i = 1; i <= user_num_sess; i++) {
    user_s = &user_session[i-1];
    if ((user_s->State > USER_STATE_IDLE) &&
        (mem_comp (remip, user_s->RemIp, IP_ADRLEN) == __TRUE) &&
        (port == user_s->RemPort)) {
      return (i);
    }
  }
  /* Check if this is a new connection. */
  for (i = 1; i <= user_num_sess; i++) {
    user_s = &user_session[i-1];
    if (user_s->State == USER_STATE_IDLE) {
      mem_copy (user_s->RemIp, remip, IP_ADRLEN);
      user_s->RemPort = port;
      return (i);
    }
  }
  return (0);
}
```

The **user_kill_session()** function to initialize the session to a default state, close any eventually opened files and release any eventually allocated buffers.

```

static void user_kill_session (USER_INFO *user_s) {

  user_s->State = USER_STATE_IDLE;
  if (user_s->File != NULL) {
    user_fclose (user_s->File);
    user_s->File = NULL;
  }
  mem_set (user_s->RemIp, 0, IP_ADRLEN);
  user_s->RemPort = 0;
  user_s->Flags   = 0;
  user_s->Retries = 0;
  user_s->Tout    = 0;
}
```

The **user_run_server()** function to maintain the application jobs, timeouts, etc. This function shall be frequently called from the main loop.

```

void user_run_server () {
  USER_INFO *user_s;
  int i;

  for (i = 0; i < user_num_sess; i++) {
    user_s = &user_session[i];

    switch (user_s->State) {
      case USER_STATE_ACTIVE:
        if (sec_tick == __TRUE) {
          if (--user_s->Tout == 0) {
            /* A timeout expired. */
            user_kill_session (user_s);
          }
        }
        break;
    }
  }
}
```

Note

- There is only **one socket** used for all user sessions.

Configuring TCPnet

RL-TCPnet is easy to customize for each application you create. The configuration settings are collected and moved to the **Net_Config.c** system configuration file. Most of the configuration options are set at compile time and integrated into the application code.

Each of the embedded device that is connected to a local area network must have a unique MAC address, IP address, and host name. Hence, the compile time configuration in Net_Config.c is not sufficient when several embedded ethernet devices are produced. For this reason, RL-TCPnet provides **runtime configuration** of certain ethernet network parameters. The runtime configuration allows you to dynamically configure the embedded system when the system has already started. The configuration parameters can be stored to the system EEPROM or some other system NV memory.

Static Configuration

RL-TCPnet must be configured for the embedded applications you create. All configuration settings are found in the **Net_Config.c** file, which is located in the **\Keil\ARM\RL\TCPnet\User** folder. Configuration options allow you to configure:

System Settings

- Local Host Name
- Memory Pool Size
- System Tick Timer Interval
- Code for the System Error Function

Ethernet Network Interface

- Enable or Disable Ethernet interface
- Local MAC Address
- Local IP Address
- Subnet Mask
- Default Gateway IP Address
- Primary DNS Server
- Secondary DNS Server
- Enable or Disable IGMP protocol
- IGMP Membership Table size
- Enable or Disable NetBIOS Name Service
- Enable or Disable Dynamic Host Configuration
- DHCP Vendor Class Identifier
- ARP Cache Table Size
- ARP Cache Entry Timeout
- Number of Retries to resolve an IP Address
- ARP Resend Timeout
- Enable or Disable Gratuitous ARP Notification

PPP Network Interface

- Enable or Disable PPP interface
- Local IP Address
- Subnet Mask
- Primary DNS Server
- Secondary DNS Server
- Enable or Disable Obtain Client IP address automatically
- Enable or Disable default Gateway on remote Network
- Async Control Character Map
- Retransmissions and Timeouts

SLIP Network Interface

- Enable or Disable SLIP interface
- Local IP Address
- Subnet Mask
- Primary DNS Server
- Secondary DNS Server
- Enable or Disable default Gateway on remote Network

UDP Settings

- Enable or Disable the UDP protocol
- Number of available UDP Sockets

TCP Settings

- Enable or Disable the TCP protocol
- Number of available TCP Sockets
- Number of Retries to Resend the TCP data packet
- TCP Resend Timeout
- Default Connection Timeout for the Keep-alive Timer

HTTP Server Settings

- Enable or Disable the HTTP Server
- Number of HTTP Sessions

- Port Number
- Enable or Disable HTTP Authorization
- Realm string for Authorization
- Username
- Default Password

Telnet Server Settings

- Enable or Disable the Telnet Server
- Number of Telnet Sessions
- Port Number
- Enable or Disable Telnet Authorization
- Username
- Default Password

TFTP Server Settings

- Enable or Disable the TFTP Server
- Number of TFTP Sessions
- Port Number
- Inactive Session Timeout
- Number of Retries

FTP Server Settings

- Enable or Disable the FTP Server
- Number of FTP Sessions
- Port Number
- Enable or Disable FTP Authorization
- Username
- Default Password

DNS Client Settings

- Enable or Disable a DNS Client
- DNS Cache Table Size

SMTP Client Settings

- Enable or Disable a SMTP Client
- SMTP Inactive Timeout

SNMP Agent Settings

- Enable or Disable a SNMP Agent
- Community Name
- Port Number
- Trap Port Number
- IP addres of Trap Server

You must copy the **Net_Config.c** file to your project folder and add it to your project. To customize the configuration, you must change the settings specified in **Net_Config.c**

Note

- The **Configuration Wizard** feature, in µVision, helps you to make the configuration changes easily with just a couple of mouse clicks and provides a good overview of the complete configuration.

System

The System configuration allows you to modify the system parameters such as:

- **Local Host Name** specifies the name for your hardware. You can access your hardware by this name. This name is also used as a host name when sending emails with the SMTP Client. The name length is limited to 15 characters.

```
#define LHOST_NAME     "mcb2100"
```

- **Memory Pool size** specifies the amount of RAM allocated for the Memory Pool. The memory blocks are dynamically allocated and released by the system Memory Management Functions.

  Memory size is specified in 4-byte words, however the µVision Configuration Wizard converts this value to number of bytes. For example the value 2000 for MEM_SIZE specifies the memory pool of 8000 bytes.

```
#define MEM_SIZE       2000
```

- **Tick Timer interval** specifies the system tick timer interval. It is specified in milliseconds. Allowed values are: 10ms, 20ms, 25ms, 40ms, 50ms, 100ms, or 200 ms. You should set this value as high as possible.

```
#define TICK_INTERVAL  100
```

Note

- You can start with the default values specified in the **Net_Config.c** configuration file. The system error function **sys_error()** is also provided in the configuration file. This function gets called on system critical errors. If you get this function called in your application, check the **error code**.
- For error code **ERR_MEM_ALLOC**, you need to increase the memory size for the Memory Pool. When the error code is **ERR_TCP_ALLOC**, you need to increase the number of available TCP sockets etc.

Ethernet Interface

The Ethernet Network Interface can be enabled or disabled. The network settings can be specified for Local Ethernet Address, Local IP Address, Default Gateway IP Address, and Net Mask.

The network settings set here are valid for Ethernet Network Interface only. They must be set for each network interface separately. You must carefully configure those parameters to match the settings of your local LAN.

- **Ethernet Network Interface** switch enables or disables the Ethernet Network interface. Ethernet Interface should be disabled only when PPP or SLIP interface is used to reduce the code size. When this value is set to 1 the Ethernet Interface is enabled.

```
#define ETH_ENABLE     1
```

- **MAC Address** specifies a six byte Local Ethernet MAC address. It must be unique for each ethernet controller located in your local area network.

```
#define _MAC1          0x1E
#define _MAC2          0x30
#define _MAC3          0x6C
#define _MAC4          0xA2
#define _MAC5          0x45
#define _MAC6          0x5E
```

- **IP Address** specifies your local static four byte IP address.

```
#define _IP1           192
#define _IP2           168
#define _IP3           0
#define _IP4           100
```

- **Subnet Mask** specifies the Net Mask. This is normally the class C for small LANs: **255.255.255.0**. It is used by the system to check if the given IP address belongs to an internal LAN or an external WAN.

```
#define _MSK1          255
#define _MSK2          255
#define _MSK3          255
#define _MSK4          0
```

- **Default Gateway** specifies the IP address of the default gateway. It is used when the external WAN is accessed. If your application is to be used only on a local LAN, then you don't have to specify the Default Gateway.

```
#define _GW1           192
#define _GW2           168
#define _GW3           0
#define _GW4           254
```

- **Primary DNS Server** specifies the IP address of the primary DNS Server. The DNS Client sends IP address resolution requests to this address. This setting is irrelevant when the DNS Client is disabled by configuration.

```
#define _pDNS1         194
#define _pDNS2         25
#define _pDNS3         2
#define _pDNS4         129
```

- **Secondary DNS Server** specifies the IP address of the secondary DNS Server. This DNS Server is used when the primary DNS Server is down or not accessible. In this case, the DNS Client automatically switches to a backup secondary DNS Server if a non-zero address is provided.

```
#define _sDNS1          194
#define _sDNS2          25
#define _sDNS3          2
#define _sDNS4          130
```

- **NetBIOS Name Service** switch enables or disables the NBNS Name Service. When NBNS is enabled, you can access your hardware device by the **name** of your LAN. For example you can type: PING mcb2100 instead of PING 192.168.0.100. NBNS is enabled when this value is set to 1.

```

#define NBNS_ENABLE    1
```

- **Dynamic Host Configuration** switch enables or disables the DHCP Service. When DHCP is enabled, your hardware device obtains all the network parameters like IP address, net mask, default gateway, primary and secondary DNS servers, automatically from your DHCP server on the LAN. Dynamic Host Configuration does not work without a DHCP Server running on your LAN. You must also enable the **NBNS Service** to access the hardware by name. A dynamically assigned IP address is normally not known or can even change during the lifetime of the connection.

```

#define DHCP_ENABLE    1
```

- **Vendor Class Identifier** specifies a string, which is optionally added to DHCP request message. If this is an empty string, the VCI option is not added. The Vendor Class Identifier can be used to selectively identify a device on DHCP server. For example: DHCP server can assign an IP address to a specific Vendor Class group only and ignore all other DHCP clients with different VCI.

```

#define DHCP_VCID      ""
```

Note

- If you want to use a **static IP** for your application in a network where a DHCP address assignment is used, disable the DHCP client in the configuration and set the static IP address out of the range of the DHCP Server address pool.
- You can use **Ethernet** and **PPP** (or SLIP) network interfaces **simultaneously**.

ARP

The Ethernet Address Resolution Protocol module (ARP) **caches** the addresses of remote peers. This greatly improves the system performance. The following ARP parameters can be configured:

- **Cache Table size** specifies the size of ARP cache table. It defines how many cache entries may be kept in the ARP cache. You must increase this value if you are using multiple simultaneous IP connections.

```
#define ARP_TABSIZE    10
```

- **Cache Timeout in seconds** specifies the timeout for an ARP cache entry. After a timeout, the Permanent IPs are refreshed, and the Temporary IPs are removed from the cache. The Timeout (Keep-alive) Timer is reset on every access to the cache entry. This means when an ethernet packet is received from a remote peer.

```
#define ARP_TIMEOUT    150
```

- **Number of Retries** is the maximum number of retries to resolve the ethernet MAC address of the remote peer before the ARP module gives up. When the ARP request is not responded to by the remote peer, the ARP module sends another request to retrieve the remote peer's ethernet MAC address. This process is repeated ARP_MAXRETRY times before the ARP module gives up.

```
#define ARP_MAXRETRY   4
```

- **Resend Timeout in seconds** specifies the resend timeout. When this timeout has expired and no response has been received from the remote peer, an ARP module resends the ARP request.

```
#define ARP_RESEND     2
```

- **Send Notification on Address changes** switch enables or disables the Gratuitous ARP Service. When it is enabled, the embedded host will broadcast a Gratuitous ARM notification at startup, or when the local IP Address has changed.

```
#define ARP_NOTIFY     0
```

IGMP

The Internet Group Management Protocol (IGMP) is required for sending and receiving IP Multicast packets. In order to recive Multicast packets, a host must first join a Host Group with specified Host Group IP address. The following IGMP parameters can be configured:

- **IGMP Group Management** switch enables or disables the IGMP protocol with IP Multicasting. It is enabled when this value is set to 1.

```
#define IGMP_ENABLE    1
```

- **Membership Table size** specifies the size of IGMP Host Group table. It defines how many Host Groups a host can join to. A default value of 5 means that a host can be a member of max. 5 different Host Groups.

```
#define IGMP_TABSIZE   5
```

PPP Interface

The **P**oint to **P**oint **P**rotocol (PPP) Network Interface can be enabled or disabled. The other network settings you can configure are Local IP Address, Net Mask, Primary DNS, Secondary DNS Server, and Async Control Character map.

The network settings in this topic are only valid for the PPP Interface. They must be set for each network interface separately. You must carefully configure those parameters to match the settings of your remote peer. PPP Network Interface supports some of the automatic configuration protocols.

The **network settings** are used to **route packets** from the stack to the proper network interface on transmit.

- **PPP Network Interface** switch enables or disables the PPP Network interface. When the PPP Interface is not used, it should be disabled to reduce the application code size. It is enabled when this value is set to 1.

```
#define PPP_ENABLE     1
```

- **IP Address** specifies your local static four-byte IP address.

```
#define _IP1P          192
#define _IP2P          168
#define _IP3P          125
#define _IP4P          1
```

- **Subnet Mask** specifies the Net Mask. This is normally class C for small LANs: **255.255.255.0**. The system uses it to route packets to network interfaces.

```
#define _MSK1P         255
#define _MSK2P         255
#define _MSK3P         255
#define _MSK4P         0
```

- **Primary DNS Server** specifies the IP address of the primary DNS Server. The DNS Client sends IP address resolution requests to this address. This setting is irrelevant when the DNS Client is disabled by configuration.

```
#define _pDNS1P        194
#define _pDNS2P        25
#define _pDNS3P        2
#define _pDNS4P        129
```

- **Secondary DNS Server** specifies the IP address of the secondary DNS Server. This DNS Server is used when the primary DNS Server is down or not accessible. In this case, the DNS Client automatically switches to a backup secondary DNS Server if a non-zero address is provided.

```
#define _sDNS1P        194
#define _sDNS2P        25
#define _sDNS3P        2
#define _sDNS4P        130
```

- **Obtain Client IP address automatically** option applies when the PPP dial-up connection dials to a remote PPP Server. If it is enabled, the local PPP IP address is obtained from the remote server automatically.

  When this option is **enabled** the **Local IP address** and **Subnet Mask** configuration settings are ignored. The Subnet Mask is automatically set to 255.255.255.255 when the PPP link is connected.

```
#define PPP_GETIP      1
```

- **Use Default Gateway on remote Network** option applies when both Ethernet and PPP dial-up networks are used simultaneously. When TCPNet finds a packet with a destination

IP address does not belong to either the LAN or the PPP interface, this option determines which gateway to use.

If this option is disabled, these packets are forwarded to the default gateway on the LAN. If this option is enabled, these packets are forwarded to the Dial-up PPP network.

```
#define PPP_DEFGW       1
```

- **Async Control Character Map** specifies the Map of Control Characters that are transmitted with an escape character. All 32 control characters are mapped into a 4-byte map table. For example:
  - ASCII control character NULL (0) is mapped to the least significant bit of _ACCM4.
  - Control character STX (1) is mapped to bit 1 of _ACCM4.
  - Control character CR (13) is mapped to bit 4 of _ACCM3.

  By default, no control characters are mapped.

  ```
  #define _ACCM1          0x00
  #define _ACCM2          0x00
  #define _ACCM3          0x00
  #define _ACCM4          0x00
  ```

  When the XON/XOFF protocol is used for the flow control, the data bytes 0x11 and 0x13 should never be sent in a packet. Those values are used to start and stop the RS232 stream. They must be sent as a 2 byte sequence containing the ESCAPE character followed by the XON/XOFF character xor-ed with 0x20.
  - **XON** (17 = 0x11) is sent as 0x7D, 0x31
  - **XOFF** (19 = 0x13) is sent as 0x7D, 0x33

  Control Character Map for XON/XOFF protocol would then be:

  ```
  #define _ACCM1          0x00
  #define _ACCM2          0x0A
  #define _ACCM3          0x00
  #define _ACCM4          0x00
  ```

- **Retransmissions and Timeouts** configures timeouts and the number of retransmissions for various PPP protocols. When using slow serial links (for example a GSM data link), the timeout values should be increased if the PPP connect fails.

  Link Control Protocol (**LCP**):
  - **LCP Number of Retries** specifies the number of retransmissions before the LCP module gives up.

    ```
    #define LCP_MAXRETRY    2
    ```

  - **LCP Retry Timeout in seconds** is the timeout after which the LCP module retransmits the packet.

    ```
    #define LCP_RETRYTOUT   2
    ```

  Password Authentication Protocol (**PAP**):
  - **PAP Number of Retries** specifies the number of retransmissions before the PAP module gives up.

    ```
    #define PAP_MAXRETRY    3
    ```

  - **PAP Retry Timeout in seconds** is the timeout after which the PAP module retransmits the packet.

    ```
    #define PAP_RETRYTOUT   3
    ```

  Internet Protocol Control Protocol (**IPCP**):
  - **IPCP Number of Retries** specifies the number of retransmissions before the IPCP module gives up.

    ```
    #define IPCP_MAXRETRY   3
    ```

  - **IPCP Retry Timeout in seconds** is the timeout after which the IPCP module retransmits the packet.

```

#define IPCP_RETRYTOUT 2
```

Note

- If both PPP and Ethernet Interfaces are used simultaneously, you must not set the same network group address for both the Ethernet and PPP interfaces.

SLIP Interface

The **S**erial **L**ine **IP** (SLIP) protocol is a very simple way to transmit IP packets over a serial line. It does not provide any framing or error control and is therefore not widely used today.

You can enable or disable the SLIP interface. You can also configure the Local IP Address, Net Mask, Primary DNS, and Secondary DNS Server IP address.

Network settings in this topic are valid for the SLIP Interface only. They must be set separately for each network interface. You must carefully configure those parameters to match the settings of your remote peer. SLIP protocol does not support any of the automatic configuration protocols.

The **network settings** are used to **route packets** from the stack to the proper network interface on transmit.

- **SLIP Network Interface** switch enables or disables the SLIP Network interface. When the SLIP Interface is not used, it should be disabled to reduce the application code size. It is enabled when this value is set to 1.

```
#define SLIP_ENABLE    1
```

- **IP Address** specifies your local static four byte IP address.

```
#define _IP1S          192
#define _IP2S          168
#define _IP3S          225
#define _IP4S          1
```

- **Subnet Mask** specifies the Net Mask. This is normally class C for small LANs: **255.255.255.0**. It is used by the system to route packets to network interfaces.

```
#define _MSK1S         255
#define _MSK2S         255
#define _MSK3S         255
#define _MSK4S         0
```

- **Primary DNS Server** specifies the IP address of the primary DNS Server. The DNS Client sends IP address resolution requests to this address. This setting is irrelevant when the DNS Client is disabled by configuration.

```
#define _pDNS1S        194
#define _pDNS2S        25
#define _pDNS3S        2
#define _pDNS4S        129
```

- **Secondary DNS Server** specifies the IP address of the secondary DNS Server. This DNS Server is used when the primary DNS Server is down or not accessible. In this case, the DNS Client automatically switches to a backup secondary DNS Server if a non-zero address is provided.

```
#define _sDNS1S        194
#define _sDNS2S        25
#define _sDNS3S        2
#define _sDNS4S        130
```

- **Use Default Gateway on remote Network** option applies when both Ethernet and SLIP dial-up networks are used simultaneously. If enabled, data that cannot be sent to a local LAN is forwarded to the Dial-up SLIP network instead.

```
#define SLIP_DEFGW     1
```

Note

- If both SLIP and Ethernet Interfaces are used simultaneously, you must not set the same network group address for both the Ethernet and SLIP interfaces.

UDP Socket

The User Datagram Protocol (UDP) is unreliable and only provides simple transport layer addressing to transfer data streams. Because of it simplicity and small protocol overhead, it is suitable for direct peer-to-peer communication on local LANs.

The following UDP options are configurable:

- **UDP Sockets** switch enables or disables the UDP Socket service in your application. It is enabled when this value is set to 1. It should be set to 0 when the UDP connections are not used.

```
#define UDP_ENABLE     1
```

- **Number of UDP Sockets** specifies the number of available UDP sockets. This number is usually set to the maximum number of simultaneously opened UDP connections.

```
#define UDP_NUMSOCKS   5
```

Note

- When the UDP Sockets are **not enabled**, the ARM linker does **not link** the UDP support modules to your application and thus reduces the code size and memory usage.

TCP Socket

The Transmission Control Protocol (TCP) is a connection oriented reliable protocol. TCP is much more complex than UDP and introduces more protocol overhead. It implements a number of protocol timers to ensure reliable and synchronized communication between the two end systems. It is thus suitable to use on WANs.

The following TCP options are configurable:

- **TCP Sockets** switch enables or disables the TCP Socket service in your application. It is enabled when this value is set to 1. It should be set to 0 when the TCP connections are not used.

```
#define TCP_ENABLE     1
```

- **Number of TCP Sockets** specifies the number of available TCP sockets. This number is usually set to the maximum number of simultaneously opened TCP connections.

```
#define TCP_NUMSOCKS   5
```

- **Number of Retries** specifies the number of retransmissions before the TCP module gives up. Data is retransmitted if it is not acknowledged within the timeout frame defined by the TCP_RETRYTOUT.

```
#define TCP_MAXRETRY   5
```

- **Retry Timeout in seconds** is the timeout, after which the TCP module retransmits the data.

```
#define TCP_RETRYTOUT  4
```

- **Default Connect Timeout in seconds** is the default Keep Alive timeout. After this timeout has expired, the TCP link is disconnected. This parameter is used by applications like HTTP Server and Telnet Server.

```
#define TCP_DEFTOUT    120
```

Note

- When the TCP Sockets are **not enabled**, the ARM linker does **not link** the TCP support modules to your application and thus reduces the code size and memory usage.

HTTP Server

RL-TCPnet has integrated a tiny Hypertext Transfer Protocol Server (HTTP) that can be used to transfer integrated web pages to a remote client.

The following HTTP Server options are configurable:

- **HTTP Server** switch enables or disables the HTTP Server service in your application. It is enabled when this value is set to 1. It should be set to 0 when HTTP server is not used in your application.

```
#define HTTP_ENABLE    1
```

- **Number of HTTP Sessions** specifies the number of available HTTP sessions. You must increase this number if the web page has many objects (like gif or jpeg images). The remote HTTP client, sometimes called the web browser, opens multiple connections when downloading such pages.

```
#define HTTP_NUMSESS   3
```

- **Port Number** specifies the listening TCP port number. The default HTTP server listening port is 80.

```
#define HTTP_PORTNUM   80
```

- **Enable User Authentication** switch enables or disables the WEB Server authentication with a **username** and a **password**. The user authentication is enabled, when this value is set to 1.

```
#define HTTP_ENAUTH    1
```

- **Authentication Realm string** is the string which displays in the internet browser's authentication dialog when authentication is required. This is a zero terminated string.

```
#define HTTP_AUTHREALM "Embedded WEB Server"
```

- **Authentication Username** is the **username** identification.

```
#define HTTP_AUTHUSER  "admin"
```

- **Authentication Password** is the default **password**, which is an **empty string** that must be stored in Non volatile memory. The user can change the password later.

```
#define HTTP_AUTHPASSW ""
```

Note

- When the HTTP server is **not enabled**, the ARM linker does **not link** the HTTP support modules to your application and thus reduces the code size and memory usage.

Telnet Server

RL-TCPnet has integrated a tiny Telnet Server, which is a command-line oriented application that can be used to remotely connect to your device with the Telnet client.

The following Telnet Server options are configurable from the Net_Config.c configuration file:

- **Telnet Server** switch enables or disables the Telnet Server service in your application. It is enabled when this value is set to 1. It should be set to 0 when Telnet server is not used in your application.

  ```
  #define TNET_ENABLE    1
  ```
- **Number of Telnet Connections** specifies the number of available Telnet sessions. The default value is one, and this enables only one concurrent client connection. You should increase this number if multiple Telnet clients must connect to the Telnet server at the same time.

  ```
  #define TNET_NUMSESS   1
  ```
- **Port Number** specifies the listening TCP port number. The default Telnet server listening port is 23.

  ```
  #define TNET_PORTNUM   23
  ```
- **Enable User Authentication** switch enables or disables the Telnet Server authentication with a **username** and a **password**. The user authentication is enabled when this value is set to 1.

  ```
  #define TNET_ENAUTH    1
  ```
- **Authentication Username** is the **username** identification.

  ```
  #define TNET_AUTHUSER  "admin"
  ```
- **Authentication Password** is the default **password**, which is an **empty string** that must be stored in Non volatile memory. The user may change the password later.

  ```
  #define TNET_AUTHPASSW ""
  ```

Note

- You must also add the **Telnet_uif.c** user interface module to your project and customize it. This file is in the **\Keil\ARM\RL\TCPnet\User** folder.
- When the Telnet server is **not enabled**, the ARM linker does **not link** the Telnet support modules to your application and thus reduces the code size and memory usage.

TFTP Server

The Trivial File Transfer Protocol (TFTP) is a very simple protocol used to transfer files. Using this protocol, you can send files to the System or read files from it. This protocol can be used, for example, for Firmware Upgrade.

The following TFTP Server options are configurable from the **Net_Config.c** configuration file:

- **TFTP Server** switch enables or disables the TFTP Server service in your application. It is enabled when this value is set to 1. It should be seto to 0 when TFTP server is not used in your application.

  ```
  #define TFTP_ENABLE    1
  ```

- **Number of TFTP Sessions** specifies the number of available TFTP sessions. The default value is one, and this enables only one concurrent client connection. You should increase this number if multiple TFTP clients must connect to the TFTP server at the same time.

  ```
  #define TFTP_NUMSESS   1
  ```

- **Port Number** specifies the listening UDP port number. The default TFTP server listening port is 69.

  ```
  #define TFTP_PORTNUM   69
  ```

- **Inactive Session Timeout in seconds** is an inactive session timeout. The TFTP session closes if the TFTP file transfer is interrupted for some reason and the timeout timer expires.

  ```
  #define TFTP_DEFTOUT   15
  ```

- **Number of Retries** specifies how many times the TFTP Server tries to retransmit the data before giving up.

  ```
  #define TFTP_MAXRETRY  4
  ```

Note

- You also need to add the **TFTP_uif.c** user interface module to your project and customize it. This file is in the **\Keil\ARM\RL\TCPnet\User** folder.
- When the TFTP server is **not enabled**, the ARM linker does **not link** the TFTP support modules to your application and thus reduces the code size and memory usage.

FTP Server

The File Transfer Protocol (FTP) is a protocol used to transfer and manage files and folders. Using this protocol, you can send files to the System or read files from it. You can also create and delete folders and rename files or folders on the system.

The following FTP Server options are configurable from the **Net_Config.c** configuration file:

- **FTP Server** switch enables or disables the FTP Server service in your application. It is enabled when this value is set to 1. It should be set to 0 when FTP server is not used in your application.

```
#define FTP_ENABLE    1
```

- **Number of FTP Sessions** specifies the number of available FTP sessions. The default value is one, and this enables only one concurrent client connection. You should increase this number if multiple FTP clients must connect to the Telnet server at the same time.

  If you are using Windows Explorer as a FTP client, you should also increase this number because the Windows Explorer creates multiple simultaneous connections.

```
#define FTP_NUMSESS   3
```

- **Port Number** specifies the listening TCP port number. The default FTP server listening port is 21.

```
#define FTPT_PORTNUM  21
```

- **Enable User Authentication** switch enables or disables the FTP Server authentication with a **username** and a **password**. The user authentication is enabled when this value is set to 1.

```
#define FTP_ENAUTH    1
```

- **Authentication Username** is the **username** identification.

```
#define FTP_AUTHUSER  "admin"
```

- **Authentication Password** is the default **password**, which is an **empty string** that must be stored in Non volatile memory. The user may change the password later.

```
#define FTP_AUTHPASSW ""
```

Note

- You must also add the **FTP_uif.c** user interface module to your project and customize it. This file is in the **\Keil\ARM\RL\TCPnet\User** folder.
- One FTP session uses **two** TCP **sockets**, one control and one data socket. Have this in mind when you are configuring the number of TCP sockets.
- When the FTP server is **not enabled**, the ARM linker does **not link** the FTP support modules to your application and thus reduces the code size and memory usage.

DNS Client

The DNS Client allows you to resolve IP addresses of internet hosts that are identified by a **host name**. The address is resolved using a protocol in which a piece of information is sent by a client process (executing on the local computer) to a server process (executing on a remote computer). The address resolution procedure is completed when the client receives a response from the server containing the required address.

- **DNS Client** switch enables or disables the DNS Client service in your application. It is enabled when this value is set to 1. It should be set to 0 when the DNS Client is not used.

```
#define DNS_ENABLE     1
```

- **Cache Table size** specifies the size of the DNS Cache by defining the number of entries for the DNS Cache table. When the IP address is resolved, it is also stored to the local cache.

  When a request for resolving an IP address is received, the DNS Client first checks the local cache memory. If a valid entry is found there, the IP address is taken from the cache, and the request is not sent on to the remote DNS Server.

```
#define DNS_TABSIZE    20
```

Note

- DNS Cache entries expire after a **Time to Live** (TTL) timeout. This is defined by the DNS Server. The TTL value for resolved IP addresses is received in an answer packet from the DNS Server. The DNS Client manages the timeouts. When a timeout counter **expires**, the DNS Cache entry is **deleted** from the Cache.

SMTP Client

The SMTP Client sends email messages from the embedded system to a designated email address, which can be either local or remote. It can send log reports using dynamic email messages or trouble reports for a variety of predetermined conditions.

The following SMTP Client options are configurable from the **Net_Config.c** configuration file:

- **SMTP Client** switch enables or disables the SMTP Client service in your application. It is enabled when this value is set to 1. It should be set to 0 when the SMTP Client is not used in your application.

```
#define SMTP_ENABLE     1
```

- **Response Timeout in seconds** is an inactivity timeout. When the SMTP Client does not receive a response from SMTP Server within this timeout, it **aborts** the operation.

```
#define SMTP_DEFTOUT   20
```

SNMP Agent

The SNMP Agent manages a set of **managed objects** defined by the user in **SNMP_MIB.c** interface module. It can be used to control various system settings or IO peripherals from the SNMP manager.

The following SNMP Agent options are configurable from the **Net_Config.c** configuration file:

- **SNMP Agent** switch enables or disables the SNMP Agent service in your application. It is enabled when this value is set to 1. It should be set to 0 when the SNMP Agent is not used in your application.

```
#define SNMP_ENABLE     1
```

- **Community Name** specifies the SNMP Community where an SNMP message is destined for. Only the members of the same community can communicate with each other using SNMP protocol. Default Community name is **public**.

```
#define SNMP_COMMUNITY "public"
```

- **Port Number** specifies the listening UDP port number. The default SNMP Agent listening port is 161.

```
#define SNMP_PORTNUM   161
```

- **Trap Port Number** specifies the UDP port number for Trap operations. The default SNMP Agent trap port is 162.

```
#define SNMP_TRAPPORT  162
```

- **Trap Server** specifies the IP address of the Trap Server which receives Trap messages. This IP address is used when the Trap Server IP is not specified in **snmp_trap()** function parameter.

```
#define SNMP_TRAPIP1   192
#define SNMP_TRAPIP2   168
#define SNMP_TRAPIP3   0
#define SNMP_TRAPIP4   1
```

Error Function

Various **errors** can cause the system to crash when the system is running. For critical errors, the system calls the system error function **sys_error()**. This makes is possible to catch the exception and possibly recover from it.

The system can, for example, run out of memory or fail to allocate the TCP socket. These are usually an indication that the system configuration is wrong and needs to be corrected. The error function alerts the developer as to what requires changing.

The parameter **code** holds the **error code** of the exception. This can have the following values:

Error Code	Type of Error	Description
ERR_MEM_ALLOC	Out of memory	This is normally the case when the memory pool size is too small. You must increase the size of the memory pool in the configuration.
ERR_MEM_FREE	Invalid memory block release	Possible reason is that the link pointers were overwritten by the user. This can happen when the user application has written buffer data out of boundaries of the allocated memory block.
ERR_MEM_CORRUPT	Link pointer corrupted	The system memory link is corrupted and points to an odd address. Resuming the program execution would cause the B-Class Trap with Illegal Word Operand Access. Possible reason is writing buffer data out of boundaries of the allocated memory block.
ERR_UDP_ALLOC	No free UDP Sockets	The system has run out of UDP sockets. You must increase the number of available UDP sockets in the configuration.
ERR_TCP_ALLOC	No free TCP Sockets	The system has run out of TCP sockets. You must increase the number of available TCP sockets in the configuration.
ERR_TCP_STATE	Undefined State	TCP socket is in an undefined state. This can happen when the system memory is accidentally overwritten by the user application.

This is an example of the error function which can be customized. This function currently does nothing but stop the TCPnet system by running in an endless loop.

```
void sys_error (ERROR_CODE code) {
  /* This function is called when a fatal error is encountered. The normal */
  /* program execution is not possible anymore. Add your critical error    */
  /* handler code here.                                                    */

  switch (code) {
    case ERR_MEM_ALLOC:
      /* Out of memory. */
      break;

    case ERR_MEM_FREE:
      /* Trying to release non existing memory block. */
      break;

    case ERR_MEM_CORRUPT:
      /* Memory Link pointer is Corrupted. */
      /* More data written than the size of allocated mem block. */
      break;

    case ERR_UDP_ALLOC_SOCKET:
      /* Out of UDP Sockets. */
      break;
    case ERR_TCP_ALLOC_SOCKET:
      /* Out of TCP Sockets. */
      break;

    case ERR_TCP_STATE_UNDEFINED:
      /* TCP State machine in undefined state. */
      break;
  }

  /* End-less loop */
  while (1);
}
```

Runtime Configuration

Each embedded ethernet device must have a unique **MAC address**, **IP address**, and **host name**. This is very important when multiple devices are connected to the same LAN. Otherwise, network collisions might occur, network communications on local LAN might be disturbed, and the system might not work.

You can use the same application code for serial production of embedded devices. The **runtime configuration** feature allows you to read configuration parameters from the EEPROM and configure the ethernet network interface for each embedded device differently.

- The **MAC address** is written to the ethernet controller registers when the controller initializes (when calling the function **init_TcpNet()**). For this reason, the variable **own_hw_adr[6]** must be set before the system initializes.
- The **ethernet interface** configuration parameters must be set after the system initializes. The structure **localm[0]** can simply be overwritten with new values.
- The **local host name** can be changed by overwriting the default value, which is set in the **Net_Config.c** system configuration file. The local host name accesses the embedded system by the name instead of the IP address.
- **Dynamic Host Configuration** can be disabled at runtime. In this case, user provided network parameters defined in the **Net_Config.c** configuration file are used instead. The DHCP Client can be disabled by calling the function **dhcp_disable()** after the system initializes. The DHCP Client must be **enabled** in the configuration so that the DHCP Client code links to the application code.

Here is an example of dynamic system configuration:

```
#include <Net_Config.h>

extern U8 own_hw_adr[];
extern U8 lhost_name[];
extern LOCALM localm[];

/* The following definitions should be read out of EEPROM. */
U8 const mac_adr[6] = { 0,1,2,50,60,70 };
LOCALM const ip_config = {
  { 192,168,0,150 },             // IP address
  { 192,168,0,1 },               // Default Gateway
  { 255,255,255,0 },             // Net mask
  { 194,25,2,129 },              // Primary DNS server
  { 194,25,2,130 }               // Secondary DNS server
};
U8 const DHCP_mode = 0;
U8 const dev_name[16] = { "Keil_MCB" };

void main (void) {
  /* Main Thread of the TCPnet. */

  /* Change the MAC address for the ethernet controller. */
  mem_copy (own_hw_adr, mac_adr, 6);

  init_TcpNet ();

  /* Change the Ethernet IP configuration. */
  if (DHCP_mode == 0) {
    dhcp_disable ();
  }
  str_copy (&lhost_name, &dev_name);
  mem_copy (&localm[NETIF_ETH], &ip_config, sizeof(ip_config));

  while (1) {
    timer_poll ();
    main_ ();
  }
}
```

In this example, the new parameters are defined in the code. However, in your application, this would be the network parameters read out from the EEPROM or NV RAM.

Note

- You can change the **localm[NETIF_ETH]** - ethernet IP parameters also when the system is working. See the HTTP_demo example. The ethernet controller **MAC address** can only be changed when the system starts. It is not advisable to change it later.

Library Files

There are two types of RL-TCPnet object libraries:

- **Production version** is a release version.
- **Debug version** is used for debugging. This version prints out debug messages to a serial port.

RL-TCPnet includes six library files:

- **TCP_ARM_L.LIB** release version for microcontrollers based on ARM7TDMI™ and ARM9™ - Little Endian.
- **TCP_CM1.LIB** release version for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Little Endian.
- **TCP_CM3.LIB** release version for microcontrollers based on Cortex™-M3 - Little Endian.
- **TCPD_ARM_L.LIB** debug version for microcontrollers based on ARM7TDMI™ and ARM9™ - Little Endian.
- **TCPD_CM1.LIB** debug version for microcontrollers based on Cortex™-M0 and Cortex™-M1 - Little Endian.
- **TCPD_CM3.LIB** debug version for microcontrollers based on Cortex™-M3 - Little Endian.

You must manually include the correct library into your project.

Note

- The debug version must not be used for the production release of your firmware because it significantly **reduces** the system performance.
- The existing libraries are built for the **Little Endian** format. Applications do not work if you select Big Endian for your project.

Using RL-TCPnet

RL-TCPnet is designed as a **stand alone** TCP/IP Operating System and does not require an RTOS kernel to run.

RL-TCPnet has integrated **applications** such as WEB Server, Telnet Server, and TFTP Server. Web pages are stored in a simple ROM file system integrated into the RL-TCPnet, so RL-FlashFS is not required. For an application where only the Web configuration is needed, the RTOS kernel and the RL-FlashFS can both be omitted, and thus the code size of the application can be reduced.

You can also use RL-TCPnet together **with the RTX kernel**. In this case, the basic framework is setup a little differently.

Several network interfaces are also supported. The most popular is the **ethernet** network interface. However, sometimes the ethernet is not available, and it is necessary to dial a remote Internet Service Provider to use the Internet.

RL-TCPnet has integrated support for the serial links with the implemented serial protocols **PPP** and **SLIP**. It also supports modem device drivers with integrated default drivers for **Standard Modems** and a direct serial **Cable Connection**.

Stand Alone

RL-TCPnet is designed as a **stand alone** TCP/IP Operating System. This means it does not need any external RTOS or File System to run. It has an integrated tiny **task scheduler** that manages timeouts, events, and internal tasks. Each application (TCP, UDP, and Network Interface) is treated as a separate internal task.

A **framework** for a stand alone application basically needs to call a few system functions:

- **init_TcpNet** function needs to be called after the system starts to **initialize** the OS.
- **init_user** is the function that initializes your application. It can be placed before or after the call to **init_TcpNet**. If you call your initialization code before calling **init_TcpNet**, you cannot call any OS function from your init function (because the OS is not yet initialized).
- **main_TcpNet** function needs to be called frequently to properly handle system events and timeouts. It is normally placed in an endless loop.
- **main_user** is the main thread of your application. You must design it as a **state machine**. On each call to this function, events and timeouts must be processed according the current state of the application, and then the function should return. It should never run in an endless loop waiting for a event as this **blocks** the whole OS, and the system might stop functioning.
- **timer_tick** function is needed to generate timer ticks for the OS. This function can be called from the function that **polls** for the timer timeout or from a timer **interrupt** function. It is used to synchronize internal timeout processing to a system tick timer. It is very important that this function be called at regular time intervals.

Here is a framework for a stand alone application:

```
#include <RTL.h>

static void timer_poll () {

  if (T1IR & 1) {
    T1IR = 1;
    /* Timer tick every 100 ms */
    timer_tick ();
  }
}

void main (void) {

  init_TcpNet ();
  init_user ();

  while (1) {
    /* Poll for a system tick timer overflow. */
    timer_poll ();
    main_TcpNet ();
    /* A function call to your application. */
    main_user ();
  }
}
```

Note

- **Avoid** using **waiting loops** in your application because the RTL-TCPnet system is **blocked** if any application function runs in a loop.

With RTX Kernel

Even though RL-TCPnet is designed as a **stand alone** TCP/IP Operating System, you can use it with the RTX kernel. This is useful for large complex applications, where different jobs are implemented as tasks. In this case, the basic framework is different because the functions from the stand alone framework are moved into tasks.

The following tasks are introduced:

- **init task** is used to initialize the system. Since the task is not needed after initialization is complete, the task can be terminated.
- **tcp_poll task** runs the main thread for RL-TCPnet. It must have the **lowest priority** in the system. Otherwise, the tasks with a lower priority than the tcp_poll task will never execute.
- **timer_tick task** generates periodic ticks for RL-TCPnet. For proper timing, the interval timer and the kernel timer reload value must be set in the **Net_Config.c** configuration file. The interval time should be the same as set in the configuration file.

  When the timings are not correct, the application might fail to complete certain functions. For example, requests may retransmit too fast, or the waiting timeout might expire before receiving a valid response from a remote server.

Here is a framework for the RTX kernel application:

```
#include <RTL.h>

__task void tick_timer (void) {
  os_itv_set (10);
  while (1) {
    os_itv_wait ();
    /* Timer tick every 100 ms */
    timer_tick ();
  }
}

__task void tcp_poll (void) {
  while (1) {
    main_TcpNet ();
    os_tsk_pass ();
  }
}

__task void init (void) {
  system_init ();
  init_TcpNet ();
  os_tsk_create (tick_timer, 2);
  os_tsk_create (tcp_poll, 1);
  /* Init done, terminate this task. */
  os_tsk_delete_self ();
}

void main (void) {
  os_sys_init (init);
}
```

Follow these guidelines when using RL-TCPnet with the RTX kernel:

- Functions are **not reentrant.** This means they must not be interrupted and called again from another task. All TCPnet-related functionality should be collected in a single networking task.

- All callback functions are executed from the networking task. Use the kernel **event**, **semaphore**, and **mailbox** functions to communicate with other tasks.

Applications

RL-TCPnet has several integrated high level applications, which can be enabled or disabled in the **Net_Config.c** configuration file.

The following applications are implemented:

- **HTTP Server** is a Web Server supporting dynamic Web pages and **CGI Scripting**
- **Telnet Server** with command line interface and authorization
- **TFTP Server** for uploading Web pages to a Web Server
- **SMTP Client** for sending automated emails
- **SNMP Agent** for managing the embedded system
- **DNS Resolver** used to resolve an IP address from the Host name.

Some applications require adding the user interface modules to your project and customizing them to your needs.

You can also write your own applications and use only the built-in **socket** interface. The RL-TCPnet system has integrated **TCP socket** and **UDP socket** interfaces.

HTTP Web Server

Hypertext Mark-up Language (HTML) is the primary language for **formatting web pages**. With HTML, you describe what a page must look like, what types of fonts to use, what color the text should be, where paragraph marks must come, and many more aspects of the document.

There are two types of web pages which are stored on a web server and sent to a web client on request:

- **Static Web pages** do not change their content. When the page is requested, it is sent to the web client as it is. It is not modified.
- **Dynamic Web pages** are generated when the page is requested. Pages that show system settings or log records are examples of dynamic pages.

RL-TCPnet supports both of them. Static web pages are generally stored in the virtual **ROM file system**. The files are converted into C-code by the **FCARM file converter** and compiled into code.

Supported Features and Technologies

The Embedded Web server has integrated several advanced features, which support the usage of many advanced web technologies:

- **Script language**
  is used to generate Dynamic Web pages.
- **Ajax programming**
  allows you to move the Web page processing from the Web server to the client browser.
- **SOAP interface**
  allows you to produce some **cutting edge** user interfaces.
- **HTTP Caching**
  supports the **local caching** at the browser and improves the Web Server performance a lot.
- **Web on SD Card**
  allows you to store the complete Web site resource files on SD Card. Web files can be updated with **HTTP file upload**.
- **Access filtering**
  allows you to filter out the hosts, which are **not allowed** to connect to the Web Server.
- **Multi-language Web pages**
  You can use language information to generate language specific web pages with the help of the integrated **script language**.

Note

- To use an Embedded Web Server, you must **enable** and **configure** it in the **configuration** file.

Script Language

The Embedded Web Server provides a small **script language** which can be used to generate true **dynamic** web pages. The HTTP Server processes the script source file line by line and calls the **CGI functions** as needed. The output from a CGI function is sent to the web client as a part of the web page.

Each script line starts with a **command character** which specifies a command for the script interpreter. The script language itself is simple and works as follows:

Command	Description
i	Commands the script interpreter to **include a file** from the virtual file system and to output the content on the web browser.
t	Commands that the **line of text** that follows is to be output to the browser.
c	Calls a C function from the **HTTP_CGI.c** file. The function may be followed by the line of text which is passed to **cgi_func()** as a pointer to an environment variable.
#	This is a comment line and is ignored by the interpreter.
.	Denotes the last script line.

Here is an example of a web page written in the script language. This web page edits or changes the system password. The web page is stored in three files (two are static, and the third (main file) is the script file that generates dynamic data).

- The script file **system.cgi** contains the following:

```

i password_h.inc
c d 1 <TR><TD><IMG SRC=pabb.gif>Authentication</TD><TD><b>%s</b></TD></TR>
t <TR><TD><IMG SRC=pabb.gif>Password for user 'admin'</TD>
c d 2 <TD><INPUT TYPE=TEXT NAME=pw SIZE=10 MAXLENGTH=10 VALUE="%s"></TD></TR>
t <TR><TD><IMG SRC=pabb.gif>Retype your password</TD>
c d 2 <TD><INPUT TYPE=TEXT NAME=pw2 SIZE=10 MAXLENGTH=10 VALUE="%s"></TD></TR>
i password_f.inc
.
```

- The web page header, which is **static** and does not change when the web page is generated, is moved into the separate header file **password_h.inc**. The content of the included page header file is:

```

<HTML>
  <HEAD>
    <TITLE>System Settings</TITLE>
  </HEAD>
  <BODY TEXT=#000000 BGCOLOR=#ccddff LINK=#0000FF VLINK=#0000FF ALINK=#FF0000>
  <H2 ALIGN=CENTER>System Settings</H2>
  <FORM ACTION=index.htm METHOD=POST NAME=CGI>
  <TABLE BORDER=0 WIDTH=99%>
  <TR BGCOLOR=#aaccff>
    <TH WIDTH=40%>Item</TH>
    <TH WIDTH=60%>Setting</TH>
  </TR>
```

  It is **included** into the generated web page with the following script command:

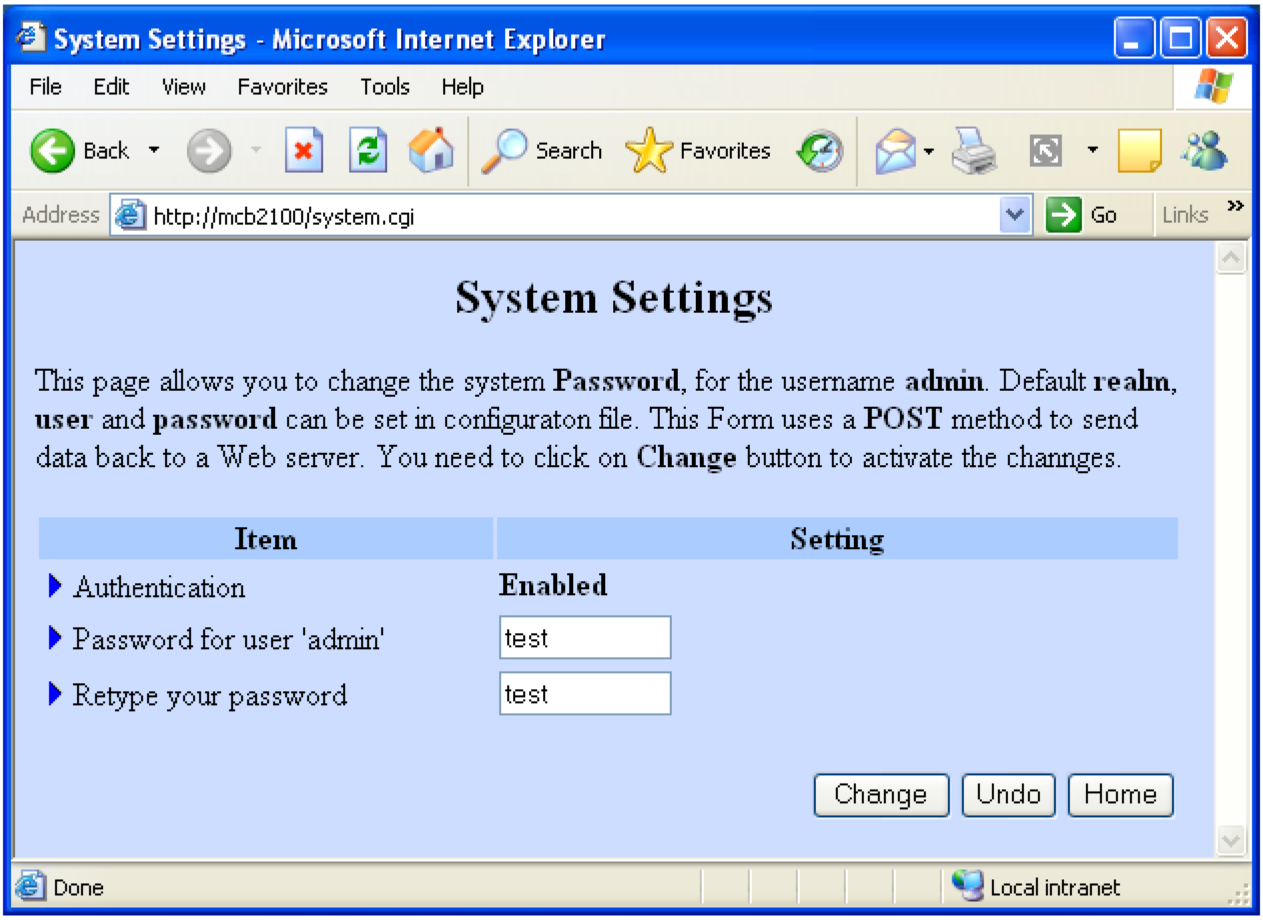
```

i password_h.inc
```

- The web page footer is also **static** and is moved into the **password_f.inc** file. It is not changed when the web page generates but is simply included in the script.

```

```

```
  <TR>
    <TD>&nbsp;</TD>
    <TD>&nbsp;</TD>
  </TR>
  <TR>
    <TD></TD>
    <TD align="right">
      <INPUT TYPE=SUBMIT NAME=set VALUE="Change" id="sbm">
      <INPUT TYPE=RESET VALUE="Undo">
      <INPUT TYPE=BUTTON VALUE="Home" OnClick="location='/index.htm'">
    </TD>
  </TR>
  </FORM>
  <p>This page allows you to change the system <b>Password</b>, for the username
    <b>admin</b>. Default <b>realm</b>, <b>user</b> and <b>password</b> can be set
    in configuration file. This Form uses a <b>POST</b> method to send data back to
    a Web server. You need to click on <b>Change</b> button to activate the changes.
  </p>
  </BODY>
</HTML>
```

This is how the generated web page looks like, with the dynamically generated items in the **Setting** column:



System Settings

This page allows you to change the system **Password**, for the username **admin**. Default **realm**, **user** and **password** can be set in configuraton file. This Form uses a **POST** method to send data back to a Web server. You need to click on **Change** button to activate the channges.

Item	Setting
▶ Authentication	**Enabled**
▶ Password for user 'admin'	test
▶ Retype your password	test

Change Undo Home

You can see the HTML source code of this web page. This data is actually sent to the web client when the client requests the web page **system.cgi**. The script file is processed by the script interpreter, and the following data is generated by the Web Server. You can compare the generated HTML source with the script file to see where the CGI interface comes in.

```
<HTML>
  <HEAD>
    <TITLE>System Settings</TITLE>
  </HEAD>
  <BODY TEXT=#000000 BGCOLOR=#ccddff LINK=#0000FF VLINK=#0000FF ALINK=#FF0000>
  <H2 ALIGN=CENTER>System Settings</H2>
  <FORM ACTION=index.htm METHOD=POST NAME=CGI>
  <TABLE BORDER=0 WIDTH=99%>
  <TR BGCOLOR=#aaccff>
    <TH WIDTH=40%>Item</TH>
    <TH WIDTH=60%>Setting</TH>
  </TR>
  <TR><TD><IMG SRC=pabb.gif>Authentication</TD><TD><b>Enabled</b></TD></TR>
  <TR><TD><IMG SRC=pabb.gif>Password for user 'admin'</TD>
  <TD><INPUT TYPE=TEXT NAME=pw SIZE=10 MAXLENGTH=10 VALUE="test"></TD></TR>
  <TR><TD><IMG SRC=pabb.gif>Retype your password</TD>
  <TD><INPUT TYPE=TEXT NAME=pw2 SIZE=10 MAXLENGTH=10 VALUE="test"></TD></TR>
  <TR>
    <TD>&nbsp;</TD>
    <TD>&nbsp;</TD>
  </TR>
  <TR>
    <TD></TD>
    <TD align="right">
      <INPUT TYPE=SUBMIT NAME=set VALUE="Change" id="sbm">
      <INPUT TYPE=RESET VALUE="Undo">
      <INPUT TYPE=BUTTON VALUE="Home" OnClick="location='/index.htm'">
    </TD>
  </TR>
  </FORM>
  <p>This page allows you to change the system <b>Password</b>, for the
username
    <b>admin</b>. Default <b>realm</b>, <b>user</b> and <b>password</b> can be
set
    in configuration file. This Form uses a <b>POST</b> method to send data
back to
    a Web server. You need to click on <b>Change</b> button to activate the
changes.
  </p>
  </BODY>
</HTML>
```

Note

- The script files use the reserved filename extension of **cgi**. Using the **cgi** filename extension for script files is **mandatory** for the Web Server script interpreter to recognize and process the script files.
- The script line length is **limited** to 120 characters.

CGI Functions

CGI functions are in the **HTTP_CGI.c** module. Copy this module to your project directory, add it to your project, and customize it. The HTTP_CGI.c module is in the **\Keil\ARM\RL\TCPnet\User** directory.

The following functions are included in this module:

- **cgi_process_var()** - processes the query string for the CGI Form **GET** method. The form in the web HTML source is created with the attribute **METHOD=GET**.
- **cgi_process_data()** - processes the returned Data for the CGI Form **POST** method. The form in the web HTML source is created with the attribute **METHOD=POST**.
- **cgi_func()** - processes a script line. It is called by the script interpreter.

The following functions are optional in this module:

- **http_accept_host()** - used for the web server access filtering. It is called when a web browser is trying to connect to a web server.
- **cgx_content_type()** - defines the HTML content type header for xml script files. This function might be used to override the default content type header for **cgx** scripts.

The following system functions are included in the RL-TCPnet library:

- **http_get_var()** - called from the **HTTP_CGI.c** module to process the **environment variables**.
- **http_get_lang()** - called from the **HTTP_CGI.c** module to retrieve the browser **preferred language** for Multi-Language Web pages.
- **http_get_info()** - called from the **HTTP_CGI.c** module to retrieve the remote machine information: **IP address** and **MAC address**.
- **http_get_session()** - called from the **HTTP_CGI.c** module to retrieve the current HTTP server running **session instance** index.
- **http_get_content_type()** - called from the **HTTP_CGI.c** module to retrieve the **content-type** html header received in XML-POST request.

Ajax Support

Ajax is a group of web development techniques used on the client-side to create interactive web applications. **Ajax** is a shorthand for *asynchronous JavaScript + XML*. With **Ajax** web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

Ajax is based on *JavaScript* and *HTTP requests*. It is not a new programming language, but a new way to use existing standards. *JavaScript* is the most popular language for **Ajax** programming due to its inclusion in and compatibility with the majority of modern web browsers.

The **XML**, a shorthand for *Extensible Markup Language*, is a simple, very flexible text format. It is a generic framework for storing any amount of any data whose structure can be represented as a **tree**. It allows the user to create the mark-up elements. XML has become the almost universally supported way of exchanging documents and data across applications and platforms.

Using XML

The TCPnet Web server supports the **Ajax** technology. The idea behind is to move the page processing from the Web Server to a local computer. **Java Script** processes the web page on a local computer and generates a web page updates.

The benefits of using XML technology for web **page update** are obvious:

- instead of several files, only **one small** XML file is transferred for web page update,
- this allows **faster**, **flicker-free** screen updates,
- the used LAN bandwith is very small,
- the web server can easily handle more clients at the same time.

The components of XML file are tagged. You must use this format for generated XML responses. The object ID's and their values must be specified within XML body - enclosed with tags **<form>** and **</form>**.

The following objects are defined:

- **Text** object

```
<text>
  <id>text_id</id>
  <value>text_value</value>
</text>
```

- **Checkbox** object

```
<checkbox>
  <id>checkbox_id</id>
  <checked>true/false</checked>
</checkbox>
```

- **Select** object

```
<select>
  <id>option_id</id>
  <value>true/false</value>
</select>
```

- **Radio** object

```
<radio>
  <id>radio_id</id>
  <value>true/false</value>
</radio>
```

The other http objects are optional. You can add them yourself.

XML Example

Here is an example of a XML web page update written in the script language. This web page displays the value of analog input (AD). When first opened, a complete **AD** web page is generated by the Embedded web server. Later, when you enable the periodic update of the page, this page is updated with Java Script function **updateMultiple()**. This function is in **xml_http.js** file.

- The script file **ad.cgx** contains the following:

```

t <form>
t <text>
t <id>ad_value</id>
c x<value>0x%03X</value>
t </text>
t </form>
.
```

- The script interpreter processess this file and generates the following response:

```

<form><text><id>ad_value</id><value>0x06A</value></text></form>
```

  The XML response does not contain any spaces or CR-LF characters between the XML tags.
- The Java Script function **periodicUpdateAd()** activates the periodic timeouts for the page update. This function is specific for AD page module and is included in **ad.cgi** script file:

```

function periodicUpdateAd() {
  if(document.getElementById("adChkBox").checked == true) {
    updateMultiple(formUpdate,plotADGraph);
    ad_elTime = setTimeout(periodicUpdateAd, formUpdate.period);
  }
  else
    clearTimeout(ad_elTime);
}
```

- The update **interval** and **url** are defined in the **formUpdate** object. When update interval expires, an xml response is requested from the url specified in this java object.

```

var formUpdate = new periodicObj("ad.cgx", 500);
```

- The actual update of the analog bar and AD values on the web page is done by the java function **plotADGraph()**:

```

function plotADGraph() {
  adVal = document.getElementById("ad_value").value;
  numVal = parseInt(adVal, 16);
  voltsVal = (3.3*numVal)/1024;
  tableSize = (numVal*100/1024);
  document.getElementById("ad_table").style.width = (tableSize + '%');
  document.getElementById("ad_volts").value = (voltsVal.toFixed(3) + '
V');
}
```

  This function is called as a call-back function from the **updateMultiple()** java function.

- The XML script files use the reserved filename extension **cgx**. Using the **cgx** file extension for XML script files is **mandatory** for the Web Server script interpreter to recognize and process the XML script files.
- There must be no **spaces** or **CR-LF** characters between XML tags, or some web browsers might have problems when parsing the XML response.
- The script line length is **limited** to 120 characters.

How it works

In **Ajax** programming, the browser sends standard HTTP request to an Embedded Web server, such as **GET** or **POST**. The Embedded Web server checks the file extension of the requested file. If the file extension is **cgx**, the requested file is an XML script file. This file is processed by internal script interpreter of the Web server. As a result the XML response is generated and sent back to the browser.

Here is a an example of a typical data flow.

- Web browser sends a standard http GET request:
-
- `GET /ad.cgx HTTP/1.1\r\n`
- Web server processess the script file **ad.cgx** and sends the **XML** response:
-
- `<form><text><id>ad_value</id><value>0x06A</value></text></form>`
- The **JavaScript** XML parser in the Web browser processes this response and updates the object with id **ad_value** on the web page.
-
- `<input type="text" readonly size="10" id="ad_value" value="0x06A">`

SOAP Support

The **SOAP** protocol, originally defined as *Simple Object Access Protocol*, is a simple **XML-based** protocol to let applications exchange information over HTTP. It is used by Microsoft **Silverlight** web service application.

SOAP is for communication between applications. It is important for application development to allow Internet communication between programs. A better way to communicate between applications is over HTTP, because HTTP is supported by all Internet browsers and servers. **SOAP** provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

With the Microsoft Silverlight and SOAP it is possible to produce some **cutting edge** user interfaces. The SOAP objects are used as the replies from the Embedded Web server to generate the classes at the Silverlight application.

SOAP Interface

The **SOAP** messages in HTTP consist of the **POST** request, submitted by the client, and a **response** generated by the Web server. The Embedded Web server handles the SOAP messages different. Instead of processing them internally and notifying the user via user callback functions, the Web server delivers a complete SOAP message to the user via the callback function.

The SOAP messages in general are large. Embedded systems that run the Web server with SOAP, need much more RAM for message buffering and processing. A typical configuration would have:

- a **few MBytes** of RAM
- an **SD Card** for deploying web service application, for example the Silverlight.

The following extensions have been added to the Web server:

- the **cgi_process_data** function has been extended with the code 4 and 5 to allow processing fragmented large POST messages.
- the *Content-Type* http header for XML-encoded POST requests is **buffered**. Function **http_get_content_type** returns a pointer to the *Content-Type* string, which was received in the XML POST request.
- the *Content-Type* header for the response can be defined by the user in the **cgx_content_type** function.
- the **http_get_session** function is used to identify which Web server session has called a **cgi_process_data** callback function, if two or more clients have sent XML-POST requests at the same time.
- the **HTTP Caching** improves the Web server performance a lot when serving large web service applications.

Large POST Messages

When the Web server receives the POST request, the server checks the *Content-Type* header. For all XML-encoded content types, the server does no further processing, but delivers the data to the user in the callback function. It is a responsibility of the user to correctly **assemble** large POST messages, because they are fragmented and delivered from several TCP packets. For the first and optional subsequent packets, the Web server calls the callback function with code 4:

```
cgi_process_data (4, dat, len);
```

The user shall now start buffering the data into a buffer.

For the **last** data packet, the Web server calls the callback function with code 5:

```
cgi_process_data (5, dat, len);
```

The XML-POST data is now complete and the user can start parsing the XML encoded data. In addition the user might check what was the *Content-Type* with the function **http_get_content_type**
.

Web Pages

The web pages for the Embedded Web Server are created in the same way as the web pages for any other web server. You can use any text editor to edit the HTML code. It is a good idea to **preview** the page. Here are a few guidelines:

- Design a web page in HTML code. You can add **images** and **java script** functions as well.
- If the web page needs to show **dynamic values**, code this page in the **script language**. You must also update the CGI callback function, **cgi_func()**, accordingly. It is a good practice to program the script code and cgi_func() in parallel. Note that the module **HTTP_CGI.c** must be included in your project.
- If needed, draw or add **images**, which can be of type gif, bmp, jpg, or png for example. A good choice is to select the image type with the **better compression** because these files are included in the code. Large image files might increase the application code size by a large extent.
- Add all the web files to the **FCARM** File Converter **command file** and include the **FCARM** output C-file in your project.

Default Page

When you type the url in the browser's address bar, for example:

```
http://mcb2300
```

the Web server sends the content of **index.htm** web page. This is the default page, which is opened, when no filename is specified in the **url**. If you enter the **complete url** with a filename, for example:

```
http://mcb2300/ad.htm
```

then the Web server tries to open this page.

The default page **index.htm** is a static page. This means the content of this page is stored on Web server and sent unmodified to the web client on request. Usually this page contains links to other static or dynamic pages on the Web server.

Sometimes a dynamic page is preferred as a default web page. Embedded Web server implicitly supports also this option. When a web browser requests a default web page, the web server tries to open **index.htm** as default web page. If this page does not exist, Web server in the next step tries to open **index.cgi** as a default page. If this page is also not existing, then the Web server responds with **HTTP Error 404** - Requested File Not Found.

- If you want to use a dynamic default page, then the file **index.htm** must not exist on the Web server.

Error Pages

HTTP Server shows an Error page when it encounters error conditions such as:

- **HTTP Error 401** - Unauthorized access to Web Server
- **HTTP Error 404** - Requested File Not Found
- **HTTP Error 501** - Requested Method Not Implemented

Default HTTP Server **Error pages** are already included into the RL-TCPnet library. If you want to modify them, you must copy the module **HTTP_err.c** to your project and customize it. This module is located in the **\Keil\ARM\RL\TCPnet\User** folder. Modified error pages must be small because they are sent in a single TCP packet.

Web on SD Card

You can place the content of **web pages** also on SD Card, which is attached to the embedded system. This configuration is useful if you want to update or change the web content remotely. In this case, the **RL-FlashFS** system is used, which must be configured for a target device **Memory Card Drive**.

You can use any of the update methods to **change** the content of Embedded web pages:

- **HTTP File Upload**
  The Web files are uploaded using a web browser. HTTP Server must be configured for the file upload. (See *HTTP_Upload* example for details).
- **FTP Server**
  The Web files are uploaded and managed with the FTP Client, such as Windows Explorer. This might be easy, but the FTP Server must be correctly configured.
- **TFTP Server**
  The TFTP Server allows only a limited functionality of the file manipulation. Again, the TFTP Server must be enabled for the embedded system and correctly configured.

Note

- To enable the SD-Card File System, you must copy the **HTTP_uif.c** user interface module to your project directory and add it to your project. This interface module is in the **\Keil\ARM\RL\TCPnet\User** folder. It is preconfigured for the **RL-FlashFS** system, so no modifications to this file are required.
- The RL-FlashFS library code is not included in the project by default to reduce the application code size. You must include the file **HTTP_uif.c**, which contains the interface functions for the Flash File System and SD-Card.

Web Update

When Web server tries to open a web page, it searches the **external file system** first. This is usually an externally attached SD Card. If the Web server is not configured for external file system, then only the internal virtual ROM file system, which is compiled into the code, is searched.

If the requested file is found on SD Card, then the content of this file is sent to web client. If the requested file does not exist on SD Card, then the file with the same name is opened on internal ROM file system and transferred to the web client. This concept allows you to simply **replace** the web content from internal ROM file system with a new one, that is uploaded to external SD Card.

You should carefully update the **cgi** and **cgx** script files, as the change in the script files usually reflects in a change to the application code, which is in module **HTTP_CGI.c**. If you have made a mistake in the uploaded script files, the web pages might not be accessible anymore.

File System Interface

The Embedded Web Server can read files also from a generic File System. All interface functions are located in **HTTP_uif.c**, which is a user interface module. This module is in the **\Keil\ARM\RL\TCPnet\User** folder. For large Web resources, you must copy it to your project directory and add it to your project.

The HTTP_uif.c module is preconfigured for **RL-FlashFS**, so no modifications are required. You can modify this interface module to use another type of file system or to use a different storage media such as a hard disk.

The following functions are implemented in this module:

- **http_fopen()** - open a file for reading.
- **http_fclose()** - close a file that was previously opened.
- **http_fread()** - read a block of data from a file to data buffer.
- **http_fgets()** - read a string from a file to data buffer.
- **http_finfo()** - read last modification time of a file.

The following system functions are included in the RL-TCPnet library:

- **http_date()** - convert the RL date format to **UTC** time format.

- When a complete Web site is stored internally in the code, you **should not** include the **HTTP_uif.c** module into the project to reduce the code size of the image.

Http Caching

HTTP protocol supports **local caching** of static resources by the browser. Most web pages include resources that change infrequently, such as CSS files, image files, JavaScript files, and so on. These resources take time to download over the network, which increases the time it takes to load a web page. HTTP caching allows these resources to be saved, or cached, by a browser. Once a resource is cached, a browser can refer to the locally cached copy instead of having to download it again on subsequent visits to the web page.

The advantage of caching is obvious:

- the page **load time** for subsequent user visits is reduced, eliminating numerous HTTP requests for the required resources
- the total **payload size** of the responses is reduced.

How it works

The Embedded Web server supports the HTTP **local caching** by the browser. For static resources, what are basically all except the scripts, the server sends the http header with the **last modified** date tag in the response.

Resource not cached

Here is an example, recorded from **HTTP_demo**. The browser opens a web page, which is not yet cached locally.

- The browser opens a default page:

```

http://mcb2300
```

- This generates the following http request for the web server:

```

GET / HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-
shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint,
application/msword, application/xaml+xml,
application/vnd.ms-xpsdocument,
application/x-ms-xbap, application/x-ms-application, */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
Trident/4.0; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727; .NET CLR
3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.0.4506.2152; .NET CLR
3.5.30729)
Accept-Encoding: gzip, deflate
Host: mcb2300
Connection: Keep-Alive
```

- The web server opens a default page *index.htm* and sends it to the browser with the following http header:

```

HTTP/1.1 200 OK
Server: Keil-EWEB/2.1
Content-Type: text/html
Last-Modified: Thu, 19 Nov 2009 07:46:25 GMT
Connection: close
// and the content of 'index.htm'
```

- The browser, when receiving the *Last-Modifed* http header, stores this file to local cache together with the url and date tag.

Resource cached

When the browser tries to open the same page again, it first checks the local cache. The file *index.htm* is already there, so it sends a different request to the Web server.

- The browser sends a different http header in the request:

```

GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
Trident/4.0; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727; .NET CLR
3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.0.4506.2152; .NET CLR
3.5.30729)
```

```
Accept-Encoding: gzip, deflate
If-Modified-Since: Thu, 19 Nov 2009 07:46:25 GMT
Host: mcb2300
Connection: Keep-Alive
```

- Now, the Web server is also informed that the requested file is cached. The Web server checks the date, if the browser caches the same file version. If the date tags are equal, the Web server sends only a short response:

```

HTTP/1.1 304 Not Modified
Server: Keil-EWEB/2.1
Connection: close
```

- The browser now uses the locally cached *index.htm*

Outdated resource cached

If the date tags are not equal, the file cached by browser is outdated. The Web server sends the updated *index.htm*. The browser updates the **local cache** and uses an updated file.

```

HTTP/1.1 200 OK
Server: Keil-EWEB/2.1
Content-Type: text/html
Last-Modified: Thu, 19 Nov 2009 09:38:54 GMT
Connection: close
// and the new content of 'index.htm'
```

Internal Web

The Internal Web pages are included and compiled into the code. When the **FCARM** file converter reformats the web files into a C-file, adds also the time, when FCARM was executed, into a C-file.

```
const U32 FileMD = 1256735886;
```

This time is used later by the Web server as the File Modification Date. It is specified in **UTF** format. The Web server uses this date in the http responses.

File caching improves the Web server performance a lot. The following table lists the times required to load the default page from **HTTP_demo**.

Web not cached	Web cached
447.5 ms	53.1 ms

- Default Web includes four files: *index.htm*, *keil.gif*, *nxp_logo.gif* and *llblue.jpg*, with the total size of 12 kBytes.

External Web

The Web server also supports the browser **local caching** of Web pages stored on **SD Card** at the Web server. In general the files on SD Card are bigger, and so the performance gain, much better. The space, available for Internal Web pages is limited with the size of internal flash memory. So all large images, java script archives and other large web resources, have to be located on externally attached SD Card.

Static Web Resources

The **static web** resource files are copied to SD Card when the application is built and are not modified later. You might use the SD Card Reader attached to PC to copy the files. In this case, the file modification date is set correctly by the PC. If you use an embedded application to copy the files, the file modification date is most likely set to the FlashFS **default time**. This however does not create any problems in browser local caching. Once the web is locally cached by the browser, the cache is always valid and is used in subsequent browser requests.

Using Web Update

Web resource files, which are updated later with one of the **update options** provided by RL-TCPnet, are **dynamic Web** resource files. You must provide the file modification **date** and **time** to the Flash File System. If this information is not available, the FlashFS uses a default file modification time. This might create troubles in **local caching** by the browser.

If you **upload** an updated web page to the server, but the FlashFS adds a default file modification date, the Web server is not able to recognize the updated files. It always reports the same *Last-Modified* date. The the browser then uses locally cached, but outdated web resources instead of the updated ones.

To load the updated web page in the browser, you have to manually **clear the cache** in the browser first and then reload the web page again.

- If the FlashFS does not have a Real Time Clock information, the Web Server will not work for updated Web pages.

Using RAM File System

You can add a **real dynamic image** feature to the Embedded Web Server. In this case, the **RL-FlashFS** system is used, which must be configured for a target device **SRAM**.

You can, for example, use a routine which generates a **png image** file which represents the graph of the temperature over the last 12 hours and store it to the RAM File System. This file is then sent to the web browser on request. You can create a more professional appearance on the web page using this feature. The measurement results for the above example are displayed in a **real graphic** format.

To enable the RAM File System, you must copy the **HTTP_uif.c** user interface module to your project directory and add it to your project. This interface module is in the **\Keil\ARM\RL\TCPnet\User** folder. It is preconfigured for the **RL-FlashFS** system, so no modifications to this file are required.

Note

- The RL-FlashFS library code is not included in the project by default to reduce the application code size. You must include the file **HTTP_uif.c** only when real dynamic images are required.

FCARM File Converter

This file converter reformats all the web files into a single C-file, which is then included and compiled into the project. All the files are stored in the **Virtual ROM File System**.

The FCARM file converter has integrated also a **file optimization** algorithm to compress the html, java script and cgi script files. This allows you to create more compact and smaller executable images.

The FCARM command line syntax is:

```
FCARM <{>inputlist<}> <{> TO outputfile<}> <{>directives<}>
```

or

```
FCARM @commandfile
```

Where

`inputlist` is a comma-separated list of web files for the file converter to include in the output file.

The *inputlist* uses the following general format:

```
filename <{> , ... <}>
```

Where

`filename` is the name of an html, script or image file. The filename must be specified with a file extension, but without the path information.

`outputfile` is the name of the output C-file containing converted web files.

`directives` are commands and parameters that control the operation of the file converter.

`commandfile` is the name of a command file that can contain an *inputlist*, *outputfile*, and *directives*.

File Converter Directives

The following table describes all FCARM Converter directives.

Directive	Description
PRINT	Specifies the name of the listing file.
NOPRINT	Disables generation of the listing (LST) file.
PAGEWIDTH	Specifies the number of characters on a line in the listing file.
PAGELENGTH	Specifies the number of lines on a page in the listing file.
ROOT	Specifies the root path where web files are located relative to the project directory path.

Disabling Optimization

To disable the FCARM optimization for a web content file, you have to add the **tilde prefix** (**~**) to the file name. For example, to disable optimization for *jquerry.js*, specify this file in a list as *~jquerry.js*.

This allows you to selectively disable optimization for a file of interest. For example if you want to debug java script file, it is very hard to trace the code that is compacted to a single line with no spaces, no comments and no line feed separators.

Examples

The following command line example invokes FCARM, specifies the web file **index.htm**. The output C-file is **index.c** which the file converter creates.

```
FCARM index.htm
```

The following command line example invokes FCARM, specifies the web files **index.htm**, **keil.gif**, **llblue.jpg**, **system.cgi**, and specifies the C-file **web.c** which the file converter creates.

```
FCARM index.htm, keil.gif, llblue.jpg, system.cgi to web.c
```

The following command line example invokes FCARM, specifies the web files **index.htm**, **keil.gif**, **llblue.jpg**, **system.cgi**, specifies the C-file **web.c**, and disables generation of the listing file. The web files are located in subfolder **Web_Files**.

```
FCARM index.htm, keil.gif, llblue.jpg, system.cgi to web.c nopr
root(Web_Files)
```

PRINT Directive

Abbreviation

PR

Arguments

PRINT (*filename*)

Default

The name of the generated listing file with a **.LST** extension.

µVision

None.

Description

The **PRINT** directive specifies the name of the listing file. If no **PRINT** directive is specified the listing file is given the name of the generated c-source file with a **.LST** extension.

See Also

NOPRINT

Example

```
FCARM index.htm, keil.gif to web.c print (Sample.lst)
```

NOPRINT Directive

Abbreviation

NOPR

Arguments

NOPRINT

Default

The listing file is generated using the basename of the output file.

µVision

None.

Description

The **NOPRINT** directive prevents the file converter from generating a listing file.

See Also

PRINT

Example

```
FCARM index.htm, keil.gif to web.c print(web.lst)
```

PAGEWIDTH Directive

Abbreviation **PW**

Arguments **PAGEWIDTH** (*number*)

Default **PAGEWIDTH (132)**

µVision None.

Description The **PAGEWIDTH** directive specifies the number of character per line that may be printed to the converter listing file. Lines with more than the specified number of characters are broken into two or more lines. The valid range of values is 72-132 columns.

See Also **PAGELENGTH**

Example

```
FCARM index.htm to web.c pagewidth (132)
```

PAGELENGTH Directive

Abbreviation **PL**

Arguments **PAGELENGTH** (*number*)

Default **PAGELENGTH (60)**

µVision None.

Description The **PAGELENGTH** directive specifies the number of lines printed per page in the converter listing file. The minimum page length is 10 lines per page. The maximum page length is 65535.

See Also **PAGEWIDTH**

Example

```
FCARM index.htm to web.c pagelength (55)
```

ROOT Directive

Abbreviation

RO

Arguments

ROOT (*directory*)

Default

The current project directory path is used as the web container.

µVision

None.

Description

The **ROOT** directive defines the root path where the web files are located relative to the project directory path.

See Also

None.

Example

```
FCARM index.htm, keil.gif to web.c root(Web_Files)
```

Telnet Server

Telnet is most often used for **remote login**. A user typically uses a Telnet client program to open a Telnet connection to a remote server. The server then treats the Telnet client like a local terminal and allows the user to log in and access the server's resources as if the user was using a directly-attached terminal. Telnet is still used this way quite extensively by UNIX users, who often need to log in to remote hosts from their local machines.

It is the client and server devices that decide whether Telnet is used for remote access or for some other purpose. When Telnet is used to access a remote device, the protocol itself is used to:

- set up the connection between the client and server machines
- encode data to be transmitted according to the rules of the Telnet Network Virtual Terminal (NVT)
- facilitate the negotiation and use of options.

Using the RL-TCPnet Embedded Telnet Server, you can build a simple **command line** interface that enables a Telnet client to access and control the remote embedded system.

- In order to use the Embedded Telnet Server, you have to **enable** and **configure** it in the **configuration** file.

Command Line Interface

The Embedded Telnet Server calls the **tnet_process_cmd()** function when a command is received from the user. A command is any sequence of characters that is terminated by the **CRLF** sequence (the **Enter** key is pressed). The Telnet Server assembles this command and passes it as an argument to the **tnet_process_cmd()** function.

The command line interface functions are located in the **Telnet_uif.c** module. You must copy this module to your project directory, add it to your project, and customize it. You can add new commands or remove existing commands from the module. The module is located in the **\Keil\ARM\RL\TCPnet\User** folder.

The following functions are implemented in this module:

- **tnet_cbfunc()** - copies various system messages to the sending buffer for transmission to the Telnet Client. You can also modify this function to support different languages.
- **tnet_msg_poll()** - polls the upper-layer application for Unsolicited messages.
- **tnet_process_cmd()** - called by the Telnet Server to process the commands entered by a remote user.

The following system functions are included in the RL-TCPnet library:

- **tnet_ccmp()** - compares the content of the command buffer.
- **tnet_set_delay()** - sets a delay before a repeated call to **tnet_proc_cmd()**. You can use this function to implement a continuously updating screen.
- **tnet_get_info()** - called from **Telnet_uif.c** to obtain the **IP address** and **MAC address** of the remote machine.

Sending Reply Message

The **tnet_process_cmd** function processes the Telnet command when it is received from a remote user. This function then generates a reply message and sends it back to the user. This is how the **command-line** interface works.

When the reply message is short, the whole message can be sent in a **single packet**. However, when long reports are generated, **multiple packets** must be sent to transfer the whole message. This is the case, for example, when the log files are displayed. Both single and multiple packets are supported by the Embedded Telnet Server.

Short Reply

A short reply message is sent in a **single** TCP packet. When a telnet command is received, a reply message is generated and sent to the remote telnet client.

In the following example, the telnet command **HELP** is sent by the Telnet client:

```
MCB2100> HELP
```

This command is answered by the predefined help message **tnet_help**. This message is copied to the output buffer and sent to the remote telnet client. The following code sends the reply message.

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pcgi) {
   ..
  if (tnet_ccmp (cmd, "HELP") == __TRUE || tnet_ccmp (cmd, "?") == __TRUE) {
    /* 'HELP' command, display help text */
    len = str_copy (buf,tnet_help);
    return (len);
  }
   ..
}
```

Long Reply

A long reply message requires multiple calls to the function **tnet_process_cmd**. Each call to this function generates part of the reply message until the entire message is generated and sent. To distinguish between different calls to the function, the argument **pvar** is used. This argument is a pointer to a variable that is set to 0 on the first call and not altered on each subsequent call to this function. The function's return value, which specifies the number of bytes in the reply message, cannot exceed 1500. Hence the high bits of the function's return value is used to store the flags:

- **Repeat flag** - bit 14
  This flag tells the Telnet Server whether the function **tnet_process_cmd** must be called again or not (because the command processing is complete). The return value must be OR-ed with 0x4000 to call the function again.
- **Disconnect flag** - bit 15
  This flag tells the Telnet Server to disconnect the telnet connection. If this flag is set, the Telnet Server disconnects the current Telnet session. The return value must be OR-ed with 0x8000 to disconnect.

In the following example, the **MEAS** command is given by the user using the Telnet client.

```
MCB2100> MEAS 100
```

When a new telnet command is received, the function **tnet_process_cmd** is called with the argument ***pvar** set to 0. The command buffer **cmd** is checked to identify the command.

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {

  switch (MYBUF(pvar)->id) {
    case 0:
      /* First call to this function, the value of '*pvar' is 0 */
      break;

    case 1:
      /* Repeated call, command 'MEAS' measurements display. */
        ..
      /* Request a repeated call, bit 14 is a repeat flag. */
      return (len | 0x4000);
        ..
  }

  /* Check if the command 'MEAS' is entered. */
  if (tnet_ccmp (cmd, "MEAS") == __TRUE) {
    MYBUF(pvar)->id = 1;
    if (len > 5) {
      /* We must be careful here, because data is overlaid. */
      sscanf ((const S8 *)&cmd[5], "%d", &temp);
      MYBUF(pvar)->nmax = temp;
    }
    len = str_copy (buf,(U8 *)meas_header);
    if (MYBUF(pvar)->nmax) {
      /* Bit 14 is a repeat flag. */
      len |= 0x4000;
    }
    return (len);
  }
```

When a command is recognized, you can reuse the same command buffer to store local variables, which might be needed in repeated calls. During the repeated call to this function, the **cmd** buffer is locked and is not altered by the system. You can use it as temporary storage of variables for the repeated calls. Each Telnet session has its own buffer of size 96 bytes. You can use only 95 bytes since the last byte is not available.

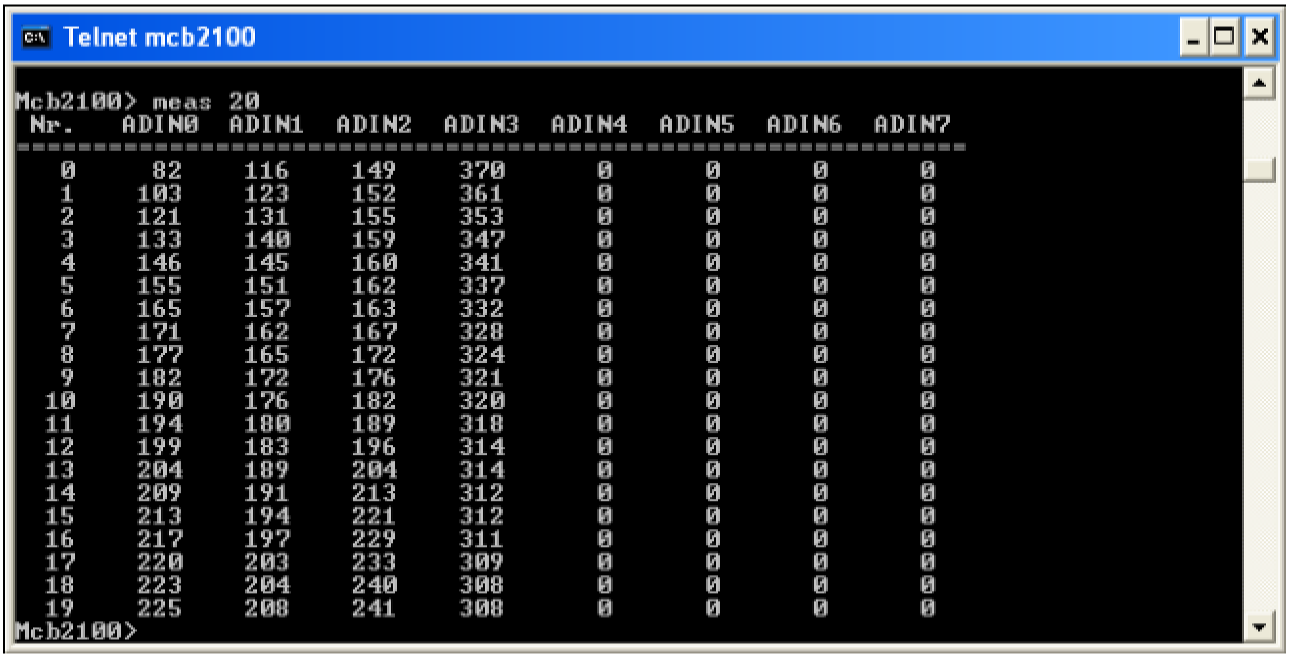
The above example uses 3 bytes of a storage variable pointed by **pvar** pointer for the following structure:

```
typedef struct {
  U8 id;
  U8 nmax;
  U8 idx;
} MY_BUF;
#define MYBUF(p)         ((MY_BUF *)p)
```

When the call to **tnet_process_cmd()** is repeated for the same command, the value of a storage variable pointed to by argument **pvar** is not altered anymore. You can use the value of *pvar to process the command differently. The ***pvar** buffer now holds the private structure **MY_BUF**, which is valid for the lifetime of processing the command. When the command processing is finished, this buffer is not used anymore until the next command.

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {

  switch (MYBUF(pvar)->id) {
    case 0:
      /* First call to this function, the value of '*pvar' is 0 */
      break;

    case 1:
      /* Repeated call, command 'MEAS' measurements display. */
      while (len < buflen-80) {
        /* Let's use as much of the buffer as possible. */
        /* This will produce less packets and speedup the transfer. */
        len += sprintf ((S8 *)(buf+len), "\r\n%4d", MYBUF(pvar)->idx);
        for (val = 0; val < 8; val++) {
          len += sprintf ((S8 *)(buf+len), "%7d", AD_in(val));
        }
        if (++MYBUF(pvar)->idx >= MYBUF(pvar)->nmax) {
          /* OK, we are done. */
          return (len);
        }
      }
      /* Request a repeated call, bit 14 is a repeat flag. */
      return (len | 0x4000);

    case 2:
      /* Repeated call, TCP status display. */
     ..
}
```

After giving a **MEAS** command, the Telnet Client screen looks like this:



```
Mcb2100> meas 20
 Nr.    ADIN0   ADIN1   ADIN2   ADIN3   ADIN4   ADIN5   ADIN6   ADIN7
=====================================================================
   0      82     116     149     370       0       0       0       0
   1     103     123     152     361       0       0       0       0
   2     121     131     155     353       0       0       0       0
   3     133     140     159     347       0       0       0       0
   4     146     145     160     341       0       0       0       0
   5     155     151     162     337       0       0       0       0
   6     165     157     163     332       0       0       0       0
   7     171     162     167     328       0       0       0       0
   8     177     165     172     324       0       0       0       0
   9     182     172     176     321       0       0       0       0
  10     190     176     182     320       0       0       0       0
  11     194     180     189     318       0       0       0       0
  12     199     183     196     314       0       0       0       0
  13     204     189     204     314       0       0       0       0
  14     209     191     213     312       0       0       0       0
  15     213     194     221     312       0       0       0       0
  16     217     197     229     311       0       0       0       0
  17     220     203     233     309       0       0       0       0
  18     223     204     240     308       0       0       0       0
  19     225     208     241     308       0       0       0       0
Mcb2100>
```

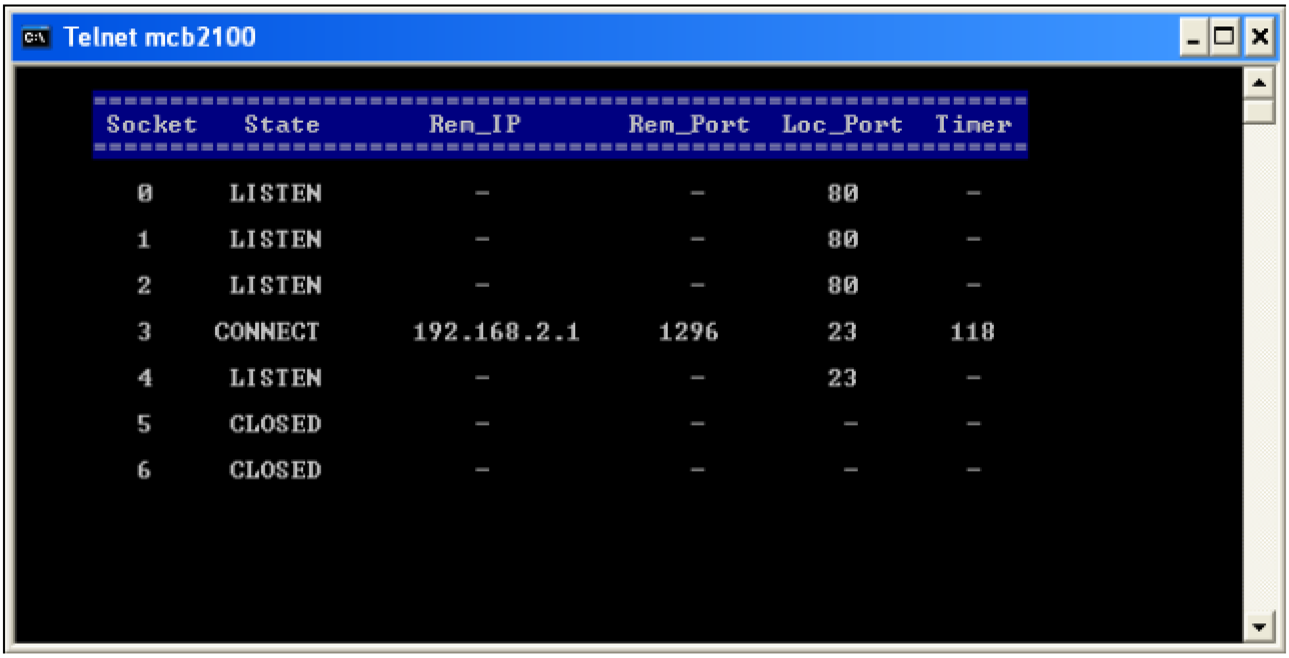
Note

- You can check the **Telnet_demo** example to see how the Telnet Server works. This example is located in the **\Keil\ARM\Boards\Phytec\LPC229x\RL\TCPnet** folder.

Continuous Screen Update

Continuous screen updates can be used for continuous **monitoring** of measurement or status variables. When enabled, the telnet client screen refreshes periodically. The **tnet_set_delay** function activates a delayed repeating call to the **tnet_process_cmd** function.

The screen for status monitoring appears as follows:



```
==============================================================
 Socket   State         Rem_IP        Rem_Port  Loc_Port  Timer
==============================================================

    0     LISTEN          -              -         80        -
    1     LISTEN          -              -         80        -
    2     LISTEN          -              -         80        -
    3    CONNECT     192.168.2.1       1296        23       118
    4     LISTEN          -              -         23        -
    5     CLOSED          -              -          -        -
    6     CLOSED          -              -          -        -
```

The refresh interval, after which the screen regenerates and reflects the most recent status, is 2 seconds.

Note

- You can check the **Telnet_demo** example to see how the Telnet Server works. This example is in the **\Keil\ARM\Boards\Phytec\LPC229x\RL\TCPnet** folder.

TFTP Server

Trivial File Transfer Protocol (TFTP) is a simple protocol for **exchanging files** between two TCP/IP machines. TFTP servers allow connections from TFTP clients to perform file send and receive operations. TFTP users initiate connections by starting a TFTP client program, which generally uses a command-line interface.

The TFTP protocol supports only file send and receive operations. File delete, file move, and file rename are not supported. Due to its limitations, TFTP is a complement to the regular File Transfer Protocol (FTP) and not a replacement. It is used only when its simplicity is important, and its lack of features is acceptable. The most common application is bootstrapping, although it can be used for other purposes.

Embedded TFTP Server can be used to **upload** HTTP Web pages or to **download** log files to a remote PC. In this case, the **Flash File System** must be used, and the Embedded **Web Server** must be properly configured.

Note

- In order to use an Embedded TFTP Server, you have to **enable** and **configure** it in the **configuration** file.

File System Interface

The Embedded TFTP Server can store files in a generic File System. All interface functions are located in **TFTP_uif.c**, which is a user interface module. This module is in the **\Keil\ARM\RL\TCPnet\User** folder. You must copy it to your project directory and add it to your project.

The TFTP_uif.c module is preconfigured for **RL-FlashFS**, so no modifications are required. You can modify this interface module to use another type of file system or to use a different storage media such as a hard disk.

The following functions are implemented in this module:

- **tftp_fopen()** - open a file for reading or writing.
- **tftp_fclose()** - close a file that was previously opened.
- **tftp_fread()** - read a block of data from a file to the TFTP data buffer.
- **tftp_fwrite()** - write a block of data from the TFTP data buffer to a file.

FTP Server

File Transfer Protocol (FTP) is a standard network protocol used to **exchange** and **manipulate files** over a TCP/IP-based network. FTP is built on a client-server architecture and utilizes separate control and data connections between the client and server applications. FTP is used with user-based password authentication or with anonymous user access.

FTP file manipulation means that you can: **create** and **delete** files on FTP server, **rename** files, create folders and subfolders, print the folder listings, etc.

FTP applications were originally interactive command-line tools with a standardized command syntax. Various graphical user interfaces have been developed for all types of operating systems in use today.

FTP can be run in **active** or in **passive mode**, which control how the second data connection is opened.

- In **active mode** the client sends the server the IP address port number that the client will use for the data connection, and the **server opens** the connection.
- In **passive mode** the server sends the client an IP address and port number and the **client opens** the connection to the server. This mode is used, when the client is located behind a firewall and unable to accept incoming TCP connection.

Embedded FTP Server can also be used to **upload** HTTP Web pages or to **download** log files to a remote PC. In this case, the **Flash File System** must be used, and the Embedded **Web Server** must be properly configured.

Note

- In order to use an Embedded FTP Server, you have to **enable** and **configure** it in the **configuration** file.

File System Interface

The Embedded FTP Server can store files in a generic File System. All interface functions are located in **FTP_uif.c**, which is a user interface module. This module is in the **\Keil\ARM\RL\TCPnet\User** folder. You must copy it to your project directory and add it to your project.

The FTP_uif.c module is preconfigured for **RL-FlashFS**, so no modifications are required. You can modify this interface module to use another type of file system or to use a different storage media such as a hard disk.

The following functions are implemented in this module:

- **ftp_fopen()** - open a file for reading or writing.
- **ftp_fclose()** - close a file that was previously opened.
- **ftp_fread()** - read a block of data from a file to the FTP data buffer.
- **ftp_fwrite()** - write a block of data from the FTP data buffer to a file.
- **ftp_fdelete()** - delete a specified file.
- **ftp_frename()** - rename a file from old to a new name.
- **ftp_ffind()** - find a file in a folder for printing a directory listing.

Supported Commands

The Embedded FTP Server supports only a subset of standard FTP Commands. The following FTP Commands are supported:

Code	Command	Description
USER	User Name	Starts login with name identifying the user.
PASS	Password	Continues login with the user's password.
QUIT	Logout	Closes the user connection.
SYST	System	Identifies the operating system at the server.
NOOP	No Operation	Sends an OK reply.
PWD	Print Working Directory	Returns the name of the current working directory.
CWD	Change Working Directory	Changes the current working directory of the user.
MKD	Make Directory	Creates a sub directory in the current working directory.
RMD	Remove Directory	Removes the directory.
TYPE	Representation Type	Supports **A**SCII and **I**mage types.
PORT	Data Port	Specifies the data port to be used in data connection.
PASV	Passive	Requests the server to listen on a data port and wait for a connection.
LIST	List	Sends a directory listing to the user.
NLST	Name List	Sends a directory listing to the user.
RETR	Retrieve	Sends a file content to the user.
STOR	Store	Saves a captured user file on server.
DELE	Delete	Deletes a specified file from server.
RNFR	Rename From	Specifies the name of existing file to rename. (must be followed by RNTO).
RNTO	Rename To	Renames an existing file to new name.
HELP	Help	Returns a list of supported commands.
SIZE	Size	Returns the size of a specified file.
MDTM	Last-modified Time	Returns last-modified time of a specified file.

SMTP Client

Simple Mail Transfer Protocol (SMTP) is a widely used protocol for the delivery of **e-mails** between TCP/IP systems and users. All steps in the e-mail system use SMTP with the exception of the final retrieval step by an e-mail recipient.

An Embedded SMTP Client can send e-mails to various recipients. A typical use is to send **automated** e-mail notifications to different e-mail addresses.

E-mail content can be a static **predefined e-mail** message or a **real dynamic** message. An example of a real dynamic email is one that contains the measurement results from a log file or an email with current measurement values.

SMTP Client interface functions are in the user interface module **SMTP_uif.c**, which is in the **\Keil\ARM\RL\TCPnet\User** folder. You must copy it to your project directory and add it to your project. Customize this module by changing the From address, the To address, and the body of the email message.

The following function is in the module:

- **smtp_cbfunc()** - callback function to compose the e-mail.
- **smtp_accept_auth()** - callback function to accept/deny the authentication advertised by SMTP Server.

The following function is in the RL-TCPnet library:

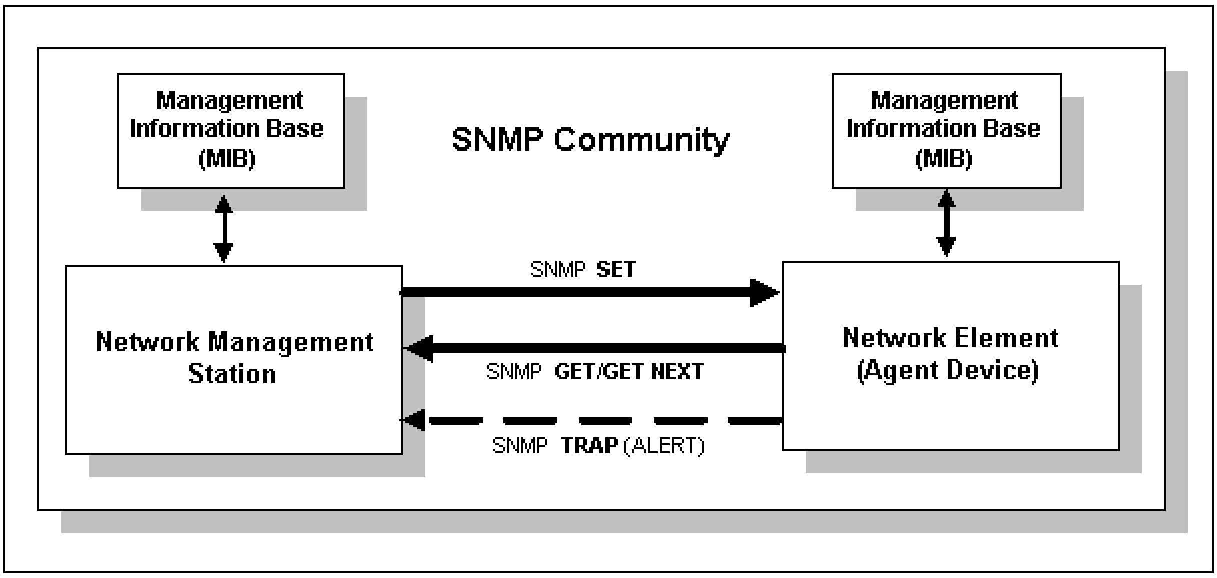
- **smtp_connect()** - connect to SMTP Server and send an e-mail.

- In order to use an Embedded SMTP Client, you must **enable** and **configure** it in the **configuration** file.

SNMP Agent

Simple Network Management Protocol (SNMP) is mainly used in network management systems to **monitor** network-attached devices for conditions that warrant administrative attention. It is the most popular network management protocol in the TCP/IP protocol suite.

SNMP is a simple request/response protocol that communicates management information between two types of SNMP software entities: **SNMP managers** and **SNMP agents**.


Management Information Base (MIB)
SNMP Community
Management Information Base (MIB)
SNMP SET
Network Management Station
SNMP GET/GET NEXT
Network Element (Agent Device)
SNMP TRAP (ALERT)

In summary, the SNMP Management program performs the following operations:

- The **GET** operation receives a specific value about a managed object, such as the available hard disk space from the agent's MIB.
- The **GET-NEXT** operation returns the "next" value by traversing the **MIB tree** of managed object variables.
- The **SET** operation changes the value of a managed object's variable. Only variables whose object definition allows read/write access can be changed.
- The **TRAP** operation sends a message to the Management Station when a change occurs in a managed object, and that change is important enough to send an alert message.

The SNMP Agent validates each request from an SNMP manager before responding to the request, by verifying that the manager belongs to an **SNMP community** with access priviliges to the agent. An SNMP community is a logical relationship between an SNMP agent and one or more SNMP managers. The community has a **name**, and all members of a community have the same access privileges: either read-only or read-write.

An Embedded SNMP Agent is an optimized and compact implementation for embedded systems. Currently it implements **SNMP version 1**.

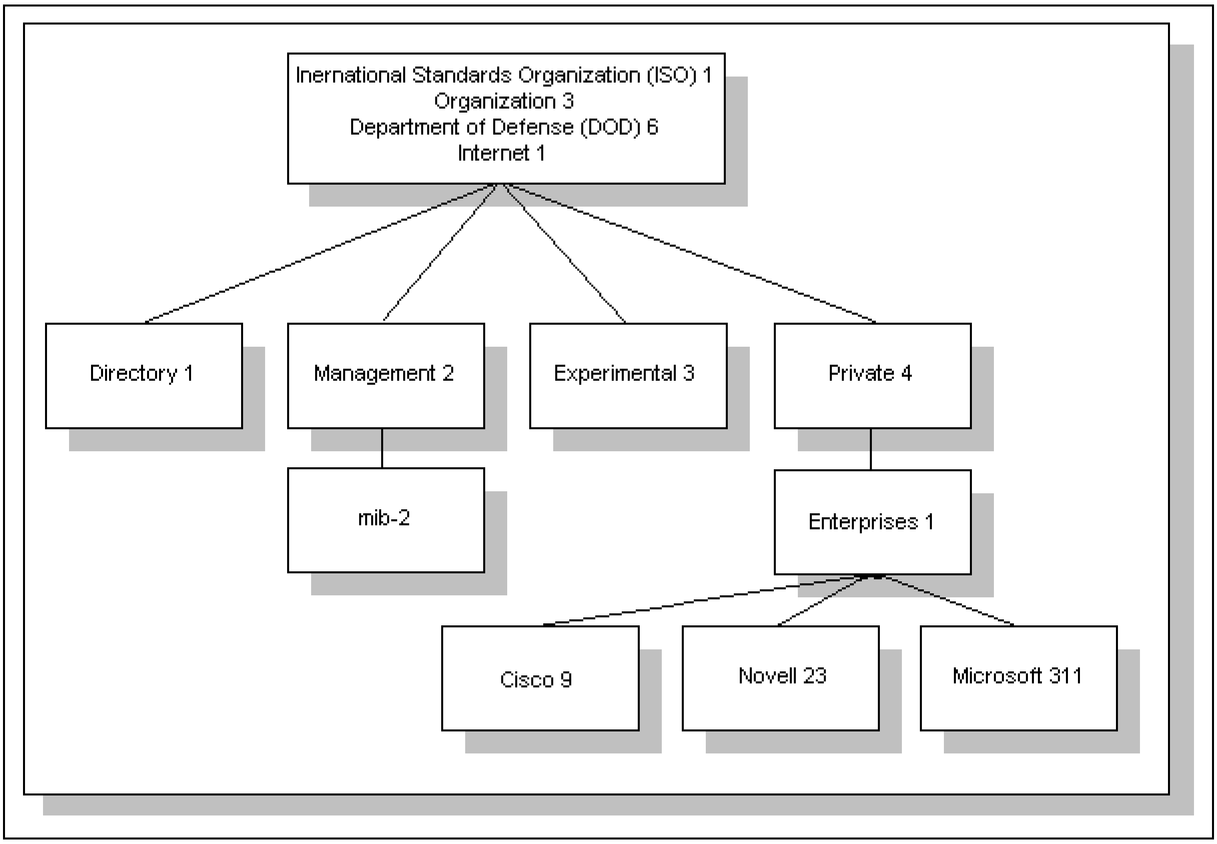
- In order to use an Embedded SNMP Agent, you must **enable** and **configure** it in the **configuration** file.

MIB Database

The data base controlled by the SNMP Agent is referred to as the **SNMP Management Information Base** (MIB). It is a standard set of statistical and control values. SNMP allows the extension of these standard values with values specific to a particular agent through the use of private MIBs.

The definitions of MIB variables supported by a particular agent are incorporated in descriptor files, written in **Abstract Syntax Notation** (ASN.1) format, made available to network management client programs so that they can become aware of these MIB variables and their usage.

The **OID naming scheme** is governed by the Internet Engineering Task Force (IETF). The IETF grants authority for parts of the name space to individual organizations such as Microsoft, Novell or Cisco. For example, Microsoft has the authority to assign the OIDs that can be derived by branching downward from the node in the MIB name three that starts with 1.3.6.1.4.1.311. Novell's OIDs branch down from 1.3.6.1.4.1.23. etc. You can see this structure in the diagram below.



The MIB variables are referred to as MIB object identifiers - **OID**s. OID names are hierarchy structured and unique. SNMP uses the **OID** to identify objects on each network element (device running SNMP agent) that can be managed using SNMP.

MIB Interface

The Embedded SNMP Agent manages MIB variables that are located in **SNMP_MIB.c**, which is a user interface module. This module is in the **\Keil\ARM\RL\TCPnet\User** folder. You must copy it to your project directory and add it to your project.

The SNMP_MIB.c module has implemented a scaled-down **MIB-II** Management Information Base. Only the **System MIB** is defined by default. The user might expand this table by adding his own MIB variables.

The user can register a **callback function** with a MIB variable. This function gets called, when the SNMP Manager accesses the MIB variable. This concept allows the SNMP Manager to control the SNMP Agent system. For example to change LED outputs, to write text on embedded LCD module, to read push buttons or analog inputs, etc.

MIB Entry

The **MIB_ENTRY structure** describes the MIB variable. SNMP Agent uses this description to process local MIB variables. This structure is defined in Net_Config.h as follows:

```
typedef struct mib_entry {          /* << SNMP-MIB Entry Info >>            */
  U8    Type;                       /* Object Type                          */
  U8    OidLen;                     /* Object ID length                     */
  U8    Oid[MIB_OIDSZ];             /* Object ID value                      */
  U8    ValSz;                      /* Size of a Variable                   */
  void *Val;                        /* Pointer to a variable                */
  void (*cb_func)(int mode);        /* Write/Read event callback function */
} MIB_ENTRY;
```

The components of **MIB_ENTRY** structure are:

- the *Type* defines the MIB variable type:

MIB *Type*	Description	Size
MIB_INTEGER	Signed Integer	1, 2 or 4 bytes
MIB_OCTET_STR	Octet String entry	max. 110 characters
MIB_OBJECT_ID	Object Identifier entry	max. 13 bytes
MIB_IP_ADDR	IP Address entry	4 bytes
MIB_COUNTER	Counter entry	1, 2 or 4 bytes
MIB_GAUGE	Gauge entry	1, 2 or 4 bytes
MIB_TIME_TICKS	Time Ticks entry	4 bytes

- The *Type* component may be or-ed with the **MIB_ATR_RO** read-only attribute. A read-only variable can not be changed by the SNMP Manager.
- the *OID* specifies the Object Identification Name of the variable. It is length encoded.
  - *OidLen* specifies the length of the *Oid[]* array.
  - *Oid[MIB_OIDSZ]* array specifies the OID name - a length encoded binary array.
- the *Val* specifies the Pointer to the variable and it's Size.
  - *ValSz* specifies the size of *Val* variable.
  - _*Val_ is a pointer to the actual variable.
- the *cb_func* specifies a Callback function which is called, when the variable is accessed by SNMP Manager. The callback function is not registered, when the value of *cb_func* is **NULL**.

  Parameter *mode* of the callback function specifies the access mode of SNMP Manager:

Mode	Type of Access
MIB_READ	Reads a MIB variable.
MIB_WRITE	Writes to a MIB variable.

MIB Table

The **snmp_mib** table is defined as an array. The components of this array are of type **MIB_ENTRY**.

```
const MIB_ENTRY snmp_mib[] = {
  /* ---------- System MIB ----------- */
  /* SysDescr Entry */
  { MIB_OCTET_STR | MIB_ATR_RO,
    8, {OID0(1,3), 6, 1, 2, 1, 1, 1, 0},
    MIB_STR("Embedded System SNMP V1.0"),
    NULL },
  /* SysObjectID Entry */
  { MIB_OBJECT_ID | MIB_ATR_RO,
    8, {OID0(1,3), 6, 1, 2, 1, 1, 2, 0},
    MIB_STR("\x2b\x06\x01\x02\x01\x01\x02\x00"),
    NULL },
  /* SysUpTime Entry */
  { MIB_TIME_TICKS | MIB_ATR_RO,
    8, {OID0(1,3), 6, 1, 2, 1, 1, 3, 0},
    4, &snmp_SysUpTime,
    NULL },
    ..
}
```

In the following example, we will construct a MIB variable entry **LedOut**. It will allow SNMP Manager to control LED diodes on an evaluation board.

- The MIB variable **type** is Integer. An U8 variable is sufficient, because the LED port is 8-bit:

  ```
    /* LedOut Entry */
    { MIB_INTEGER,
  ```

- The OID reference is 1.3.6.1.3.1.0. It is defined in the **Experimental** MIB branch of the MIB tree:

  ```
      6, {OID0(1,3), 6, 1, 3, 1, 0},
  ```

  - the first byte defines the **length** of the OID name,
  - macro **OID0** calculates the first byte of OID value from 1st and 2nd address byte,
  - the value of an OID address byte must be less than 128, otherwise an **extended** encoding must be used.
- The variable size and location is described with the help of **MIB_INT** macro:

  ```
      MIB_INT(LedOut),
  ```

  The following macros are defined:

Macro	Variable Definition
MIB_STR	Octet String size and location.
MIB_INT	Signed or Unsigned Integer size and location.
MIB_IP	IP Address size and location.

- The **write_leds** is specified as callback function. It gets called when the **LedOut** is written:

  ```
      write_leds },
  ```

- Finally we need the actual variable definition:

  ```
  static U8 LedOut;
  ```

For the **LedOut** control we actually need the following parts of code to be defined in SNMP_MIB.c module:

```
static U8 LedOut;
static void write_leds (int mode);

const MIB_ENTRY snmp_mib[] = {
    ..
  /* LedOut Entry */
  { MIB_INTEGER,
    6, {OID0(1,3), 6, 1, 3, 1, 0},
    MIB_INT(LedOut),
    write_leds },
    ..
}

static void write_leds (int mode) {
  /* No action on read access. */
  if (mode == MIB_WRITE) {
    LED_out (LedOut);
  }
}
```

Extended OID encoding

The value of OID address byte must be less than **128**. If it is not, an OID address must be encoded in extended format. This is because the high bit of an address byte is an address **extension bit**.

For example, the OID address 1.3.6.1.4.1.**311**.0 is encoded as:

```
  8, {OID0(1,3), 6, 1, 4, 1, 130, 55, 0},
```

The address value for the highlighted numbers is calculated as:

```
  (130-128) * 128 + 55 = 311
```

The OID address 1.3.6.1.4.1.**31036**.50.1.1.0 is encoded as:

```
  12, {OID0(1,3), 6, 1, 4, 1, 129, 242, 60, 50, 1, 1, 0},
```

where the value 31036 is calculated as:

```
  (129-128) * 128 * 128 + (242-128) * 128 + 60 = 31036
```

DNS Resolver

Domain Name System (DNS) servers store and manage information about **domains** and respond to **resolution requests** for clients (in some cases millions of times each day). The DNS database is a **distributed** name database stored on many DNS servers. DNS uses a hierarchical tree structure for its name space and a hierarchical tree for name authorities and registration.

Since information in DNS is stored in a distributed form, there is no single server that has information about every domain in the system. The process of resolution instead relies on the hierarchy of name servers as described above.

At the top of the DNS hierarchy is the **root domain** and the **root name servers**. These are the most important servers because they maintain information about the top-level domains within the root. They also know the servers that can be used to resolve domains one level below them. Those servers can reference servers that are responsible for second-level domains. Thus, a DNS resolution requests might be sent to more than one server.

An Embedded DNS Resolver is capable of resolving the **IP address** of a host from the **host's name**. It does this by sending DNS requests to a DNS Server. The IP address of a DNS Server is specified in the **configuration** or can be obtained from the **DHCP Server** for the Local Area Network.

The Embedded DNS Resolver **caches** the resolved IP addresses. The length of time the resolved host IP address is kept in the local cache depends on the Time to Live (TTL) timeout. This is returned in an answering packet from the DNS Server. The next time a DNS is requested, the cache table is checked first. If a valid host is found, the IP address is resolved from the cache and no actual DNS request is sent to the DNS Server.

You must use the DNS Resolver when a remote host uses a **Dynamic IP**, which changes each time the remote host logs on to the internet.

Starting DNS

Start the DNS Resolver by calling the function **get_host_by_name()**. DNS Requests are routed to the DNS Server IP address of an active network interface. If you are using a **PPP** or **SLIP** interface and no ethernet interface, you must enable the **Use default gateway on remote network** option in the configuration.

You must also specify a **callback function**, which is called from the DNS Client when a DNS event occurs.

```
static void dns_cbfunc (U8 event, U8 *ip) {
  switch (event) {
    case DNS_EVT_SUCCESS:
      printf("IP Address: %d.%d.%d.%d\n",ip[0],ip[1],ip[2],ip[3]);
      break;
    case DNS_EVT_NONAME:
      printf("Host name does not exist.\n");
      break;
    case DNS_EVT_TIMEOUT:
      printf("DNS Resolver Timeout expired, Host IP not resolved.\n");
      break;
    case DNS_EVT_ERROR:
      printf("DNS Resolver Error, check the host name, labels, etc.\n");
      break;
  }
}
```

When the required host is found in the local DNS Cache, the callback function is called immediately with the result code **DNS_EVT_SUCCESS** and provides the IP address of the host to the function. In this case, no actual DNS request packet is sent to the remote DNS Server.

Note

- To use an Embedded DNS Client, you must **enable** and **configure** it in the **configuration** file.
- You can also provide the **IP address** in a string format to specify the host name. The DNS Client decodes it and returns the decoded IP address to the callback function.

Device Drivers

RL-TCPnet uses device drivers to interface the physical transport media. The purpose of a device driver is to move all device specific functions to a single module, which can be customized by the user. The device driver functions are OS independent.

- **Ethernet Network Driver** interfaces with the ethernet controller. This driver sends the ethernet packets to a network and receives incoming packets, which it then stores in a memory buffer.
- **Modem Driver** handles the modem connection when using the PPP or SLIP network interface. It dials a remote target number on request and accepts incoming calls.
- **Serial Driver** is used when using the PPP or SLIP network interface. This is an interrupt driven serial interface that stores received characters in a buffer and sends outgoing data to a serial port.

All driver functions must be included in a single driver module. It is a good practice to **name** this module by the type of the ethernet controller like CS8900A.c or Ax88796.c.

Ethernet Driver

For the Ethernet Driver in polling mode, the system frequently calls the function **poll_ethernet()**. This mode gives you more simplicity and less performance. However, if the ethernet controller has an integrated **large memory buffer**, polling mode is a good choice.

The **required functions** for implementation of the driver are:

- **init_ethernet()** - initializes the ethernet controller
- **poll_ethernet()** - reads the packet out of the ethernet controller's buffer
- **send_frame()** - writes a packet to the ethernet controller.

You can also use the Ethernet interface in interrupt mode.

RL-TCPnet includes several Ethernet Network drivers. These are located in the **\Keil\ARM\RL\TCPnet\Drivers** directory:

- **LAN91C111.C** - for the SMSC LAN91C111 Ethernet Controller used on the Phytec phyCore LPC229x evaluation board.
- **AT91_EMAC.C** - for the Atmel AT91SAM7X on-chip EMAC Ethernet Controller used on the Atmel AT91SAM7X-EK evaluation board.
- **STR9_ENET.C** - for the ST STR912 on-chip ENET Ethernet Controller used on the Keil MCBSTR9 evaluation board.
- **LPC23_EMAC.c** - for the NXP (founded by Philips) LPC2368 and LPC2378 on-chip EMAC Ethernet Controllers used on the Keil MCB2300 evaluation board.
- **LPC24_EMAC.c** - for the NXP (founded by Philips) LPC2468 and LPC2478 on-chip EMAC Ethernet Controllers used on the Keil MCB2400 evaluation board.
- **LM3S_EMAC.c** - for the Luminary Micro LM3S6962 and LM3S8962 on-chip EMAC Ethernet Controllers used on the Luminary Micro EK-LM3S6965 and EK-LM3S8962 evaluation boards.

You can use these source files as a template for your own Ethernet Network driver.

note

- If you want to use the Ethernet Network interface in your project, you must copy the Ethernet Network driver to your project directory and add it to your project.

Interrupt Mode

The Interrupt Device Driver gives you better performance and eliminates the risk of losing a packet when the system reaction time is slow. This can happen if the ethernet traffic is high and the user event function takes very long to process.

When the ethernet controller integrates a **small memory buffer** (for example a couple of kilobytes), the interrupt mode is necessary to prevent possible packet reception problems.

The **required functions** for implementation of the driver are:

- **init_ethernet()** - initializes the ethernet controller
- **interrupt_ethernet()** - interrupt service routine that reads the packet out of the ethernet controller's buffer and stores it to the RAM buffer
- **send_frame()** - writes a packet to the ethernet controller
- **int_disable_eth()** - disables the interrupts of the ethernet controller
- **int_enable_eth()** - enables the interrupts of the ethernet controller.

Modem Driver

The Modem Driver controls an attached modem, dials an outgoing target number, and handles incoming calls. The Modem Driver is used when the PPP or SLIP serial network interface is enabled. You must use a **Null_Modem** driver for the zero-modem link if you use a serial cable to connect the embedded device directly to a computer.

The **required functions** for the driver are:

- **init_modem()** - initialize the modem
- **modem_dial()** - dial a target number
- **modem_listen()** - initialize the modem to accept incoming calls
- **modem_hangup()** - stop the connection
- **modem_online()** - check if the modem is online
- **modem_process()** - process a character received from the modem in command mode
- **modem_run()** - the main thread for the modem driver.

RL-TCPnet includes two modem device drivers which are located in the **\Keil\ARM\RL\TCPnet\User** directory:

- **Null_Modem.c** is a zero-modem link driver used when a computer is connected directly to the target hardware. The computer **simulates** a real modem. The Null_Modem driver responds to modem commands sent from a computer in the same way as it responds to a real modem. It also works with the **direct cable connection** link supported by MS Windows.
- **Std_Modem.c** is a standard modem driver. This driver works with most of the currently available modems. Use the standard modem driver when a real modem is used to access the embedded device over a public telephone network.

You can copy these files into your project folder or use them as a **template** to write your own modem driver.

- You need to include **only one** modem driver into your project for the PPP or SLIP interface. For a zero-modem link, this is not required because the default **Null_Modem** driver is already in the RL-TCPnet library.
- If you want to customize the **Null_Modem.c** modem driver, copy it from the **\Keil\ARM\RL\TCPnet\User** directory and add it to your project. Then customize it as needed.

Serial Driver

The Serial Driver handles a **serial port** and is required for a PPP or SLIP serial network interface. The Serial Driver is **interrupt** driven since polling for a new character sometimes fails at higher baud rates and some of the received characters might be lost.

The serial receive interrupt function stores incoming characters to an intermediate **input buffer**. The serial transmit interrupt function sends outgoing data from an **output buffer** to a serial port.

If you use a serial PPP or SLIP network interface, then **copy** the serial device driver to your project directory and **add** it to your project.

The **required functions** of the driver are:

- **init_serial()** - initialize the serial interface
- **com_getchar()** - get a character from an input buffer
- **com_putchar()** - put the character to an output buffer
- **com_tx_active()** - check if the serial transmit is active.

RL-TCPnet includes various serial device drivers for different ARM device variants. The driver filename syntax used is **Serial_xxxx.c**, where **xxxx** is the device name. The drivers are located in the **\Keil\ARM\RL\TCPnet\User** directory:

- **Serial.c** is configured for Philips LPC21xx devices. Use this driver for all Keil MCB21xx evaluation boards.
- **Serial_LPC214x.c** is configured for Philips LPC214x devices. This driver uses extended features which allow configuring various baud rates using built-in fractional baud rate generator.
- **Serial_S3C44B0X.c** is configured for Samsung S3C44B0X devices.

For ARM device variants that are not currently supported, or if you are using an external UART, use the **Serial.c** device driver as a **template** and customize it for your needs.

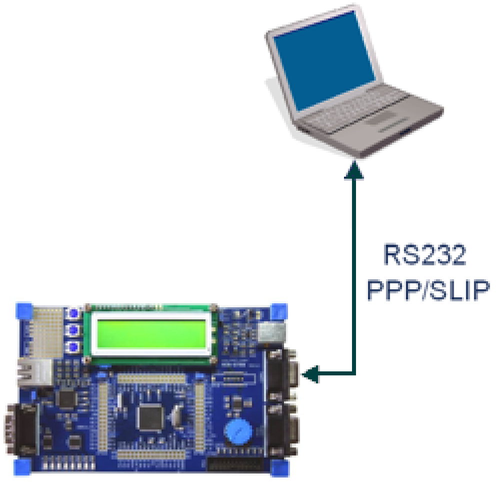
Using Serial Link

These topics include instructions on how to use a serial link, a **serial cable**, or a **modem** connection to establish a PPP or SLIP link with a Windows 2000 host.

The Keil evaluation board is the PPP (or SLIP) **server** and the Windows host is the **client**. Before a serial link can be established, the Windows host must be configured to use a **direct serial device** or a **standard modem** for dial-up network connections.

Cable Connection

If you want to connect the embedded device directly to a computer, you must use the Null Modem driver. The figure belows shows an overview of a directly connected system.



However, RL-TCPnet's Null Modem driver supports two different procedures for establishing the null modem serial link with the computer:

- **Windows Direct Serial Link**. This is a non-modem link. This system requires the exchange of string tokens between the computer and the embedded device before the PPP link can be negotiated. The SLIP link needs no negotiation because SLIP is a simple protocol over a serial line.

  The Windows client sends the string "CLIENT". The Keil evaluation board must then send the string "CLIENTSERVER". When operating as a server, the Keil evaluation board must send the string "CLIENT". The Windows client must respond with the string "CLIENTSERVER".

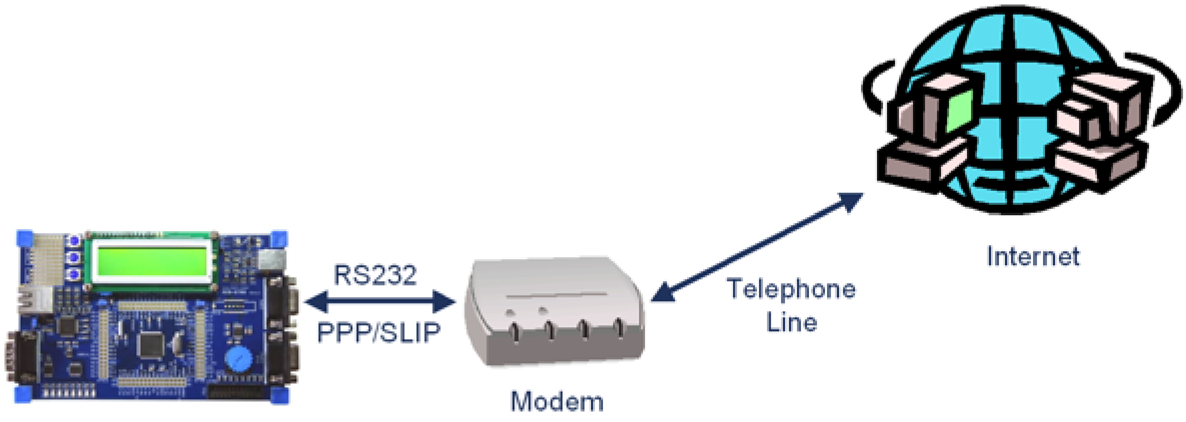
  This string exchange must occur before Windows allows the PPP connection to proceed. This string exchange is peculiar to Windows and is not required by other hosts.
- **Windows Standard Modem Link**. In this system, the **Null Modem driver** simulates an external modem. The PPP source must also be modified to operate with the Windows direct serial driver. It requires the exchange of modem command strings between the computer and the evaluation board before the PPP link can be negotiated. The strings to be exchanged are shown in the table below.

Windows Command	Reply	Description
AT\r	**OK\r**	Attention Command
ATE0V1\r	**OK\r**	Set No echo and Verbose Result reply
AT\r	**OK\r**	Attention Command
ATS0=0\r	**OK\r**	Disable Modem Auto Answer
AT\r	**OK\r**	Attention Command
ATE0V1\r	**OK\r**	Set No echo and Verbose Result reply
AT\r	**OK\r**	Attention Command
ATDTxxxxxxx\r	**CONNECT\r**	Dial Target number: xxxxxxx

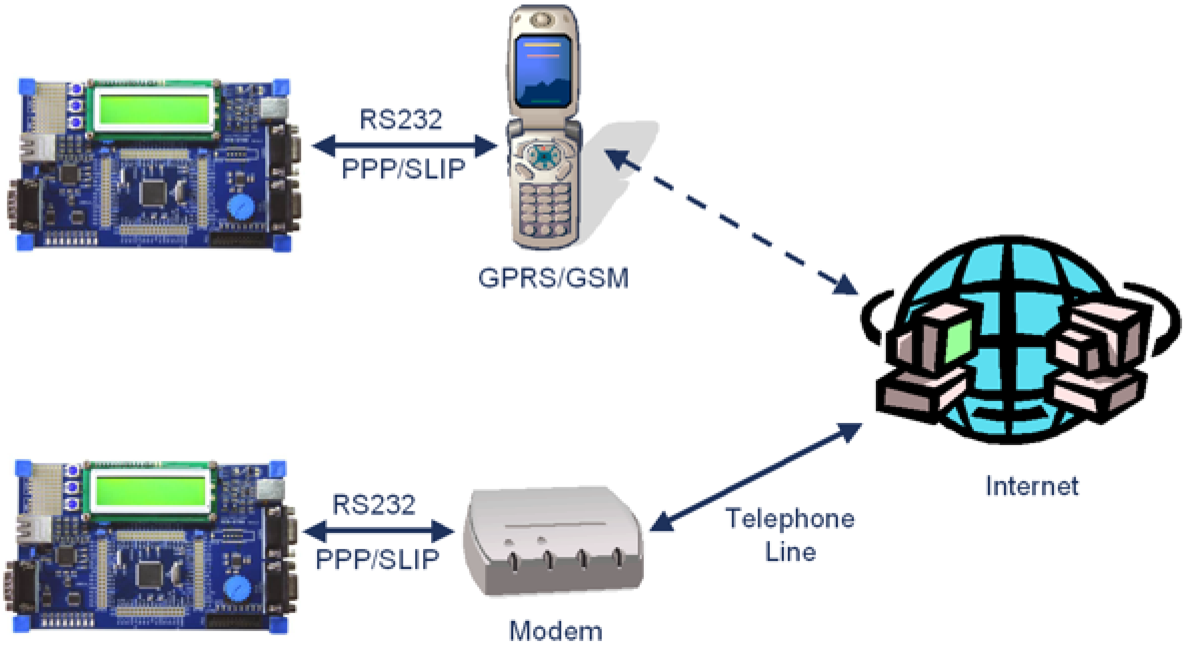
- After the "CONNECT" string is received, Windows starts the PPP negotiations and user authentication.

Modem Connection

Another way to connect to the Internet is the usage of a classic modem with an RS232 interface. In this case the SLIP or PPP interface is used to establish the modem connection.



The SLIP or PPP interface can be also used with a GPRS/GSM phone that allows a wireless connection to the Internet.



RL-TCPnet supports a **modem connection**. A modem connection can be established with any type of analog, ISDN, GSM or GPRS modem. In this case, the TCPnet modem driver takes care of the modem commands which are sent to the modem to dial, accept a call, or hangup. The modem dial-up connection is setup like a standard Internet dial-up connection.

Windows Dial-up

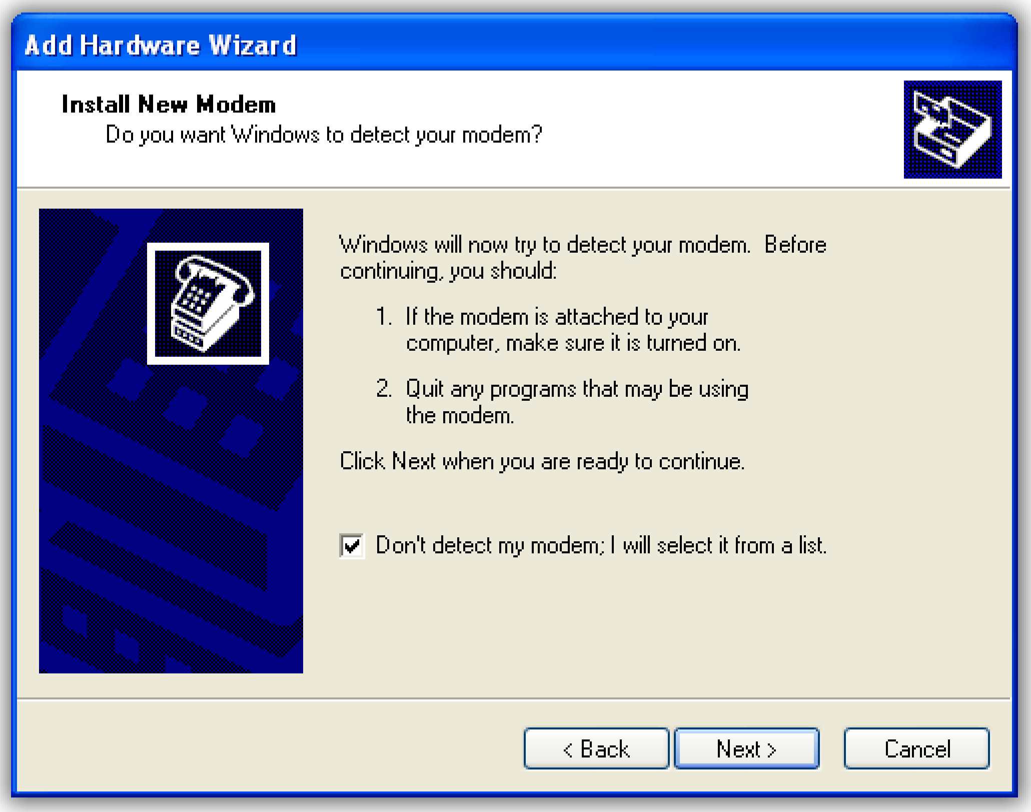
These instructions tell you how to configure your Windows dial-up connection so you can connect to a system that uses a serial **PPP link** or a **SLIP link**.

The PPP link is preferable because it provides **link control** protocol, IP **address negotiation**, and user **authentication**. All these features are not available in a SLIP link. SLIP is a simple protocol over the serial line. It was widely used with UNIX systems in the past.

Add Direct Serial Link

Follow these steps to add the direct serial link device in Windows XP.

1. Open the **Phone and Modem Options** applet found in the control panel. Select **Add** from the Modems tab on the Phone and Modem Options applet. The Add Hardware Wizard dialog appears. Enable the **Don't detect my modem...** option and click **Next**.



Add Hardware Wizard

Install New Modem

Do you want Windows to detect your modem?

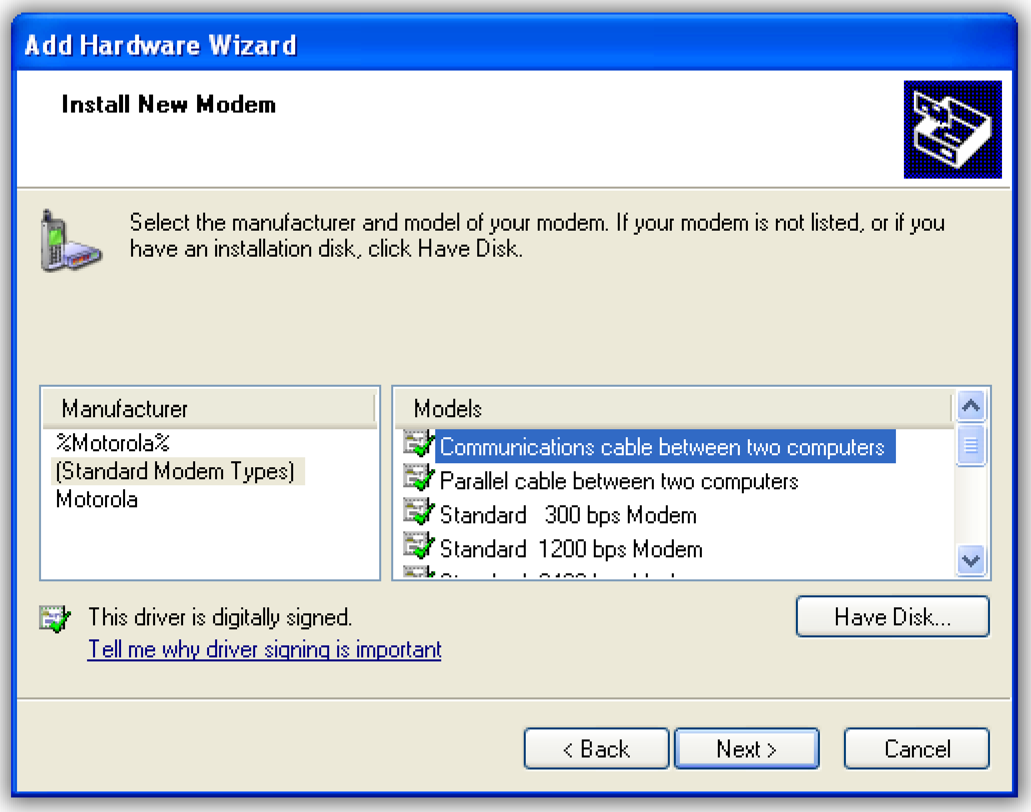
Windows will now try to detect your modem. Before continuing, you should:

1. If the modem is attached to your computer, make sure it is turned on.
2. Quit any programs that may be using the modem.

Click Next when you are ready to continue.

☑ Don't detect my modem; I will select it from a list.

2. On the next dialog, select **(Standard Modem Types)** in the left column, **Communications cable between two computers** in the right column, and click **Next**.



Add Hardware Wizard

Install New Modem

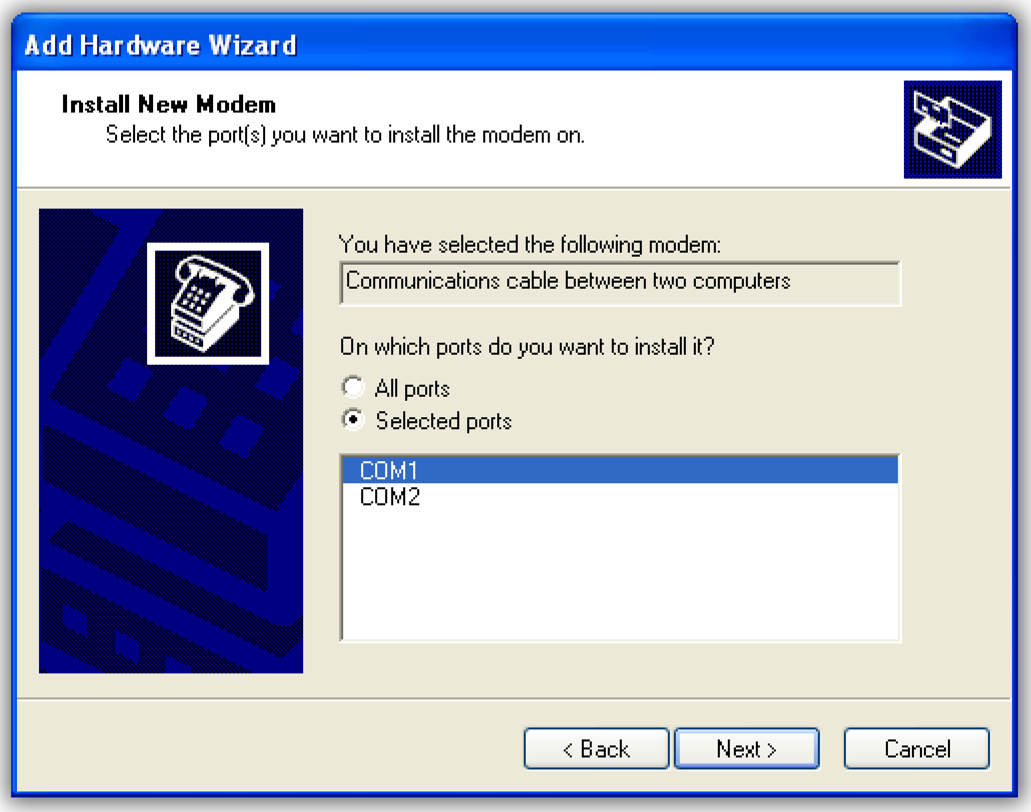
Select the manufacturer and model of your modem. If your modem is not listed, or if you have an installation disk, click Have Disk.

Manufacturer
%Motorola%
(Standard Modem Types)
Motorola

Models
Communications cable between two computers
Parallel cable between two computers
Standard 300 bps Modem
Standard 1200 bps Modem

This driver is digitally signed.
Tell me why driver signing is important

3. Select the appropriate communications port and click **Next**.



Add Hardware Wizard

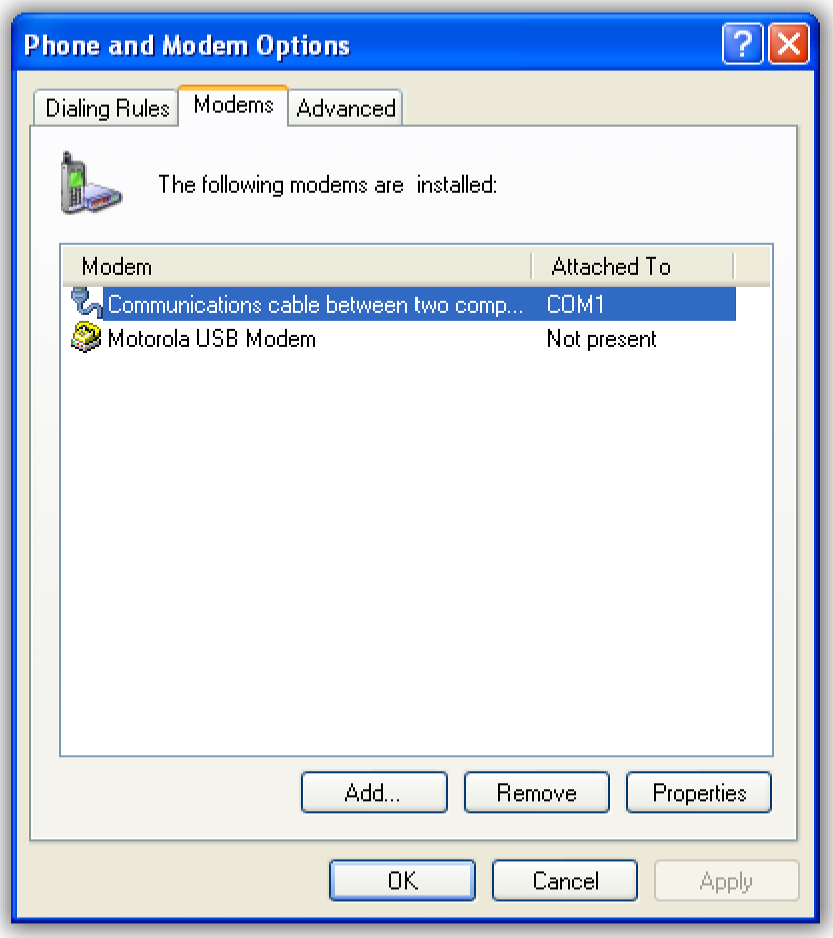
Install New Modem
Select the port(s) you want to install the modem on.

You have selected the following modem:
Communications cable between two computers

On which ports do you want to install it?
○ All ports
● Selected ports

COM1
COM2

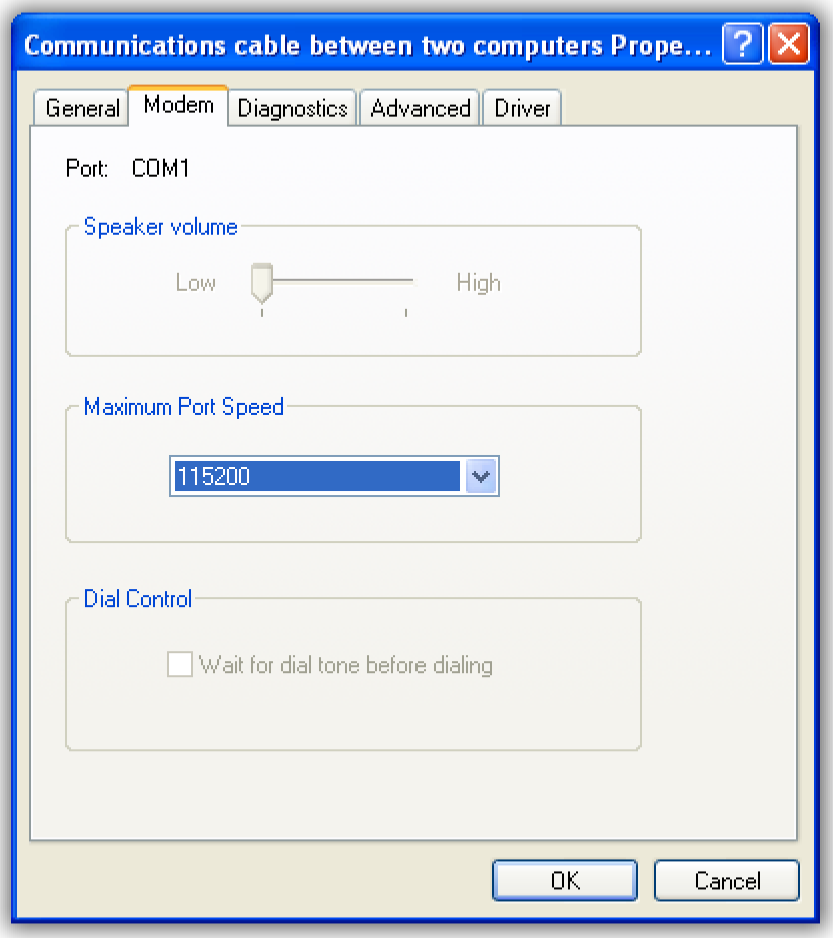
4. On the next dialog, click **Finish** to complete the installation of the Direct Serial Link device. After the new device is installed, click **OK** to close the Phone and Modem Options applet.
5. Next, limit the serial line baud rate. Open the Phone and Modem Options applet found in the control panel again. Select the **Modems** tab and double-click the modem you just installed. Alternatively, highlight it and click **Properties**.



The following modems are installed:

Modem	Attached To
Communications cable between two comp...	COM1
Motorola USB Modem	Not present

6. Limit the highest port speed to 115200 baud because the default Null Modem driver, in RL-TCPnet, uses this baud rate by default. Select 115200 from the drop down menu.



Port: COM1

115200

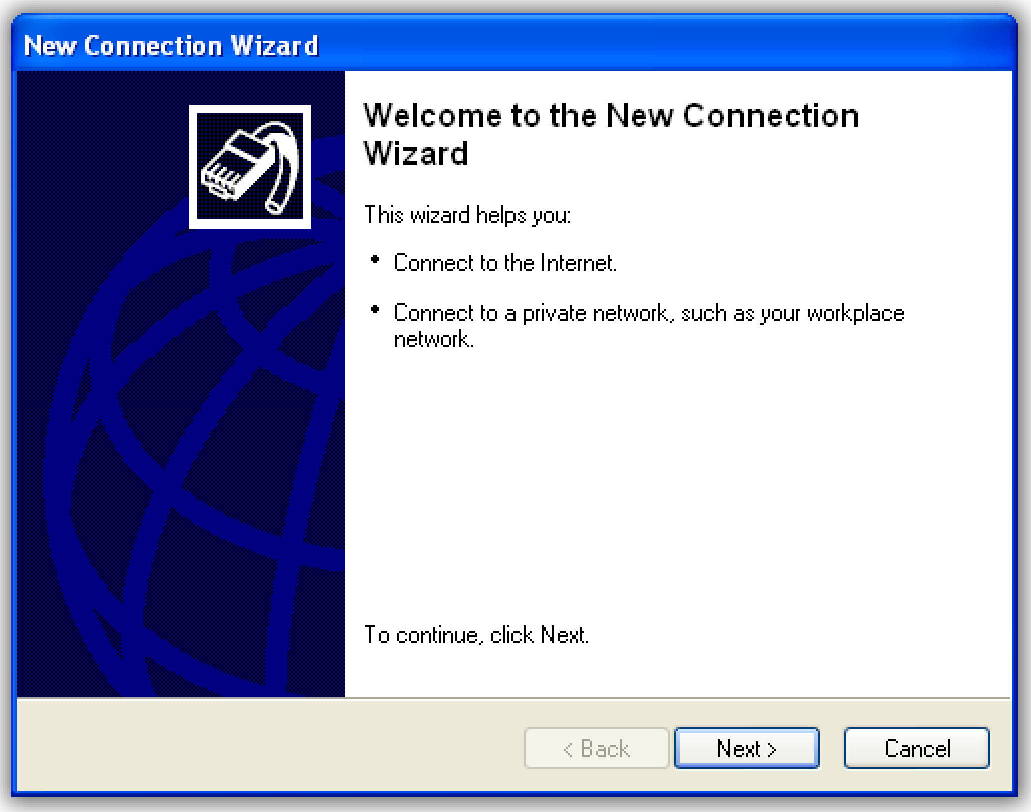
□ Wait for dial tone before dialing

7. After selecting the port speed, click **OK** twice to close the Phone and Modem Options applet.

New Dial-up Connection

Follow these steps to add a new dial-up connection in Windows XP.

1. Open the Network and Dial-Up Connections applet located in the control panel. Open **Make New Connection** to start the Network Connection Wizard and click **Next**.



New Connection Wizard

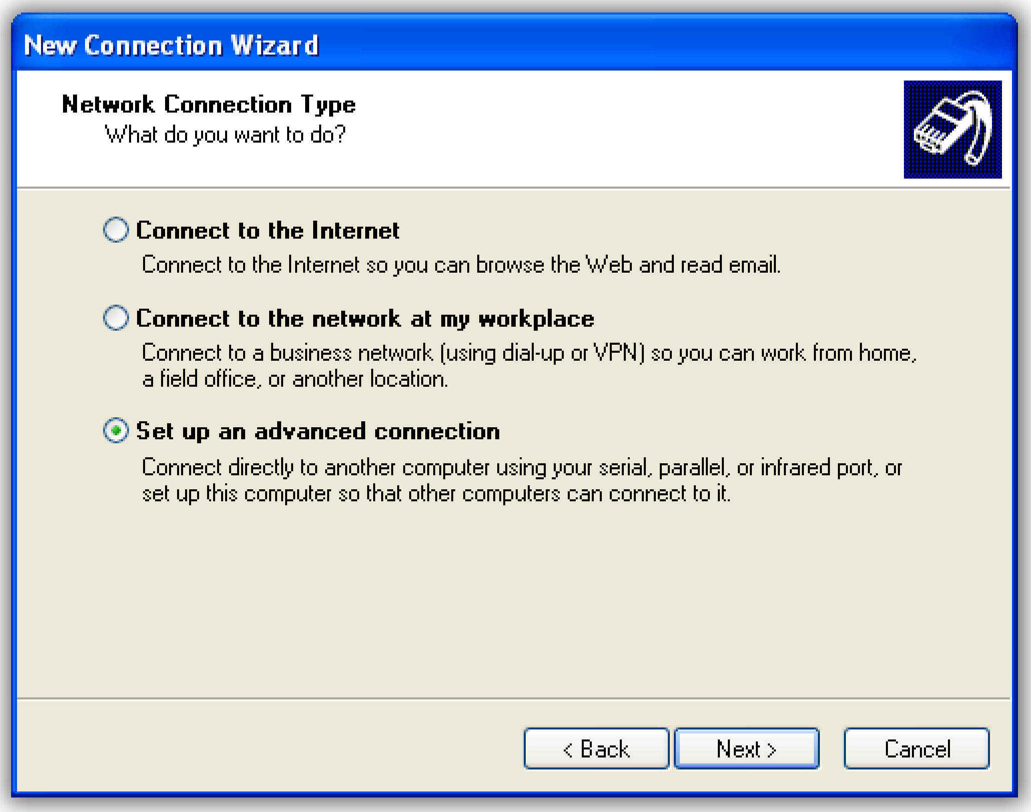
Welcome to the New Connection Wizard

This wizard helps you:

- Connect to the Internet.
- Connect to a private network, such as your workplace network.

To continue, click Next.

2. On the Network Connection Wizard dialog, select **Set up an advanced connection** and click **Next**.



New Connection Wizard

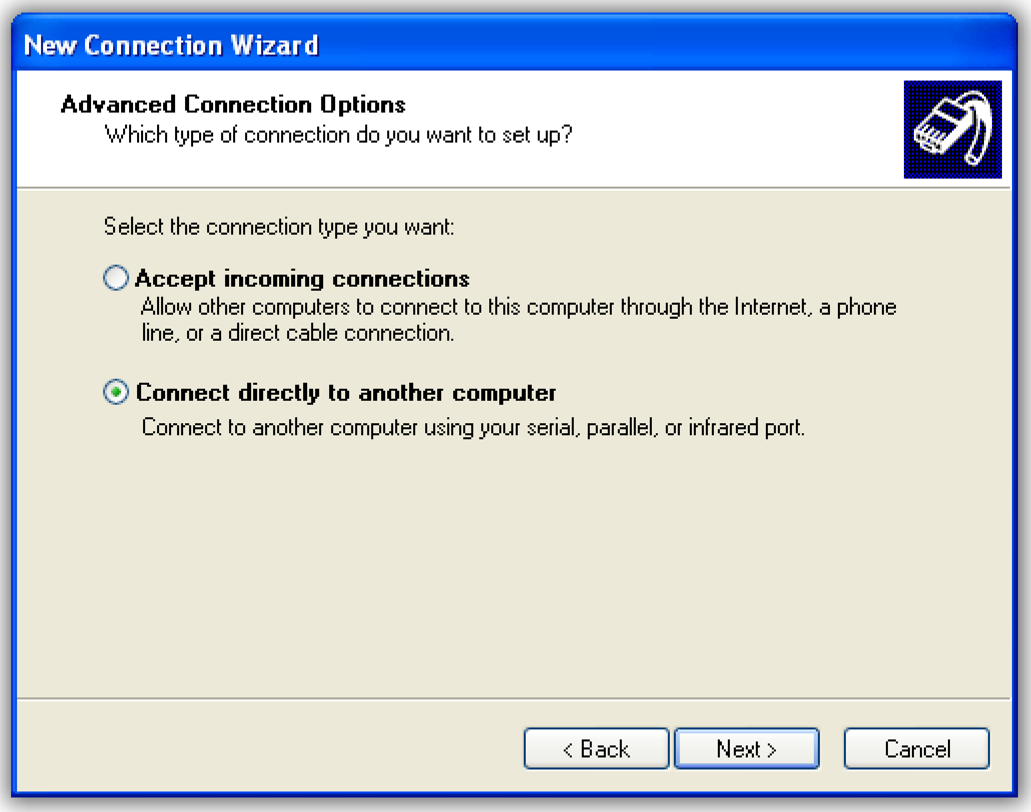
Network Connection Type
What do you want to do?

○ **Connect to the Internet**
Connect to the Internet so you can browse the Web and read email.

○ **Connect to the network at my workplace**
Connect to a business network (using dial-up or VPN) so you can work from home, a field office, or another location.

● **Set up an advanced connection**
Connect directly to another computer using your serial, parallel, or infrared port, or set up this computer so that other computers can connect to it.

3. On the Network Connection Wizard dialog, select **Connect directly to another computer** and click **Next**.



New Connection Wizard

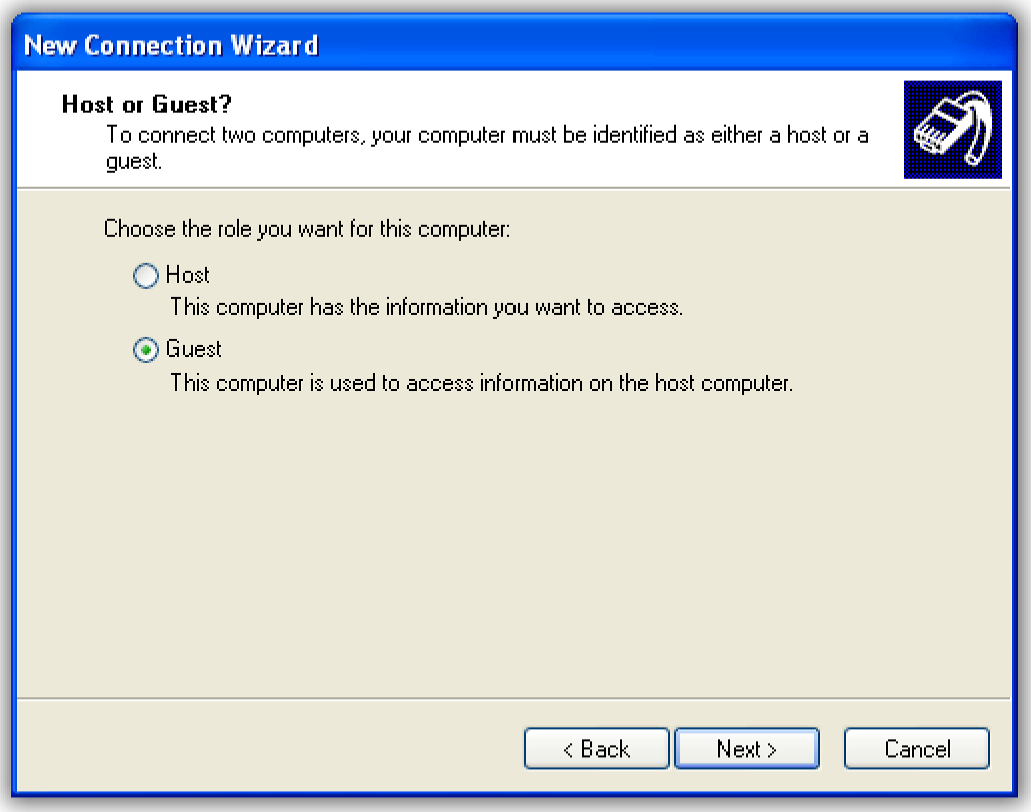
Advanced Connection Options
Which type of connection do you want to set up?

Select the connection type you want:

○ **Accept incoming connections**
Allow other computers to connect to this computer through the Internet, a phone line, or a direct cable connection.

● **Connect directly to another computer**
Connect to another computer using your serial, parallel, or infrared port.

4. On the Network Connection Wizard dialog, select **Guest** and click **Next**.



New Connection Wizard

Host or Guest?

To connect two computers, your computer must be identified as either a host or a guest.

Choose the role you want for this computer:

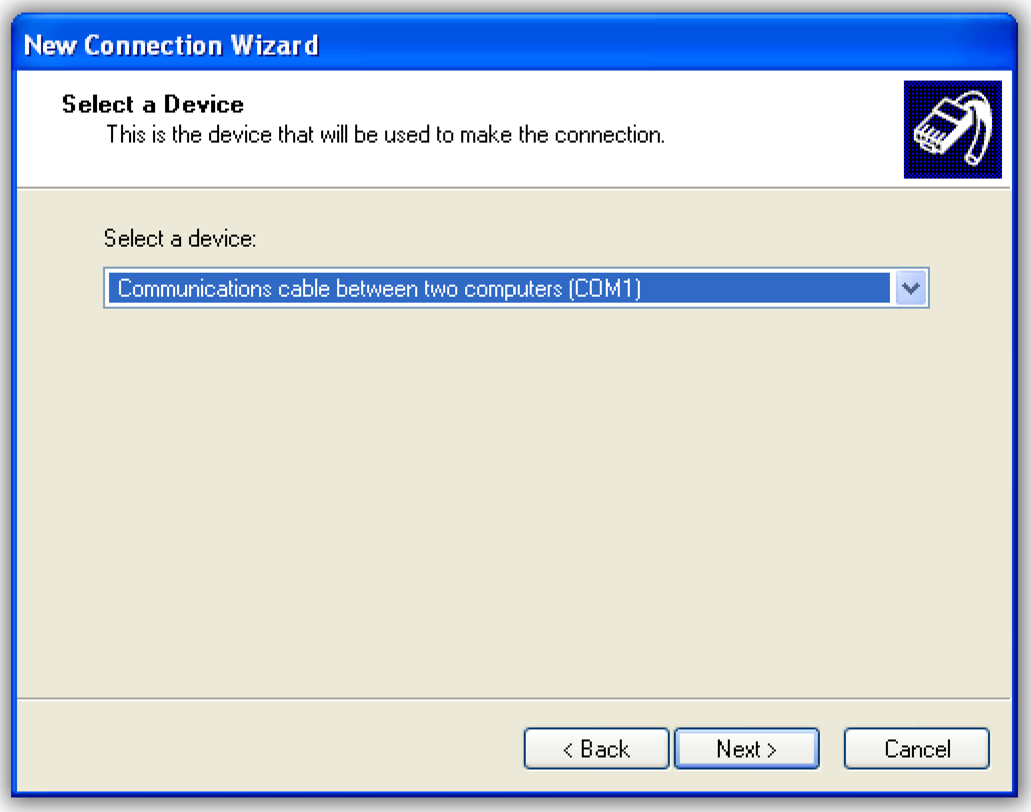
○ Host

This computer has the information you want to access.

● Guest

This computer is used to access information on the host computer.

5. On the Current Connection Wizard dialog, type in **name** for the new connection and click **Next**.
6. On the Network Connection Wizard dialog, select the **Communication cable between two computers...** option from the drop-down menu and click **Next**.



New Connection Wizard

Select a Device

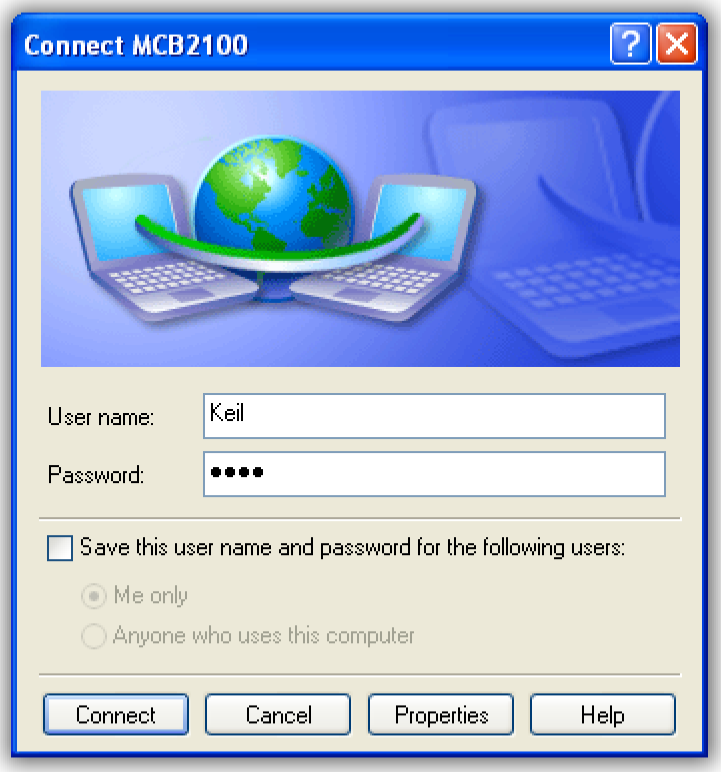
This is the device that will be used to make the connection.

Select a device:

Communications cable between two computers (COM1)

7. On the current Connection Wizard dialog, click **Next** to allow all users to use this connection. On the next dialog, name the new connection and click **Finish**.

8. The new dial-up connection dialog (Connect PPP) appears. Before using the new dial-up connection, you must verify the serial port speed. Click on **Properties** in the Connect PPP dialog.



User name: Keil

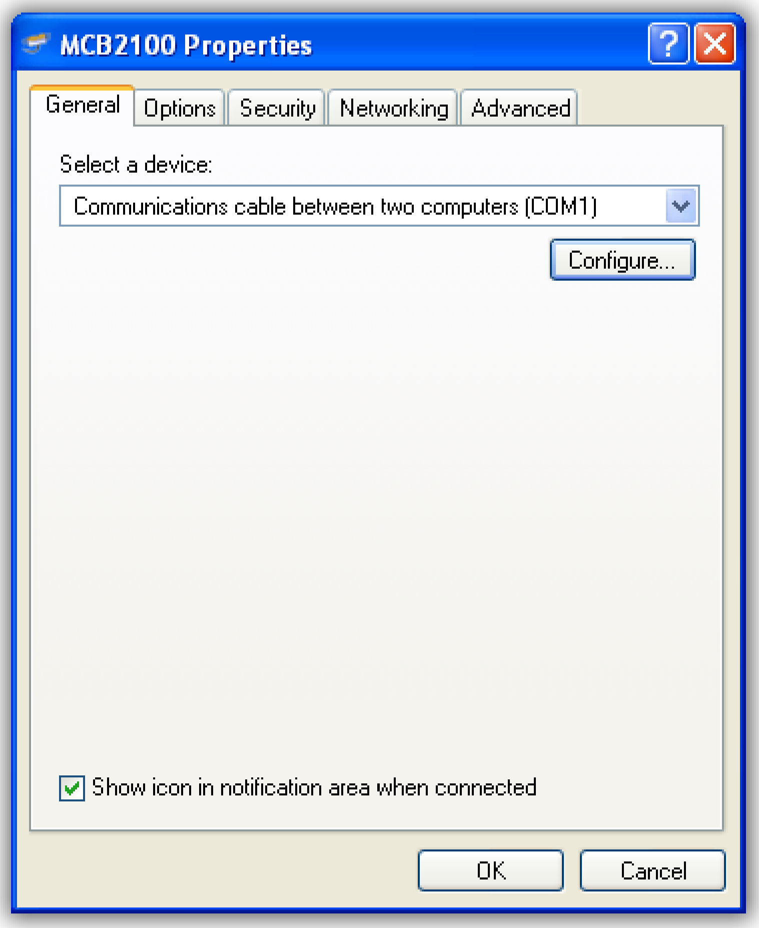
Password: ●●●●

☐ Save this user name and password for the following users:

● Me only

○ Anyone who uses this computer

9. The PPP Dialog opens and shows the General tab. Select the device from the drop-down menu (if there is more than one) and click **Configure**.

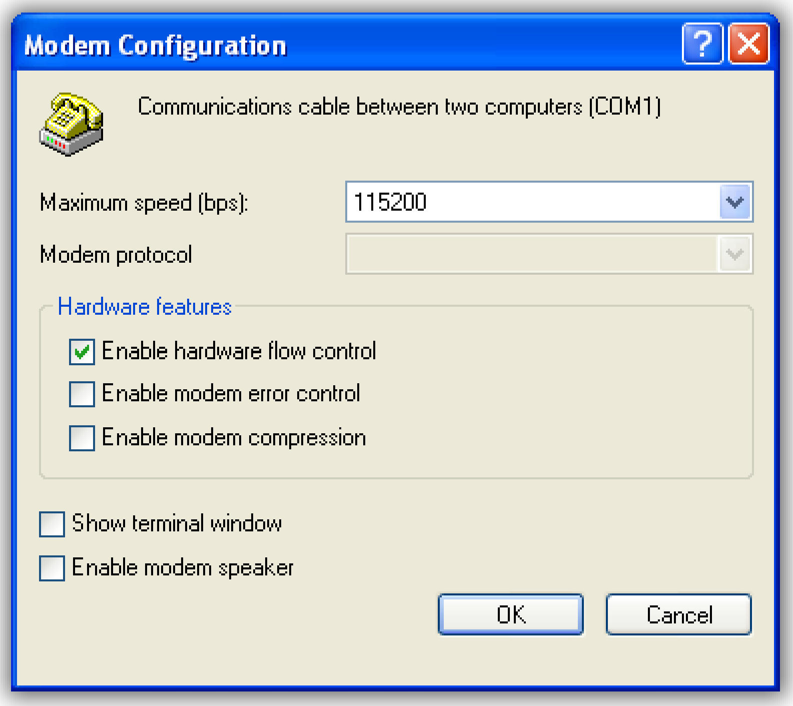


MCB2100 Properties

Select a device: Communications cable between two computers (COM1)

☑ Show icon in notification area when connected

10. The Modem Configuration dialog appears. Verify that the Maximum speed is set properly and click **OK**. The default Keil Evaluation Board baud rate is 115200 baud.



Modem Configuration

Communications cable between two computers (COM1)

Maximum speed (bps): 115200

Modem protocol

Hardware features

☑ Enable hardware flow control

☐ Enable modem error control

☐ Enable modem compression

☐ Show terminal window

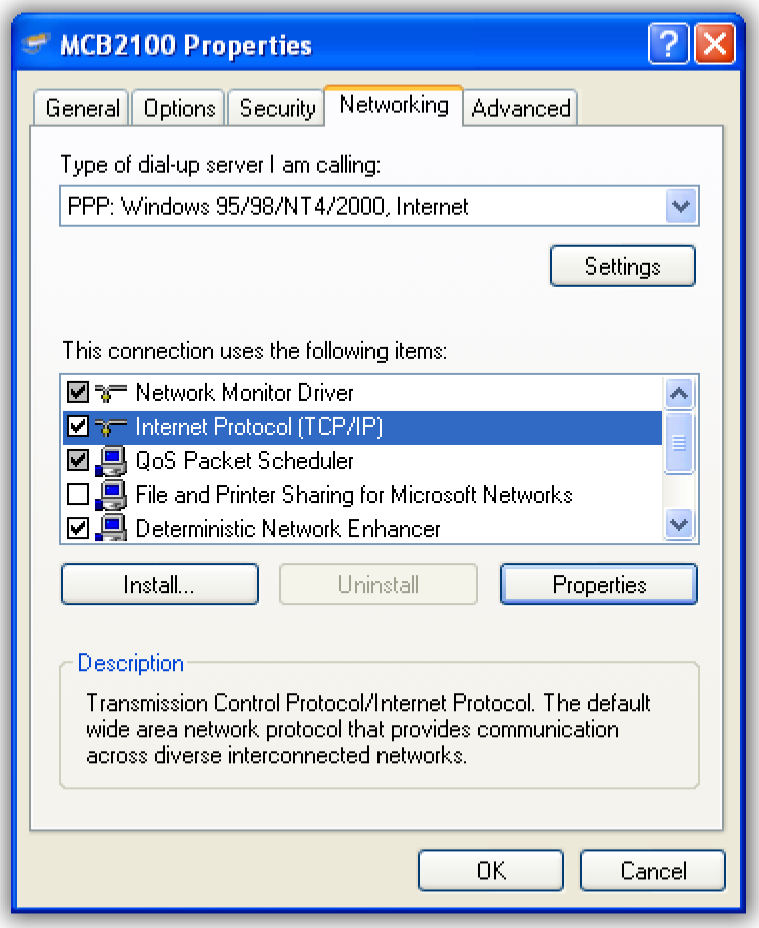
☐ Enable modem speaker

11. Click **OK** to close the dialog.

Configure PPP Dial-up

Follow these steps to configure the PPP dial-up.

1. In the Connection Properties dialog, click the **Networking** tab and select **PPP** from the "Type of dial-up server I am calling" drop-down menu. Then in the "Components checked are used by this connection:" section, check the **Internet Protocol (TCP/IP)** component and click **Properties**.



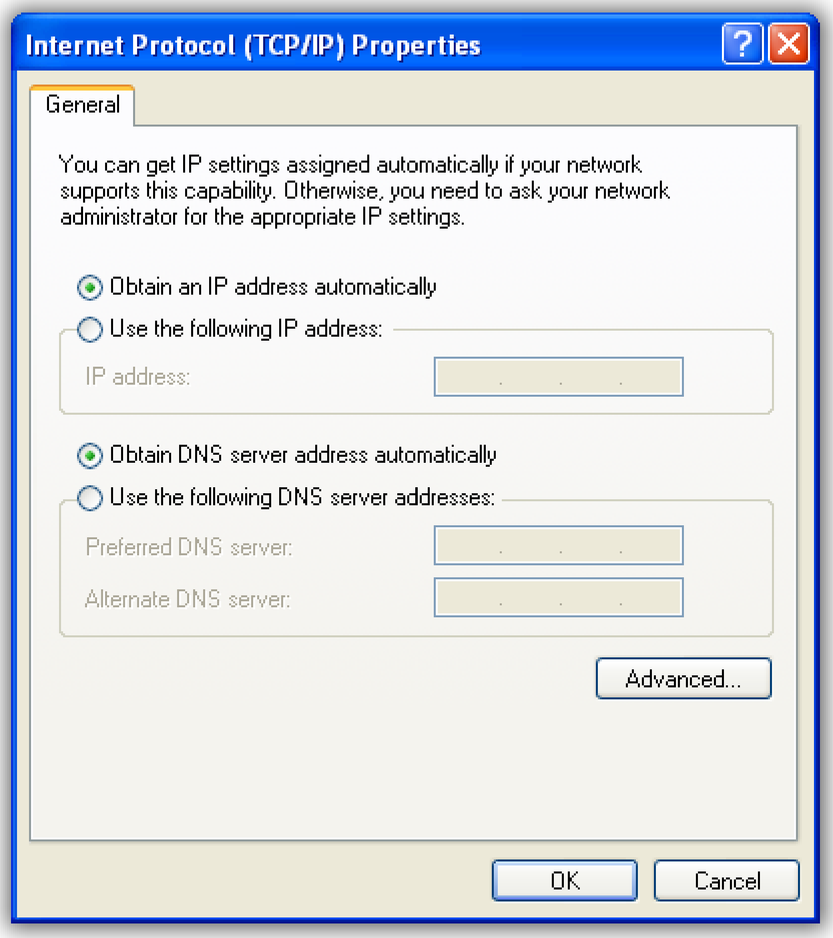
Type of dial-up server I am calling:
PPP: Windows 95/98/NT4/2000, Internet

This connection uses the following items:

- ☑ Network Monitor Driver
- ☑ Internet Protocol (TCP/IP)
- ☑ QoS Packet Scheduler
- □ File and Printer Sharing for Microsoft Networks
- ☑ Deterministic Network Enhancer

Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

2. The Internet Protocol (TCP/IP) Properties dialog appears. Select **Obtain an IP address automatically** and **Obtain DNS server address automatically** for the IP settings on this dialog.



You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

● Obtain an IP address automatically

○ Use the following IP address:

IP address: . . .

● Obtain DNS server address automatically

○ Use the following DNS server addresses:

Preferred DNS server: . . .

Alternate DNS server: . . .

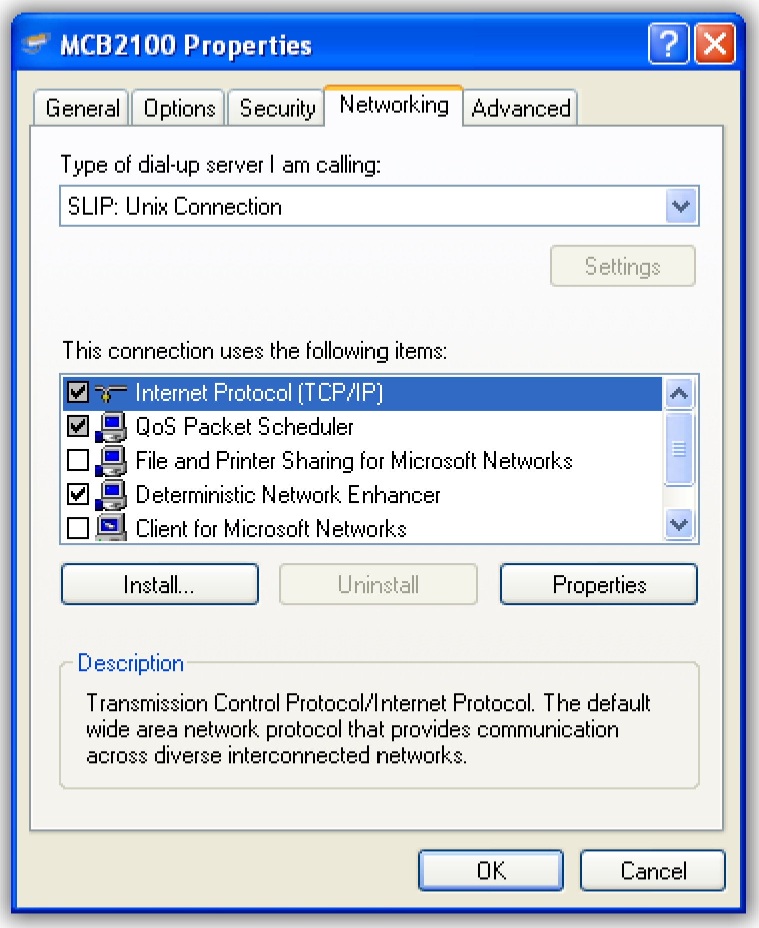
3. Click **OK** to close the options dialog. The new dial-up connection is now ready to use.

You can also provide your **own Static IP**. However, it must belong to the same PPP local network or the connection will be rejected. A typical Class C Net Mask is 255.255.255.0, which means that the first **three bytes** of your provided Static IP and the PPP Server static IP address must **match**. The **last** IP address byte must be **different**.

Configure SLIP Dial-up

Follow these steps to configure the SLIP dial-up.

1. Click the **Networking** tab in the Connection Properties dialog. Select **SLIP** from the drop-down menu for the "Type of dial-up server I am calling."
2. Enable the **Internet protocol (TCP/IP)** option from the "Components checked are used by this connection" section. Then click the **Properties** button.



MCB2100 Properties

Type of dial-up server I am calling: SLIP: Unix Connection

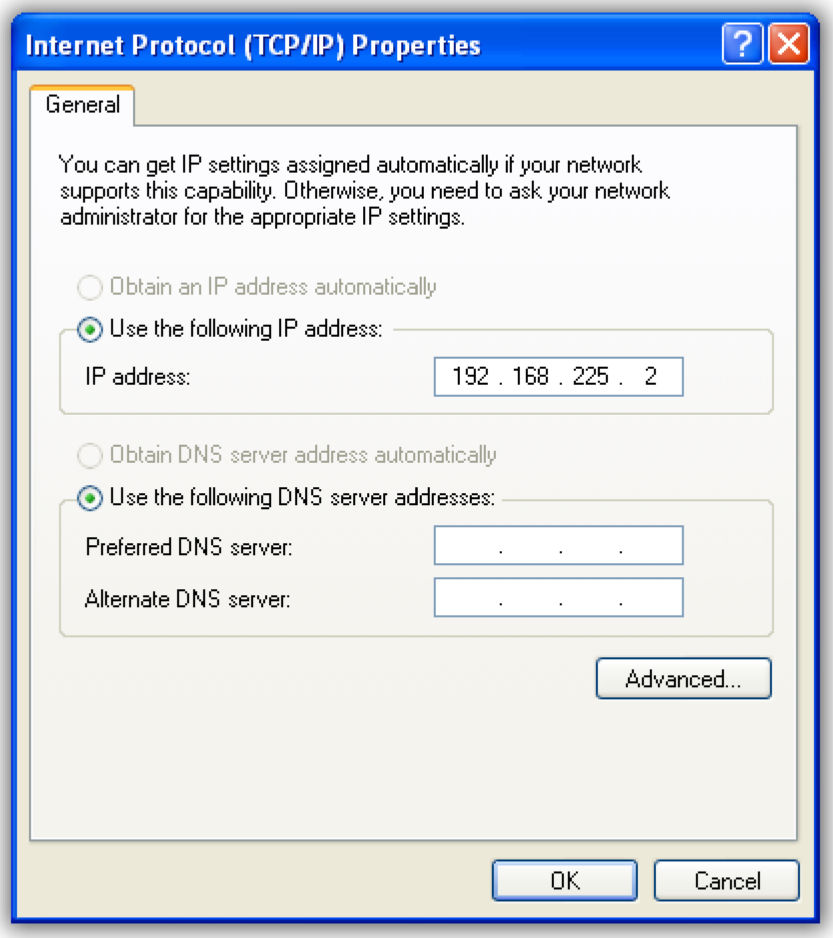
This connection uses the following items:

☑ Internet Protocol (TCP/IP)
☑ QoS Packet Scheduler
□ File and Printer Sharing for Microsoft Networks
☑ Deterministic Network Enhancer
□ Client for Microsoft Networks

Description

Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

3. On the General tab of the Internet Protocol (TCP/IP) Properties dialog, the "Use the following IP address" and "Use the following DNS server address" options are automatically selected because SLIP does not provide IP address negotiation.



Internet Protocol (TCP/IP) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

○ Obtain an IP address automatically

● Use the following IP address:

IP address: 192 . 168 . 225 . 2

○ Obtain DNS server address automatically

● Use the following DNS server addresses:

Preferred DNS server: . . .

Alternate DNS server: . . .

4. Setup the SLIP IP address of your Windows SLIP Client and click **OK** to close the options dialog. The new dial-up connection is now ready for use.

Debugging

Two versions of the RL-TCPnet Library are available.

- The **production** version is the library that is normally used. It does not include any debugging code.
- The **debug** version includes debugging messages that are output to the serial interface using the standard **printf** function. You can **redirect** the output messages to another specified device.

The RL-TCPnet library is not automatically linked with your project. You must manually include one of them.

Note

- If your target system only has one serial port available and it is used by the PPP or SLIP Network Interface, then do not enable the debug mode. This is because the debug messages will interfere with the IP packets, and the system might malfunction or crash.
- If you have a high traffic LAN and the debug mode is enabled, the system might block. If you experience unpredictable system hangs, try the application with the debug mode disabled.

Enabling Debug

Perform the following steps to enable the Debug mode:

1. **Copy** the file **Net_Debug.c** into your project folder, and add it to your project. This file is in the **\Keil\ARM\RL\TCPnet\User** folder.
2. Open the **Net_Debug.c** file in the µVision editor and **configure** the debugging.

The **Net_Debug.c** file uses the µVision Configuration Wizard to make it a simple and easy process. All the options are self explanatory.

You must add the **debug** version of the RL-TCPnet Library to your project. In this case, the **production** library is not required.

Debug Level

Several system modules output debug messages. It is possible to configure the debug output for each module separately. There are three debug levels available:

Level	Description
Off	The debug messages for the selected module are **disabled**.
Errors Only	Only **error** messages are output. This mode is useful for error tracking.
Full Debug	In this mode, **all** debug messages are output.

The system is built from several modules. The **owner module** of the displayed debug message is identified by the message prefix. The following system modules are configurable for debugging:

ID	Module	Description
MEM	Dynamic Memory Management	Allocates and releases frame buffers.
ETH	Ethernet Protocol	Handles ethernet link.
PPP	Point to Point Protocol	Handles serial line direct or modem connection PPP link.
SLIP	Serial Line Internet Protocol	Handles serial line direct or modem connection SLIP link.
ARP	Address Resolution Protocol	Handles ethernet MAC address resolution and caching.
IP	Internet Protocol	Processes the IP network layer.
ICMP	Internet Control Message Protocol	Processes ICMP messages. Best known example is the **ping**.
IGMP	Internet Group Management Protocol	Processes IGMP messages, Hosts groups and IP Multicasting.
UDP	User Datagram Protocol	Processes UDP frames.
TCP	Transmission Control Protocol	Processes TCP frames.
NBNS	NetBIOS Name Service	Maintains name access to your hardware.
DHCP	Dynamic Host Configuration Protocol	Handles automatic configuration of IP address, Net mask, Default Gateway, and Primary and Secondary DNS servers.
DNS	Domain Name Service	Handles the resolution of the IP address from a host name.
APP	Applications	This is a common debug module for all applications such as HTTP Server, Telnet Server, and TFTP Server.

An example of the debug output is:

```
ETH: *** Processing Ethernet frame ***
ETH:  Dest.MAC: 1E:30:6C:A2:45:5E
ETH:  Src. MAC: 00:11:43:A4:FE:40
ETH:  Frame len: 60 bytes
ETH:  Protocol : 0800
IP : *** Processing IP frame ***
IP :  Src. IP: 192.168.1.1
IP :  Dest.IP: 192.168.1.150
IP :  Protoc.: TCP
IP :  Id. Num: E9C1
IP :  Frm len: 40 bytes
IP :  Frame valid, IP version 4 OK
IP : Sending IP frame...
IP :  Src. IP: 192.168.1.150
IP :  Dest.IP: 192.168.1.1
IP :  Protoc.: TCP
IP :  Id. Num: 003A
IP :  Frm len: 40 bytes
ETH: Sending Ethernet frame...
ETH:  Dest.MAC: 00:11:43:A4:FE:40
ETH:  Src. MAC: 1E:30:6C:A2:45:5E
```

```
ETH:  Frame len: 54 bytes
ETH:  Protocol : 0800
```

In the above example, ethernet and IP debug messages are enabled. Received ethernet packets are processed byt the **Ethernet layer** and a debug message containing Ethernet header information is printed out. Ethernet debug information contains source and destination MAC address, ethernet frame length and ethernet protocol type.

The packet is then passed to the **IP layer**. IP layer prints out IP debug messages containing the IP header information such as source and destination IP address, frame length, protocol type etc.

Redirecting Output

Debug messages are output to a standard serial port. The **sendchar()** function outputs a single character. If required, you can customize this function to send the debug messages to some other device. In most cases, a serial UART is used to print out the debug messages.

Note

- When the **sendchar** function runs in polling mode, printing all debug messages significantly **reduces** the performance. The preferred way is to rewrite the sendchar() function to work in the interrupt mode.
- Use the **highest baud rate** possible to reduce the impact on performance from printing the debug messages.
- If the debug mode is enabled and the embedded system is connected to a **high traffic LAN** with plenty of broadcast packets, the system might malfunction.
- Printing debug messages blocks out the system task scheduler during the time when the message is being sent from the serial port. The incoming IP packets accumulate in the memory. This soon causes an **out of memory** error. Any further incoming packets are lost until some memory is released.

Function Overview

This section summarizes all the routines in the RL-TCPnet product. The functions are ordered according to the following categories:

- CGI Routines
- Ethernet Routines
- FTP Routines
- HTTP Routines
- IGMP Routines
- Miscellaneous Routines
- Modem Routines
- PPP Interface
- Serial Routines
- SLIP Interface
- SMTP Routines
- System Functions
- TCP Interface
- Telnet Routines
- TFTP Routines
- UDP Interface

The function format is same as that of the RL-RTX functions.

note

- The RL-TCPnet library does not contain all the functions that are part of the RL-TCPnet product. The Library Reference section on each function mentions whether the function is in the library or not. If a function you want to use is not in the library, you must do one of the following:
  - Include one of the provided RL-TCPnet source files that contains the function in your project. You can further customize the function.
  - Provide your own function if the RL-TCPnet source files do not contain the function you require. This is usually the case when you want to use driver functions for a different hardware.

CGI Routines

Routine	Description
cgi_func	Processes CGI script commands.
cgi_process_data	Processes data returned from an HTTP POST request.
cgi_process_var	Processes data returned from an HTTP GET request.
http_accept_host	Used for the web server access filtering.
cgx_content_type	Defines the HTML content type for **cgx** script files.

note

- The CGI routines enable you to use CGI scripts to generate dynamic HTTP pages.
- The CGI routines are not reentrant.

Ethernet Routines

Routine	Description
init_ethernet	Initializes the Ethernet controller.
int_disable_eth	Disables Ethernet controller interrupts.
int_enable_eth	Enables Ethernet controller interrupts.
interrupt_ethernet	Interrupt service routine for the Ethernet controller.
poll_ethernet	Polls the status register of the Ethernet controller.
send_frame	Sends an Ethernet frame.

note

- The Ethernet routines enable the TCPnet system to use an ethernet controller for data transfer.
- The Ethernet routines are not reentrant.

FTP Routines

Routine	Description
ftp_fclose	Closes a file that was previously opened.
ftp_fdelete	Deletes a specified file.
ftp_ffind	Lists a file directory.
ftp_fopen	Opens a file for reading or writing.
ftp_fread	Reads a block of data from a file to data buffer.
ftp_frename	Renames a file to a new name.
ftp_fwrite	Writes a block of data from buffer to a file.

note

- The FTP routines are not reentrant.

HTTP Routines

Routine	Description
http_get_info	Retrieves remote machine information.
http_get_lang	Retrieves preferred browser language settings.
http_get_session	Retrieves current session ID.
http_get_var	Retrieves HTTP environment variables.
http_get_content_type	Retrieves HTTP *Content-Type* header value.
http_fopen	Opens a file for reading.
http_flose	Closes a file that was previously opened.
http_fread	Reads a block of data from a file to data buffer.
http_fgets	Reads a string from a file to data buffer.
http_finfo	Reads a time when the file was last modified.
http_date	Converts RL date/time format to UTC format.

note

- The HTTP routines enable you to create HTTP applications and serve webpages.
- The HTTP routines are not reentrant.

IGMP Routines

Routine	Description
igmp_join	Requests that this host become a member of the Host Group identified with the "group-address".
igmp_leave	Requests that this host give up its membership in the host group identified by "group-address".

note

- The IGMP api routines enable you to use **IP Multicasting** - the transmission of an IP datagram to a "host group". A multicast datagram is delivered to all the members of its destination host group.

Miscellaneous Routines

Routine	Description
arp_cache_ip	Determines if a MAC address is in the ARP cache for the requested IP address.
dhcp_disable	Permanently disables DHCP at run-time.
get_host_by_name	Gets the IP address for a hostname.

note

- These TCPnet routines are not reentrant.

Modem Routines

Routine	Description
init_modem	Initializes the modem driver.
modem_dial	Dials the phone number of the remote modem.
modem_hangup	Disconnects the modem.
modem_listen	Sets the local modem to answer incoming calls.
modem_online	Determines if the local modem is connected.
modem_process	Processes characters that the local modem sends to RL-TCPNet.
modem_run	Executes the modem commands required to dial, listen, or disconnect.

note

- The modem routines enable you to use a modem to connect to a remote computer.
- The Modem routines are not reentrant.

PPP Routines

Routine	Description
ppp_listen	Configures the PPP interface to accept incoming connections.
ppp_connect	Starts a dial-up connection to a remote PPP server.
ppp_close	Disconnects the PPP link.
ppp_is_up	Determines whether the PPP link is established and ready to use.

note

- PPP Network Interface functions activate the PPP demon to **start** the outgoing dial-up or to **listen** to the incoming dial-in connections and to **stop** an established PPP link. The PPP Network Interface must be enabled in the configuration.
- The OS can handle simultaneous **Ethernet** and **PPP** data links. Using PPP and SLIP link at the same time is currently not supported, because both use the same serial and modem device driver.
- The PPP routines are not reentrant.

Serial Routines

Routine	Description
init_serial	Initializes the serial driver.
com_getchar	Reads a character from the serial input buffer.
com_putchar	Writes a character to the source output buffer.
com_tx_active	Determines if the serial transmitter is currently active.

note

- The TCPnet serial driver routines are not reentrant.

SLIP Routines

Routine	Description
slip_listen	Configures the SLIP interface to accept incoming connections.
slip_connect	Starts a dial-up connection to a remote SLIP server.
slip_close	Disconnects the SLIP link.
slip_is_up	Determines whether the SLIP link is established and ready to use.

note

- SLIP Network Interface routines activate the SLIP demon to **start** the outgoing dial-up or to **listen** to the incoming dial-in connections and to **stop** an established SLIP link. The SLIP Network Interface must be enabled in the configuration.
- The OS can handle simultaneous **Ethernet** and **SLIP** data links. Using PPP and SLIP link at the same time is currently not supported, because both use the same serial and modem device driver.
- The SLIP routines are not reentrant.

SMTP Routines

Routine	Description
smtp_cbfunc	SMTP call-back function.
smtp_connect	Starts the SMTP client.
smtp_accept_auth	Enables the SMTP authentication.

note

- The SMTP routines are not reentrant.

SNMP Routines

Routine	Description
snmp_trap	Sends SNMP trap message.
snmp_set_community	Changes the SNMP community.

note

- The SNMP routines are not reentrant.

System Functions

Routine	Description
init_TcpNet	Initializes RL-TCPnet system resources, protocols, and applications.
main_TcpNet	Processes RL-TCPnet operations including protocol timeouts, ARP address cache, and Ethernet controller polling.
timer_tick	Generates periodic events for RL-TCPnet.

note

- The TCPnet system functions are the core of the protocol stack. They form a operating system which calls all other protocol module functions.
- The TCPnet system functions do not require RTOS to run. However, it can run with various forms of RTOS if required.
- The TCPnet system functions are not reentrant.

TCP Routines

Routine	Description
tcp_get_socket	Allocates a TCP socket.
tcp_connect	Initiates a TCP connection.
tcp_listen	Opens a TCP socket for listening.
tcp_close	Closes a TCP socket.
tcp_abort	Closes a TCP socket immediately.
tcp_release_socket	Releases (deallocates) a TCP socket.
tcp_get_buf	Allocates memory for a TCP send buffer.
tcp_max_dsize	Changes the TCP maximum segment size.
tcp_send	Sends a TCP packet.
tcp_get_state	Retrieves the current state of the TCP socket.
tcp_check_send	Determines if a TCP socket is ready to send data.
tcp_reset_window	Resets the TCP window to maximum size.

note

- TCP Interface functions exchange data over **TCP Socket**. It is used when data security is a primary option. TCP packets require acknowledgement at the protocol level. Eventually, any lost packets are retransmitted.
- The TCP routines are not reentrant.

Telnet Routines

Routine	Description
tnet_cbfunc	TELNET call-back function.
tnet_ccmp	Compares the TELNET buffer to a command string.
tnet_get_info	Retrieves information about the remote host connected to the TELNET server.
tnet_process_cmd	Processes and executes a TELNET command.
tnet_set_delay	Sets the time delay used between TELNET command processing.
tnet_msg_poll	Polls the upper-layer application for Unsolicited messages.

note

- The telnet routines are not reentrant.

TFTP Routines

Routine	Description
tftp_fclose	Closes a TFTP file.
tftp_fopen	Opens a TFTP file.
tftp_fread	Reads a block of data from a TFTP file.
tftp_fwrite	Writes a block of data to a TFTP file.

note

- The TFTP routines are not reentrant.

UDP Routines

Routine	Description
udp_get_socket	Allocates a UDP socket.
udp_open	Opens a UDP socket for communication.
udp_close	Closes a UDP socket.
udp_release_socket	Releases (deallocates) a UDP socket.
udp_get_buf	Allocates memory for a UDP send buffer.
udp_send	Sends a UDP packet.
udp_mcast_ttl	Sets the Time to Live (TTL) for the outgoing multicast messages of a socket.

note

- UDP Interface routines exchange data over the **UDP Socket**. It is used when data security is not the primary option, because UDP packets do not require an acknowledge and can possibly be lost.
- The UDP routines are not reentrant.

RL-CAN

The RL-CAN Real-Time Library CAN Driver is a group of library routines that enable CAN communications on a variety of microcontrollers. RL-CAN gives your programs access to the on-chip CAN controller found on these devices.

A **CAN** (Controller Area Network) is a high speed (up to 1 Mbit) serial bus with message priority and error checking. Originally developed for use in automobiles, it is suitable for numerous other applications like factory automation.

Nodes on a CAN network are usually connected by a differential twisted wire pair physical interface.

Note

- **RL-CAN** is not included with the RealView® MDK-ARM™ Microcontroller Development Kit. It is available in the stand-alone product **RL-ARM**™, which also contains the RTX kernel (source code included), Flash File System, TCP/IP Stack, and USB drivers.

Overview

The RL-CAN for the **RTX Kernel** simplifies the implementation of Controller Area Network (CAN) applications. The RL-CAN includes:

- A common generic software layer.
- A hardware dependent software layer that implements the physical interface with the CAN peripheral.

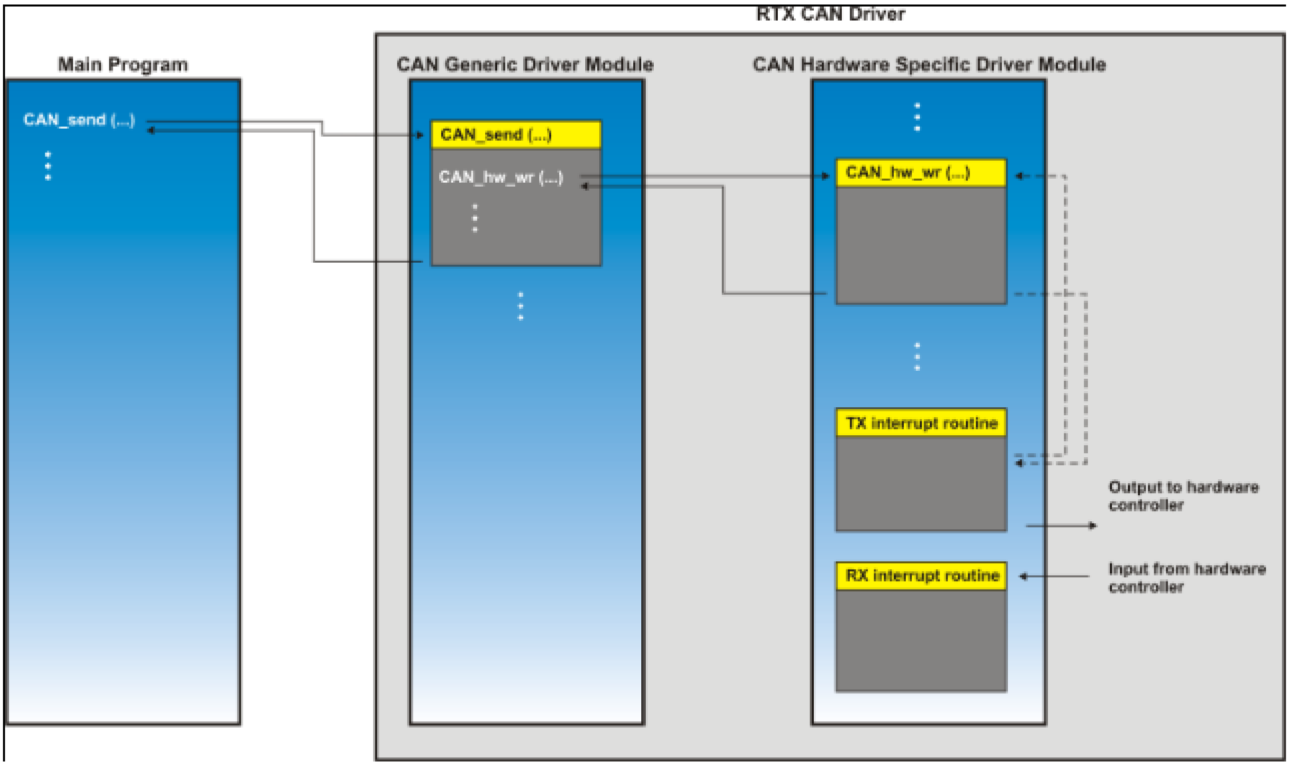
The RL-CAN runs interrupt service routines using RTX Kernel functions for **Mailbox Management** and **Memory Allocation**. The RL-CAN uses one **Memory Pool** for all CAN messages. Each CAN controller has two mailbox arrays, one to receive and one to transmit message buffering. The buffer method is First In First Out (FIFO) based.

The interrupt-based RL-CAN provides the user with an available mechanism for message transmission and reception.

The RL-CAN common generic software layer functions are located in the RTX_CAN.c file. This layer allows users to apply the same interface across different targets and to easily switch from one target to another without changing the Main Program.

The RL-CAN hardware dependent software layer functions are located in the CAN_chip.c (example: for NXP LPC23xx chip file name is CAN_LPC23xx.c) file. The hardware dependent software layer enables the generic portion to function on many different targets, with each target having its own hardware dependent software layer implementation.

The RL-CAN function diagram below shows how the Main Program's functions traverse the generic layer to the hardware layer where the functions are then executed. Input from the hardware controller follows a reverse route back to the Main Program.



See the article in Wikipedia for more information about CAN.

Features

The RL-CAN Real-Time Library CAN Driver offers a number of benefits and features that help you create applications with CAN support.

- Support for 11-bit and 29-bit message IDs.
- Support for Data Frames and Remote Frames.
- Easy-to-use library routines to send and receive CAN messages.
- Pre-configured drivers for the most popular ARM® devices.

Source Files

Source files for the Real-Time Library CAN Driver are found in the **\KEIL\ARM\Boards\<vendor>\<board>\RL\CAN\** folders. Various architectures are supported.

- **\KEIL\ARM\Boards\Atmel\AT91SAM7X-EK\RL\CAN\**
  This folder contains CAN projects for the Atmel AT91SAM7X Evaluation Board for the Atmel AT91SAM7X device family.
- **\KEIL\ARM\Boards\Keil\MCB1700\RL\CAN\**
  This folder contains CAN projects for the Keil MCB1700 Evaluation Board for the NXP LPC17xx device family.
- **\KEIL\ARM\Boards\Keil\MCB2100\RL\CAN\**
  This folder contains CAN projects for the Keil MCB2100 Evaluation Board for the NXP LPC21xx device family.
- **\KEIL\ARM\Boards\Keil\MCB2300\RL\CAN\**
  This folder contains CAN projects for the Keil MCB2300 Evaluation Board for the NXP LPC23xx device family.
- **\KEIL\ARM\Boards\Keil\MCB2400\RL\CAN\**
  This folder contains CAN projects for the Keil MCB2400 Evaluation Board for the NXP LPC24xx device family.
- **\KEIL\ARM\Boards\Keil\MCB2900\RL\CAN\**
  This folder contains CAN projects for the Keil MCB2900 Evaluation Board for the NXP LPC29xx device family.
- **\KEIL\ARM\Boards\Keil\MCB2929\RL\CAN\**
  This folder contains CAN projects for the Keil MCB2929 Evaluation Board for the NXP LPC29xx device family.
- **\KEIL\ARM\Boards\Keil\MCBSTM32\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTM32 Evaluation Board for the ST Microelectronics STM32F103 device family.
- **\KEIL\ARM\Boards\Keil\MCBSTM32C\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTM32C Evaluation Board for the ST Microelectronics STM32F105/7 device family.
- **\KEIL\ARM\Boards\Keil\MCBSTM32E\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTM32E Evaluation Board for the ST Microelectronics STM32F103 device family.
- **\KEIL\ARM\Boards\Keil\MCBSTR7\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTR7 Evaluation Board for the ST Microelectronics STR71x device family.
- **\KEIL\ARM\Boards\Keil\MCBSTR730\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTR730 Evaluation Board for the ST Microelectronics STR73x device family.
- **\KEIL\ARM\Boards\Keil\MCBSTR9\RL\CAN\**
  This folder contains CAN projects for the Keil MCBSTR9 Evaluation Board for the ST Microelectronics STR91x device family.
- **\KEIL\ARM\Boards\Luminary\EK-LM3S2110\RL\CAN\**
  This folder contains CAN projects for the Luminary EK-LM3S2110 Evaluation Board for the Luminary LM3Sxxxx device family.
- **\KEIL\ARM\Boards\Luminary\EK-LM3S8962\RL\CAN\**
  This folder contains CAN projects for the Luminary EK-LM3S8962 Evaluation Board for the Luminary LM3Sxxxx device family.
- **\KEIL\ARM\Boards\Phytec\LPC229x\RL\CAN\**
  This folder contains CAN projects for the Phytec LPC229x Evaluation Board for the NXP LPC229x device family.

The following table identifies the file names and contents. The include and source files described contain constant definitions and program code that is specific to each target hardware and microcontroller.

Filename	Description
RTX_CAN.H	This include file defines the functions and constants related to the CAN driver. This header file should be included in all source files that use CAN Driver routines. It is located in **Keil\ARM\RV31\INC** directory.
RTX_CAN.C	This source file is the common generic CAN Driver file that interfaces to the hardware layer via routines implemented in **CAN_chip.C**. It is located in **Keil\ARM\RL\CAN\SRC** directory.
CAN_chip.C	This source file contains the hardware-level driver routines. "chip" stands for name of device series, for example for NXP LPC23xx device series file name is CAN_LPC23xx.C . All files are located in **Keil\ARM\RL\CAN\Drivers** directory.
CAN_CFG.H	This include file is the configuration file that enables users to easily change

	parameters for the hardware-level driver. You may edit this file using the Configuration Wizard in µVision®.
CAN_REG.H	This include file contains type definitions that are used by the hardware-level driver routines. This include file is only used for the NXP LPC2xxx device family.

Note

- To use the CAN Driver in your own project, copy the files listed above except RTX_CAN.H and RTX_CAN.C from the appropriate folder into your project folder. Then, configure **CAN_CFG.H** to meet the requirements of your CAN application.

Function Overview

This section summarizes all the routines in the RL-CAN library. The functions are ordered according to the following categories:

- Initialization Routines
- Message Reception Routines
- Message Transmission Routines

The function format is same as that of the RL-RTX functions.

Initialization Routines

Routine	Attributes	Description
CAN_init		Initializes the CAN controller hardware, CAN driver resources, and CAN bus baudrate.
CAN_start		Starts the specified CAN controller and enables CAN bus communication.

note

- The CAN initialization routines enable you to initialize a CAN controller and start the communication.
- The CAN initialization routines are not reentrant.

Message Reception Routines

Routine	Attributes	Description
CAN_rx_object		Configures a CAN message receive object.
CAN_receive		Receives a CAN message.

note

- The CAN message reception routines enable you to configure and receive data using a CAN controller.
- The CAN message reception routines are not reentrant.

Message Transmission Routines

Routine	Attributes	Description
CAN_tx_object		Configures a CAN message transmit object.
CAN_send		Transmits a CAN DATA FRAME message.
CAN_request		Transmits a CAN REMOTE FRAME request and possibly receives a DATA FRAME response.
CAN_set		Sets the CAN DATA FRAME to send in response to a REMOTE FRAME request.

note

- The CAN message transmission routines enable you to configure, send and request data using a CAN controller.
- The CAN message transmission routines are not reentrant.

Errors

The RL-CAN library routines return one of the following error constants to indicate status.

- **CAN_OK**
  Indicates the function completed successfully without error.
- **CAN_NOT_IMPLEMENTED_ERROR**
  Indicates the function is not implemented.
- **CAN_MEM_POOL_INIT_ERROR**
  Indicates that the memory pool used for software message buffers did not initialize successfully.
- **CAN_BAUDRATE_ERROR**
  Indicates that the communication speed was incorrectly initialized.
- **CAN_TX_BUSY_ERROR**
  Indicates that the transmit hardware is busy.
- **CAN_OBJECTS_FULL_ERROR**
  Indicates that no more transmit or receive objects may be defined.
- **CAN_ALLOC_MEM_ERROR**
  Indicates there is no available memory in the CAN memory pool.
- **CAN_DEALLOC_MEM_ERROR**
  Indicates that the memory used by the transmitted or received message was not correctly deallocated.
- **CAN_TIMEOUT_ERROR**
  Indicates that the timeout expired before a message was transmitted.
- **CAN_UNEXIST_CTRL_ERROR**
  Indicates that the requested CAN controller does not exist.
- **CAN_UNEXIST_CH_ERROR**
  Indicates that the requested CAN channel does not exist.

If the function completes successfully and without error, **CAN_OK** is returned.

Note

- CAN error codes are defined as an enumerated CAN_ERROR type in **RTX_CAN.h**.

Hardware Configuration

The RL-CAN must be configured for the embedded applications you create. All configuration settings are found in the **CAN_Cfg.h** file. Configuration options allow you to:

- Specify which of the hardware available CAN controllers to use
- Specify the size of the transmit software FIFO buffer for each CAN controller
- Specify the size of the receive software FIFO buffer for each CAN controller

You need to add a CAN_Cfg.h configuration file to each project when you want to use the RL-CAN.

To customize the RL-CAN configuration, you must change the settings specified in CAN_Cfg.h file.

NXP LPC17xx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC17xx.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate).

```
#define PCLK             25000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

```
#define USE_CAN_CTRL1           1
#define USE_CAN_CTRL2           1
```

- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

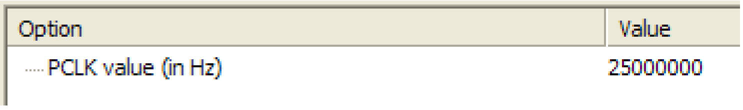
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

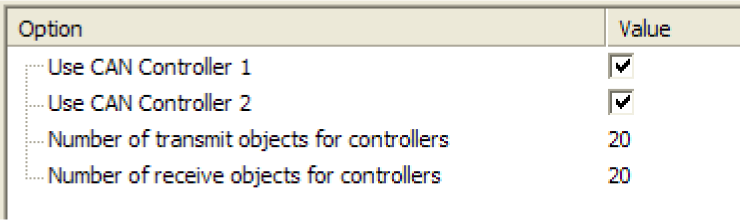
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC17xx.c** edited with Configuration Wizard:



Option	Value
PCLK value (in Hz)	25000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

NXP LPC21xx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC21xx.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate). The value of this constant must be calculated manually and is dependent on VPBDIV and MSEL settings in the Startup file. The value is equal to VPB Clock.

```
#define  PCLK            60000000    /* VPB Clock = CPU Clock,
                                        MSEL = 5,
                                        xtal = 12 MHz,
                                        CPU Clock = 60 MHz (MSEL * xtal)
*/
```

- **VIC_NUM_CTRL1_TX .. VIC_NUM_CTRL4_TX** specifies which vectored interrupt number to use for message transmission.

  Since transmission is interrupt driven, it is necessary to define unique numbers of interrupts used for each controller's transmission. Acceptable values are 0 .. 15. The lower values have a higher priority.

```
#define VIC_NUM_CTRL1_TX       11
```

- **VIC_NUM_CTRL1_RX .. VIC_NUM_CTRL4_RX** specifies which vectored interrupt number is going to be used for message reception.

  Since reception is interrupt driven, it is necessary to define unique numbers of interrupts used for each controller's reception. Acceptable values are 0 .. 15. The lower values have a higher priority.

```
#define VIC_NUM_CTRL1_RX        7
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL4** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

```
#define USE_CAN_CTRL1           1
#define USE_CAN_CTRL2           1
#define USE_CAN_CTRL3           0
#define USE_CAN_CTRL4           0
```

- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

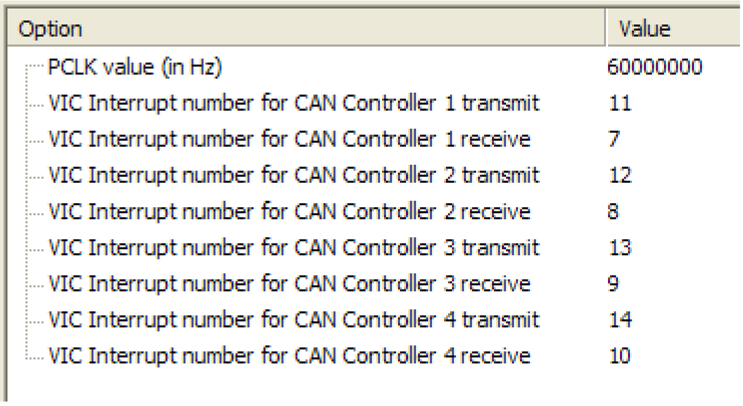
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

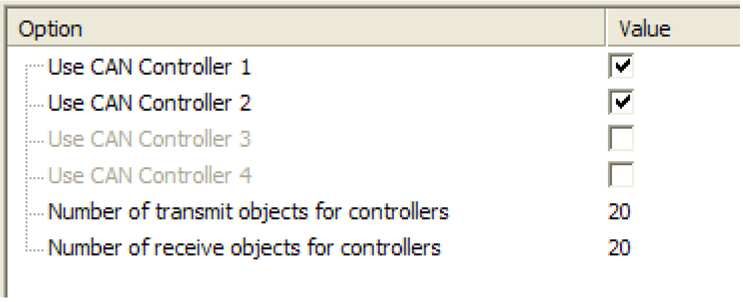
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC21xx.c** edited with Configuration Wizard:



Option	Value
PCLK value (in Hz)	60000000
VIC Interrupt number for CAN Controller 1 transmit	11
VIC Interrupt number for CAN Controller 1 receive	7
VIC Interrupt number for CAN Controller 2 transmit	12
VIC Interrupt number for CAN Controller 2 receive	8
VIC Interrupt number for CAN Controller 3 transmit	13
VIC Interrupt number for CAN Controller 3 receive	9
VIC Interrupt number for CAN Controller 4 transmit	14
VIC Interrupt number for CAN Controller 4 receive	10

File **CAN_Cfg.h** edited with Configuration Wizard:



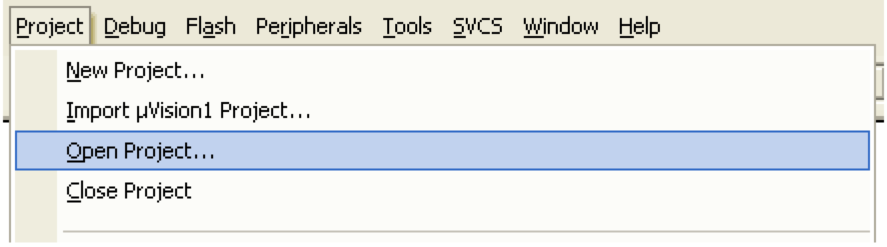
Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Use CAN Controller 3	□
Use CAN Controller 4	□
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

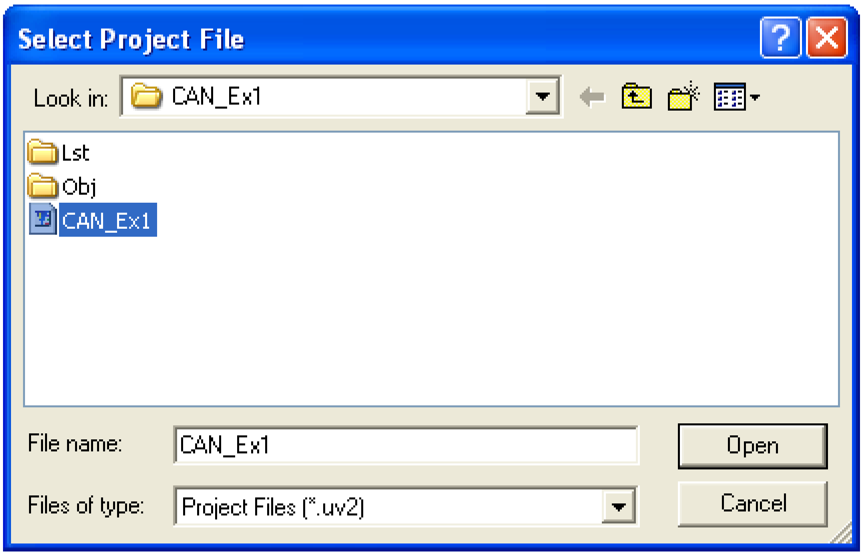
Getting Started

To run this RL-CAN LPC21xx example on a board, ensure you have:

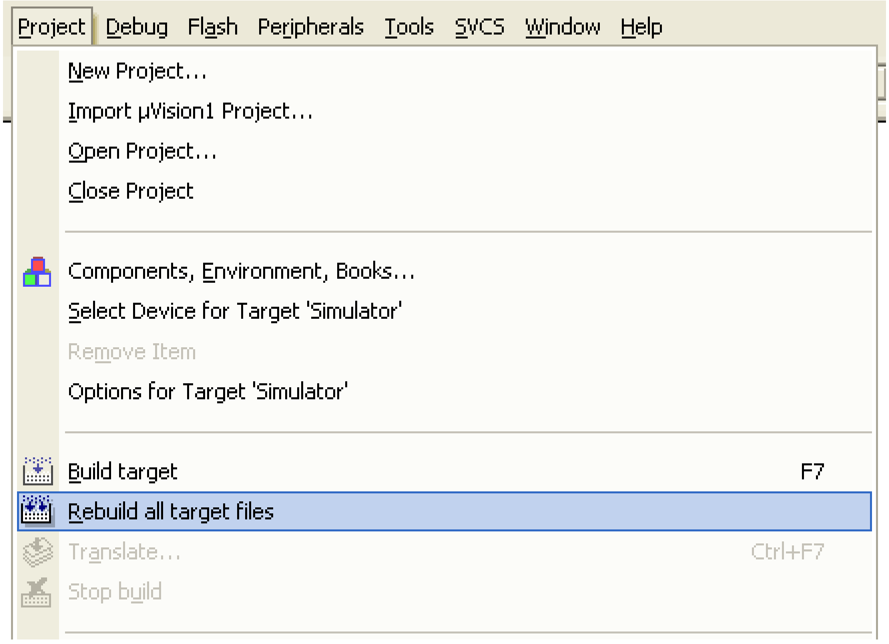
- an MCB2100 Evaluation board from Keil
- a ULINK® USB Interface Adapter from Keil
- a 5V power supply
- a loopback cable (DB-9 female to DB-9 female).

1. Copy the example from **\Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 3 IDE (**Project — Open Project...**)

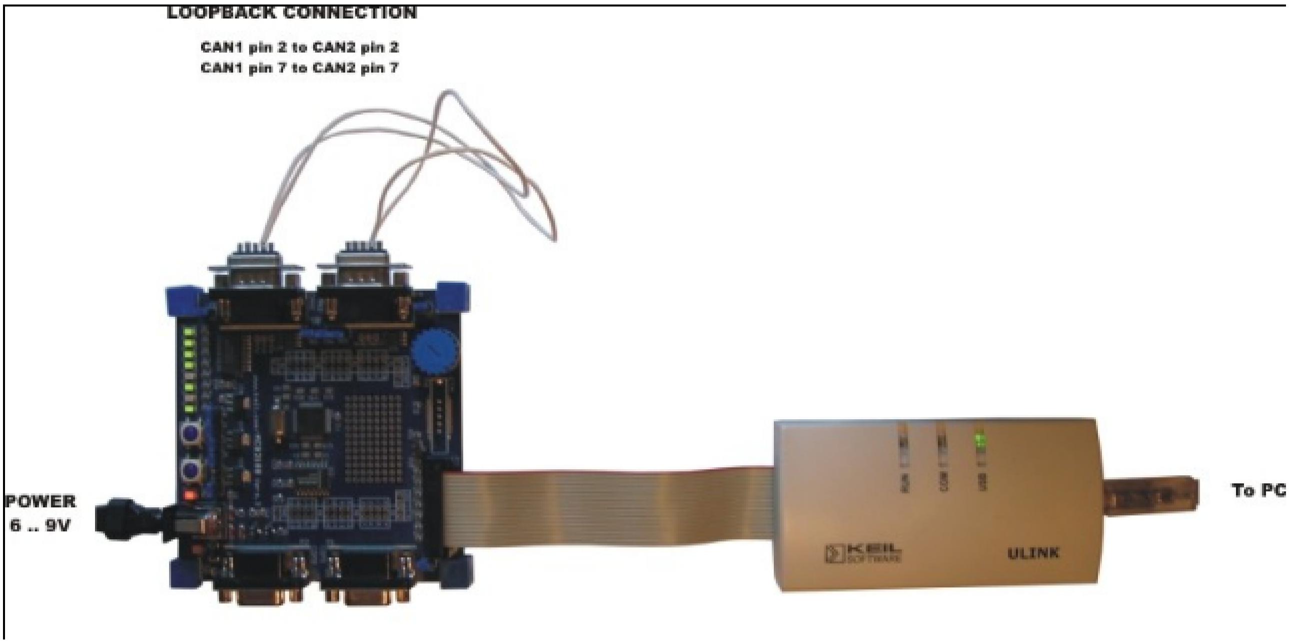




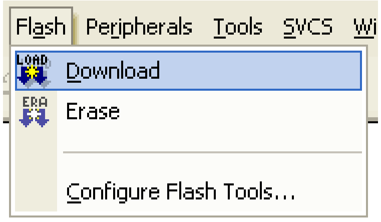
3. Rebuild the executable file from source files. Click on **Project — Rebuild all target files** on the menu or click on the toolbar button.



4. Connect the ULINK device to the PC's USB port and to the JTAG connector on the MCB2100 board.
   Connect a power cord to the barrel plug port on the MCB2100 board.



5. Make a loopback connection between the CAN1 and CAN2 ports. Connect the CAN1 pin 2 to the CAN2 pin 2 and the CAN1 pin 7 to the CAN2 pin 7.
6. Power-up the MCB2100 board.
7. Click on **Flash — Download** on the menu to download the executable file to the target LPC2100 Flash on the MCB2100 board.



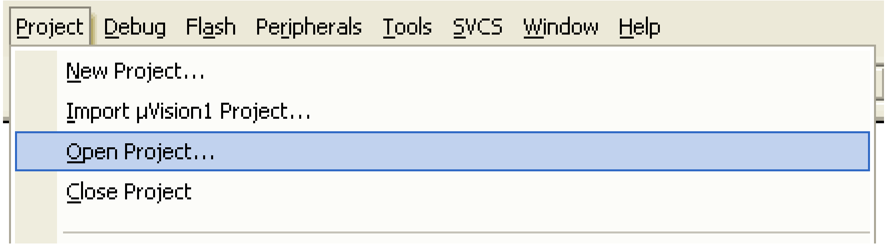
8. Rotate the potentiometer and watch the LEDs change state and flash speed according to the position of the potentiometer.

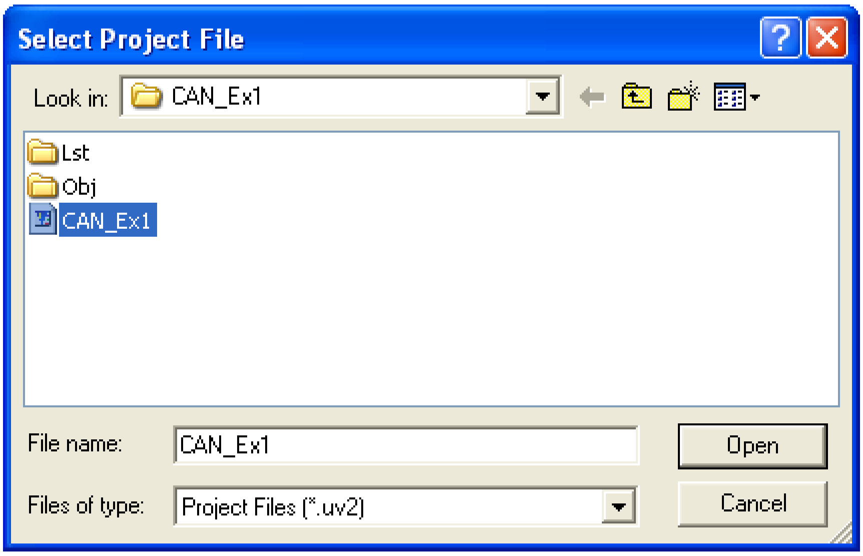
Simulation

Programs you create with the RL-CAN Real-Time Library CAN Driver may be tested using the simulation capabilities of the µVision® IDE. Simulation allows you to test your CAN application before target hardware is ready.

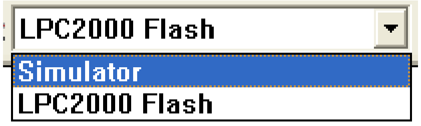
Follow these steps to test an RL-CAN example application using the µVision Simulator.

1. Copy the example from **\Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex2** directory to any directory you want to use.
2. Select and load the project file CAN_Ex1.uv2 or CAN_Ex2.uv2 into µVision . This file is located in the folder copied in step 1. In µVision, use **Project — Open Project...**

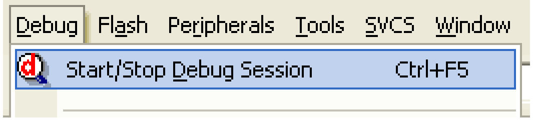

Project Debug Flash Peripherals Tools SVCS Window Help
New Project...
Import µVision1 Project...
Open Project...
Close Project



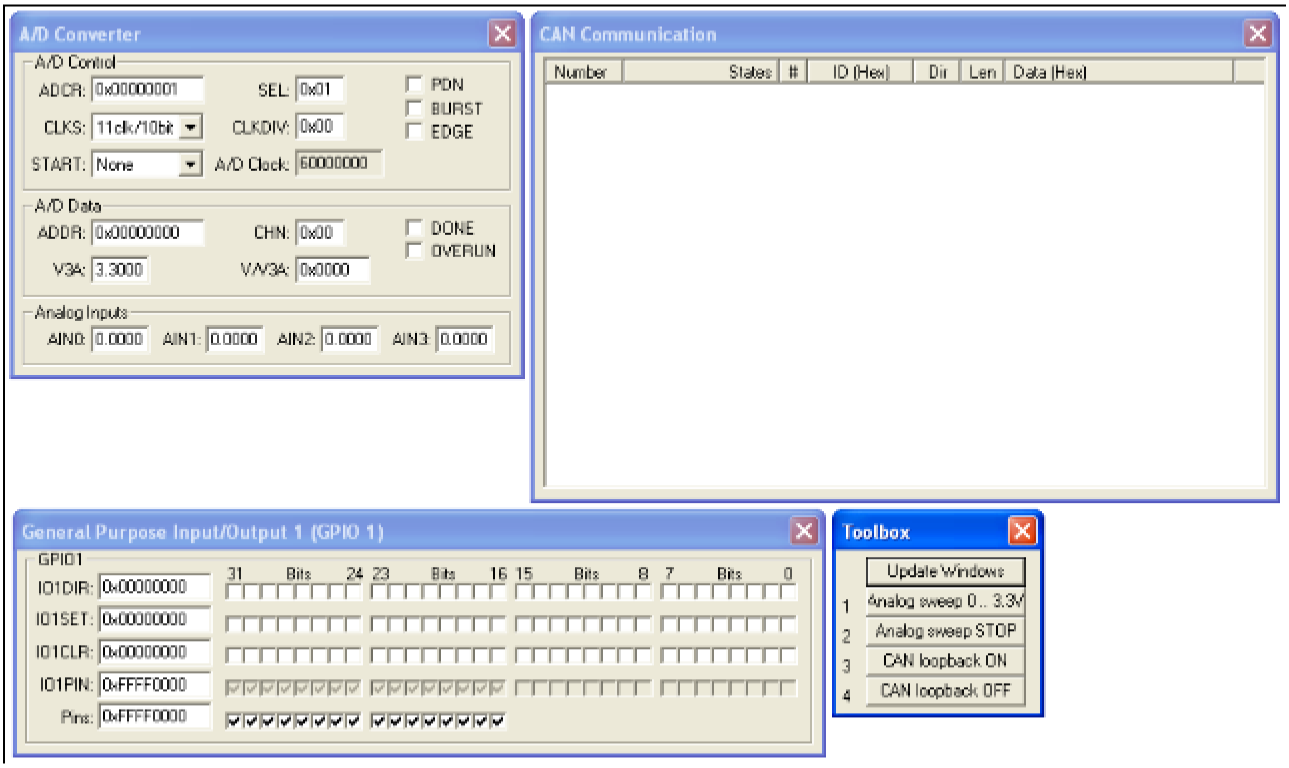
3. Select the target **Simulator**.



4. Click on **Debug — Start/Stop Debug Session** on the menu bar or click the toolbar button to run the simulation.


Debug Flash Peripherals Tools SVCS Window
Start/Stop Debug Session Ctrl+F5

5. New windows appear as shown below. These windows show the device peripheral functionality.



ADCR: 0x00000001 SEL: 0x01 □ PDN
CLKS: 11clk/10bit CLKDIV: 0x00 □ BURST
START: None A/D Clock: 60000000 □ EDGE

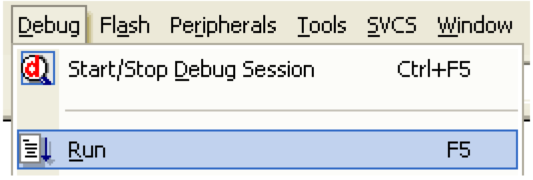
ADDR: 0x00000000 CHN: 0x00 □ DONE
V3A: 3.3000 VVV3A: 0x0000 □ OVERUN

AIN0: 0.0000 AIN1: 0.0000 AIN2: 0.0000 AIN3: 0.0000

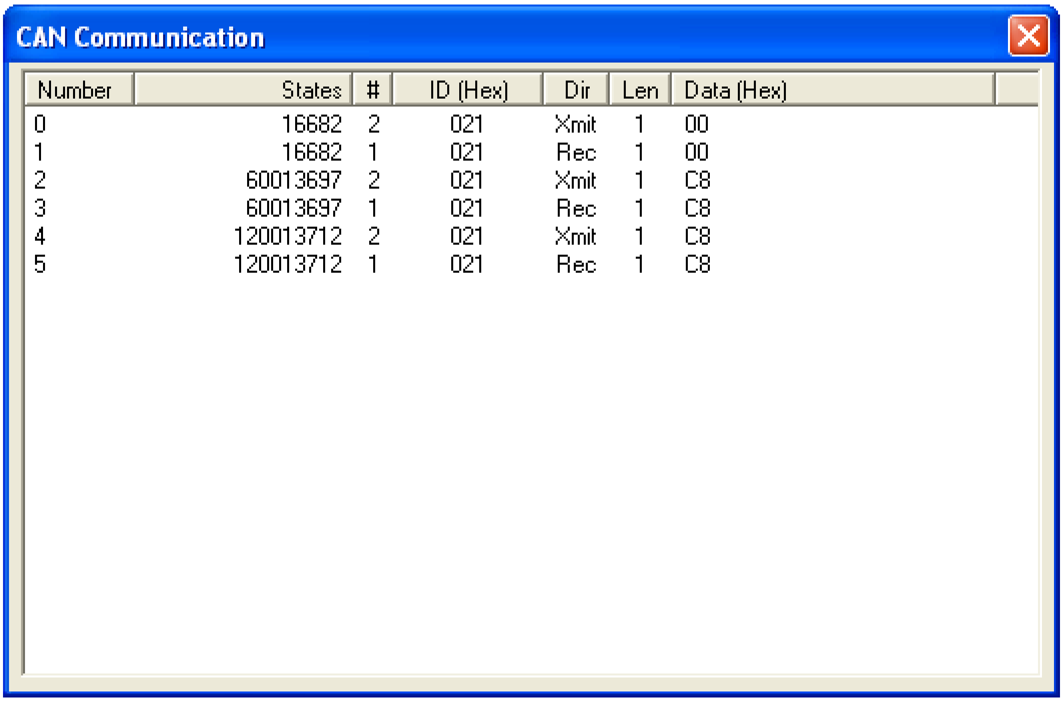
Number	States	#	ID (Hex)	Dir	Len	Data (Hex)

IO1DIR: 0x00000000
IO1SET: 0x00000000
IO1CLR: 0x00000000
IO1PIN: 0xFFFF0000
Pins: 0xFFFF0000

6. Click on **Debug — Run** to start the simulation.

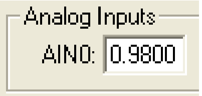


7. The communication messages display in the **CAN Communication** window.



Number	States	#	ID (Hex)	Dir	Len	Data (Hex)
0	16682	2	021	Xmit	1	00
1	16682	1	021	Rec	1	00
2	60013697	2	021	Xmit	1	C8
3	60013697	1	021	Rec	1	C8
4	120013712	2	021	Xmit	1	C8
5	120013712	1	021	Rec	1	C8

8. Click on the **Analog sweep 0 .. 3.3V** button, on the **Toolbox** window, to start the analog value changing by rising from 0 V to 3.3 V and lowering from 3.3 V to 0 V. You can stop the analog value change by clicking on **Analog sweep STOP**.



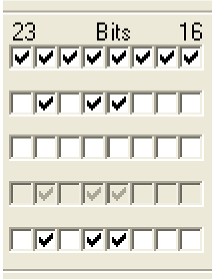
Analog Inputs

AIN0: 0.9800

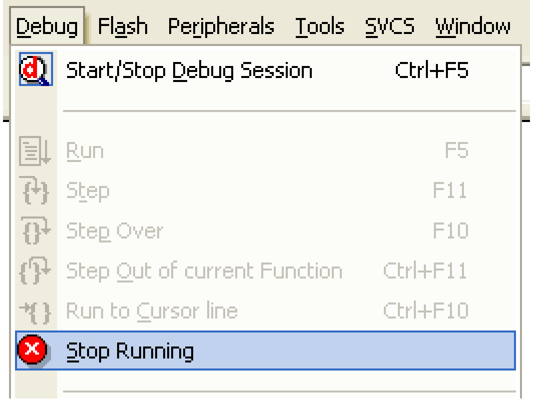
9. Click on the button **CAN loopback ON**, on the Toolbox window, to enable simulation of the hardware connection (loopback) between the CAN 1 and CAN 2 ports.

   Click on **CAN loopback OFF** to turn off the simulation of the loopback connection between the CAN ports.

10. After enabling the loopback in the previous step, you can watch changes in the **General Purpose Input/Output 1 (GPIO 1)** window on **bits 23 .. 16**, according to analog input value.



11. Stop the simulation before exiting µVision by clicking on **Debug — Stop** on the Debug menu.


Debug
Flash
Peripherals
Tools
SVCS
Window
Start/Stop Debug Session Ctrl+F5
Run F5
Step F11
Step Over F10
Step Out of current Function Ctrl+F11
Run to Cursor line Ctrl+F10
Stop Running

NXP LPC229x Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC229x.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate). The value of this constant must be calculated manually and is dependent on VPBDIV and MSEL settings in the Startup file. The value is equal to VPB Clock.

```
#define  PCLK            60000000    /* VPB Clock = CPU Clock,
                                        MSEL = 5,
                                        xtal = 12 MHz,
                                        CPU Clock = 60 MHz (MSEL * xtal)
*/
```

- **VIC_NUM_CTRL1_TX .. VIC_NUM_CTRL4_TX** specifies which vectored interrupt number to use for message transmission.

  Since transmission is interrupt driven, it is necessary to define unique numbers of interrupts used for each controller's transmission. Acceptable values are 0 .. 15. The lower values have a higher priority.

```
#define VIC_NUM_CTRL1_TX       11
```

- **VIC_NUM_CTRL1_RX .. VIC_NUM_CTRL4_RX** specifies which vectored interrupt number is going to be used for message reception.

  Since reception is interrupt driven, it is necessary to define unique numbers of interrupts used for each controller's reception. Acceptable values are 0 .. 15. The lower values have a higher priority.

```
#define VIC_NUM_CTRL1_RX        7
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL4** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

```
#define USE_CAN_CTRL1           1
#define USE_CAN_CTRL2           1
#define USE_CAN_CTRL3           0
#define USE_CAN_CTRL4           0
```

- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

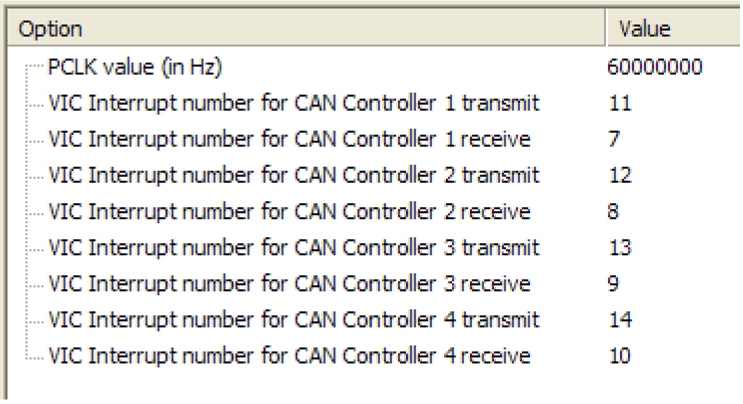
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

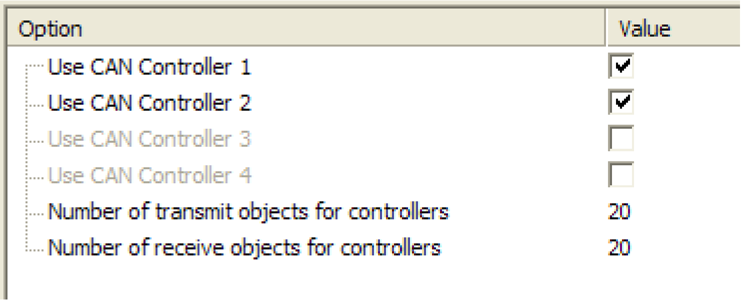
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC229x.c** edited with Configuration Wizard:



Option	Value
PCLK value (in Hz)	60000000
VIC Interrupt number for CAN Controller 1 transmit	11
VIC Interrupt number for CAN Controller 1 receive	7
VIC Interrupt number for CAN Controller 2 transmit	12
VIC Interrupt number for CAN Controller 2 receive	8
VIC Interrupt number for CAN Controller 3 transmit	13
VIC Interrupt number for CAN Controller 3 receive	9
VIC Interrupt number for CAN Controller 4 transmit	14
VIC Interrupt number for CAN Controller 4 receive	10

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Use CAN Controller 3	□
Use CAN Controller 4	□
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

NXP LPC23xx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC23xx.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate).

```
#define PCLK             24000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1          1
  #define USE_CAN_CTRL2          1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

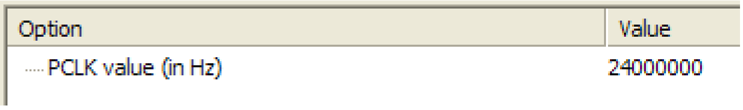
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

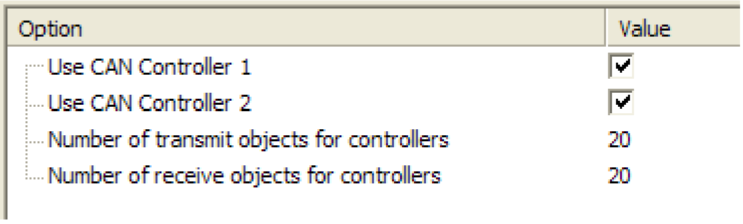
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC23xx.c** edited with Configuration Wizard:



Option	Value
PCLK value (in Hz)	24000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

NXP LPC24xx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC24xx.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate).
-

-
  ```
  #define PCLK              24000000
  ```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1           1
  #define USE_CAN_CTRL2           1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.
-

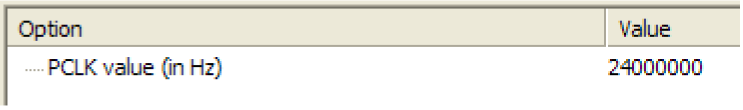
-
  ```
  #define CAN_No_SendObjects     20
  ```
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.
-

-
  ```
  #define CAN_No_ReceiveObjects  20
  ```

Using Configuration Wizard

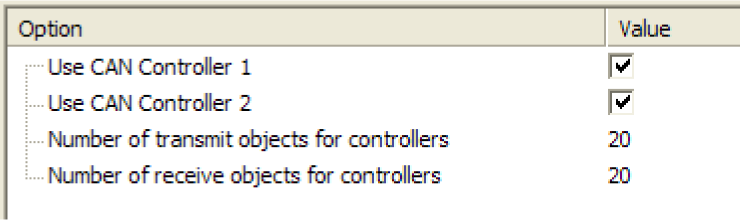
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC24xx.c** edited with Configuration Wizard:



Option	Value
PCLK value (in Hz)	24000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

NXP LPC29xx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LPC29xx.c** file:

- **PCLK** (**P**eripheral **CL**oc**K**) constant is used to calculate the correct communication speed (baudrate).

```
#define PCLK                96000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1            1
  #define USE_CAN_CTRL2            1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

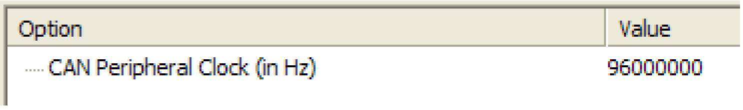
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

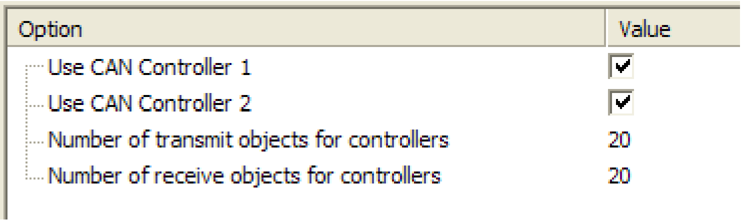
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LPC29xx.c** edited with Configuration Wizard:



Option	Value
CAN Peripheral Clock (in Hz)	96000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

ST STM32F103 Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_STM32F103.c** file:

- **CAN_CLK** (Peripheral Clock) constant is used to calculate the correct communication speed (baudrate).

```
#define CAN_CLK               36000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1          1
  #define USE_CAN_CTRL2          0
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

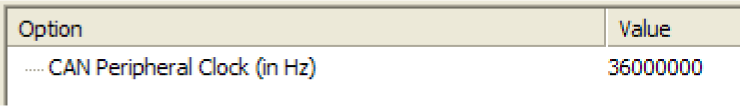
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

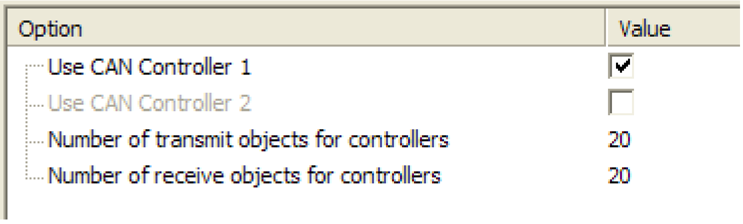
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_STM32F103.c** edited with Configuration Wizard:



Option	Value
CAN Peripheral Clock (in Hz)	36000000

File **CAN_Cfg.h** edited with Configuration Wizard:



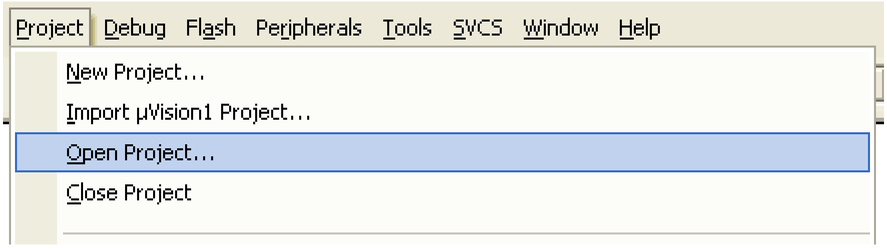
Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	□
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

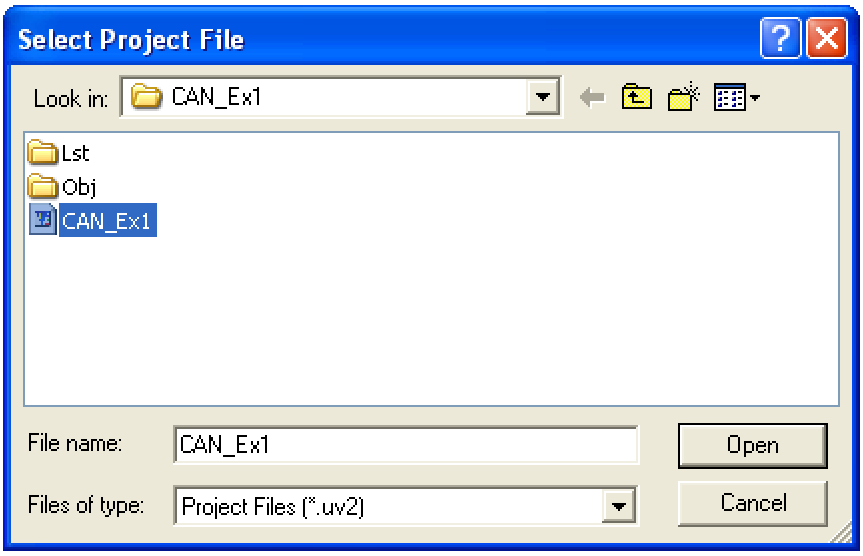
Getting Started

To run this RL-CAN STM32F103 example on a board, ensure you have:

- a MCBSTM32 Evaluation board from Keil
- a ULINK2® USB Interface Adapter from Keil
- a USB cable for 5 V power supply.

1. Copy the example from **\Keil\ARM\Boards\Keil\MCBSTM32\RL\CAN\CAN_Ex1** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 file from the copied directory into µVision® 3 IDE (**Project —> Open Project...**)



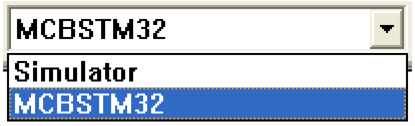


Look in: CAN_Ex1

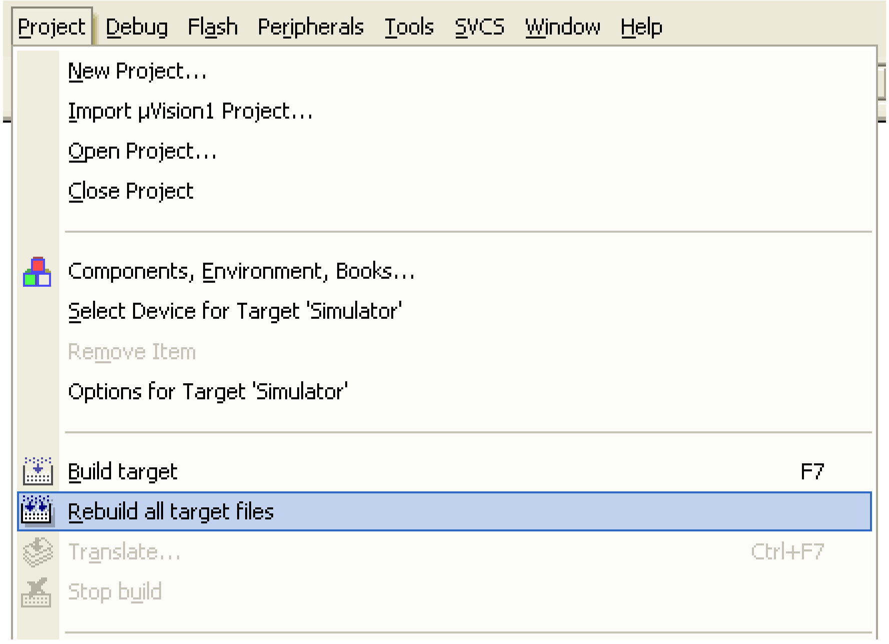
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

3. Select the target **MCBSTM32**.

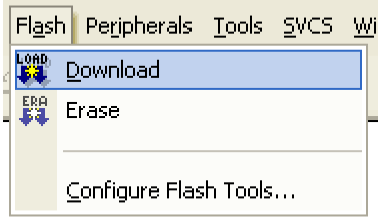


4. Rebuild the executable file from source files. Click on **Project —> Rebuild all target files** on the menu or click on the toolbar button .



5. Connect the ULINK device to the PC's USB port and to the JTAG Connector on the MCBSTM32 board.

6. Power-up the MCBSTM32 board by connecting the board's USB power input to the PC's USB port.
7. Click on **Flash —> Download** on the menu to download the executable file to the STM32 flash target on the MCBSTM32 board.



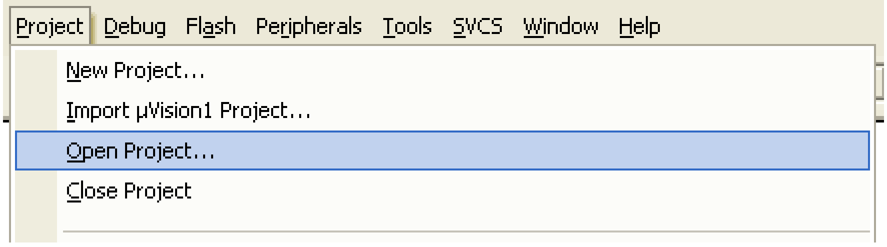
8. Rotate the potentiometer and watch the Tx and RX values on the LCD, and the state of the LEDs change according to the position of the potentiometer.

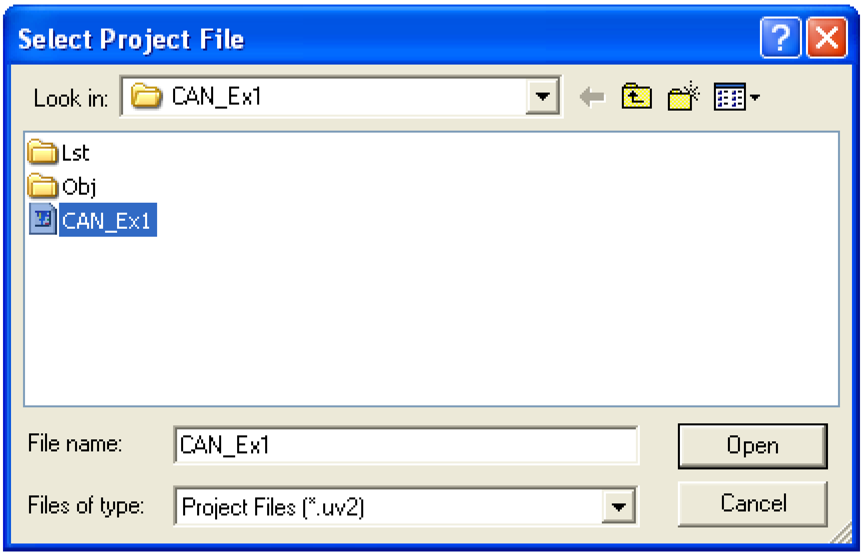
Simulation

You can test the programs you create with the Real-Time Library CAN driver (RL-CAN) using the simulation capabilities of the µVision® IDE. Simulation allows you to test your CAN application before the target hardware is ready.

Follow these steps to test an RL-CAN example application using the µVision Simulator.

1. Copy the example **\KEIL\ARM\Boards\Keil\MCBSTM32\RL\CAN\CAN_Ex1** to a new folder.
2. Select and load the project file (**CAN_Ex1.UV2**) into µVision. This file is located in the folder copied in step 1. In µVision, use **Project —> Open Project...**

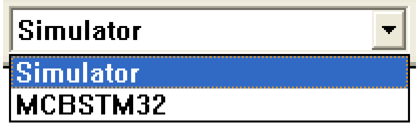




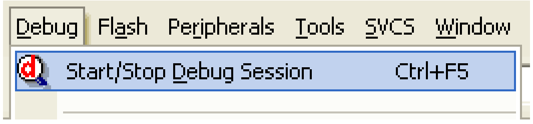
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

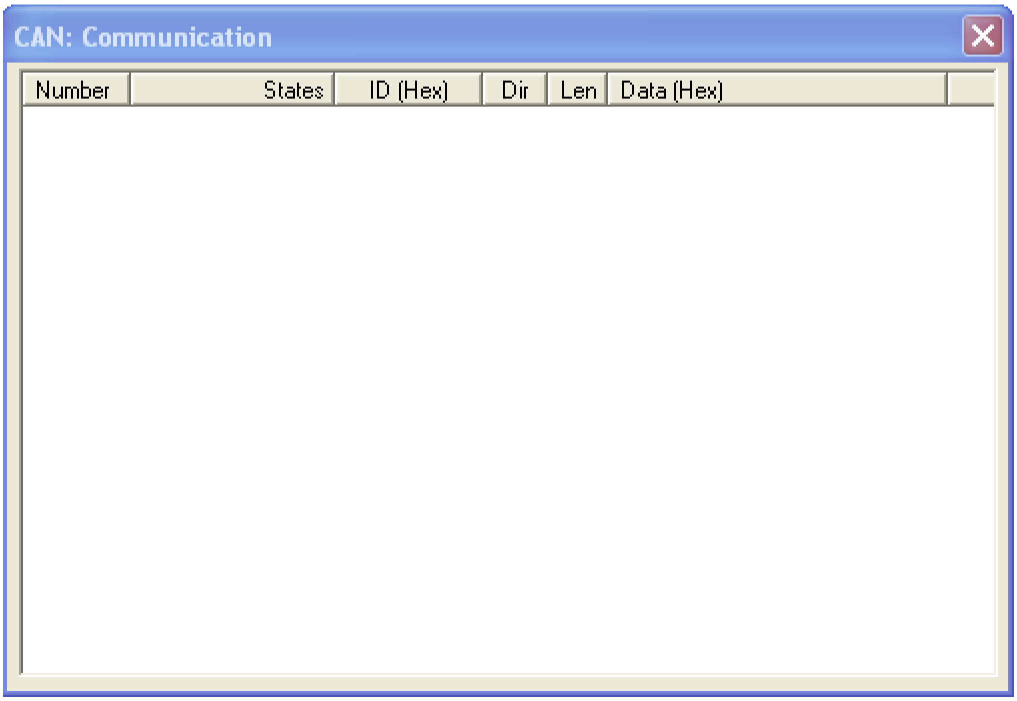
3. Select the target **Simulator**.

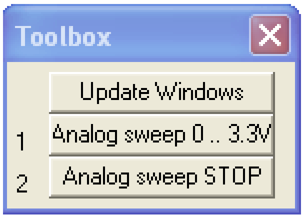


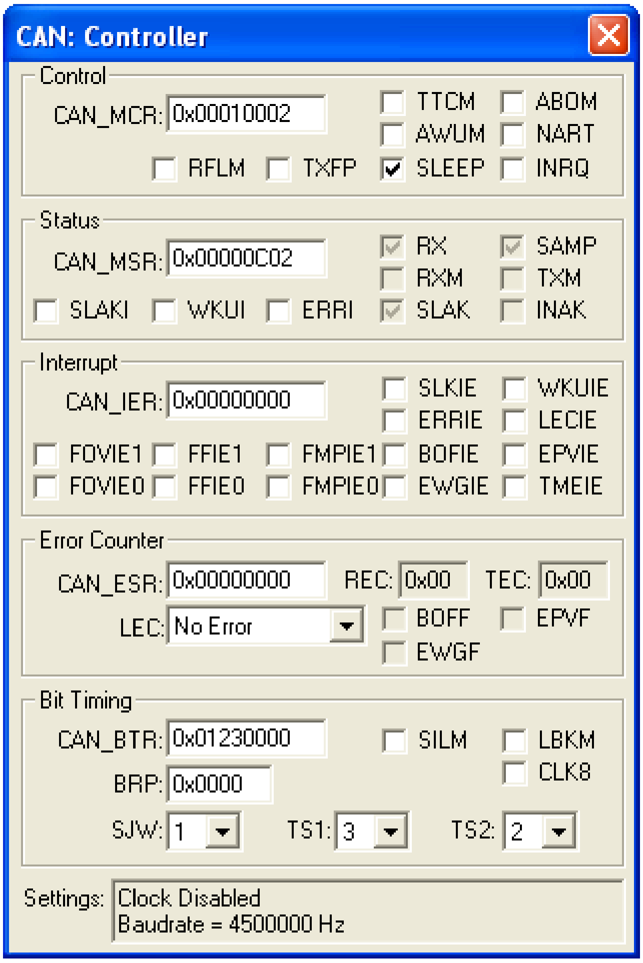
4. Click on **Debug —> Start/Stop Debug Session** on the menu bar or click the toolbar button to run the simulation.



5. New windows appear as shown below. These windows show the device peripheral functionality.







Control

CAN_MCR: 0x00010002

□ TTCM □ ABOM

□ AWUM □ NART

□ RFLM □ TXFP ☑ SLEEP □ INRQ

Status

CAN_MSR: 0x00000C02

☑ RX ☑ SAMP

□ RXM □ TXM

□ SLAKI □ WKUI □ ERRI ☑ SLAK □ INAK

Interrupt

CAN_IER: 0x00000000

□ SLKIE □ WKUIE

□ ERRIE □ LECIE

□ FOVIE1 □ FFIE1 □ FMPIE1 □ BOFIE □ EPVIE

□ FOVIE0 □ FFIE0 □ FMPIE0 □ EWGIE □ TMEIE

Error Counter

CAN_ESR: 0x00000000 REC: 0x00 TEC: 0x00

LEC: No Error □ BOFF □ EPVF

□ EWGF

Bit Timing

CAN_BTR: 0x01230000 □ SILM □ LBKM

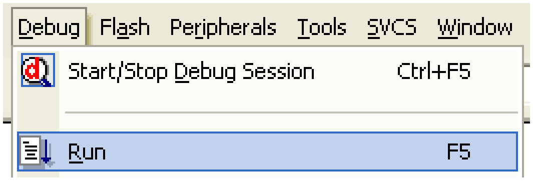
□ CLK8

BRP: 0x0000

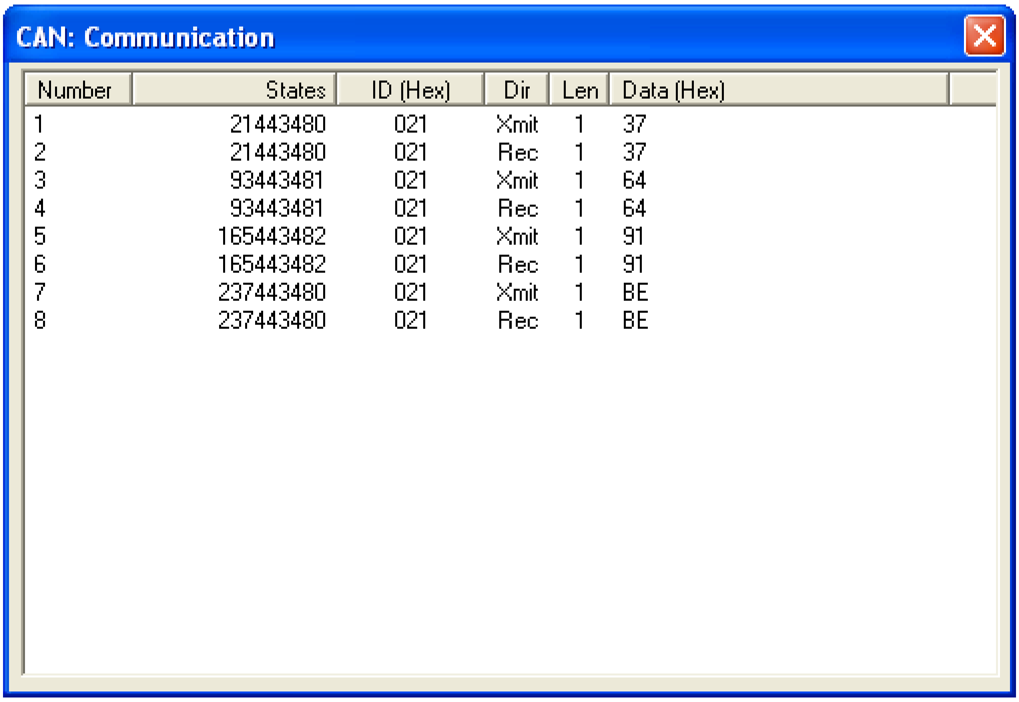
SJW: 1 TS1: 3 TS2: 2

Settings: Clock Disabled
Baudrate = 4500000 Hz

6. Click on **Debug —> Run** to start the simulation.



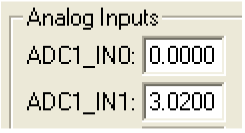
7. The communication messages display in the windows.



CAN: Communication

Number	States	ID (Hex)	Dir	Len	Data (Hex)
1	21443480	021	Xmit	1	37
2	21443480	021	Rec	1	37
3	93443481	021	Xmit	1	64
4	93443481	021	Rec	1	64
5	165443482	021	Xmit	1	91
6	165443482	021	Rec	1	91
7	237443480	021	Xmit	1	BE
8	237443480	021	Rec	1	BE

8. Click on the **Analog sweep 0 .. 3.3V** button, on the **Toolbox** window, to start the analog value rising from 0 V to 3.3 V and lowering from 3.3 V to 0 V. You can stop the analog value change by clicking on **Analog sweep STOP**.

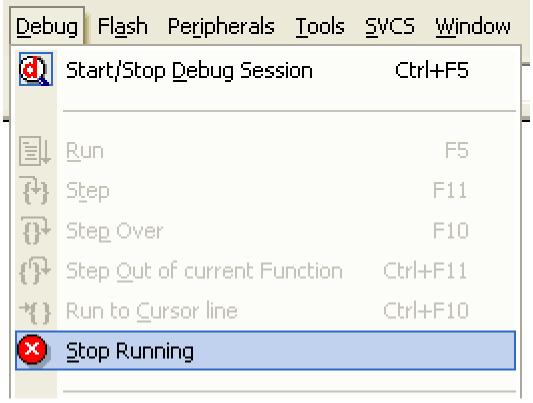


Analog Inputs

ADC1_IN0: 0.0000

ADC1_IN1: 3.0200

9. Stop the simulation before exiting µVision by clicking on **Debug —> Stop** on the Debug menu.


Debug
Flash
Peripherals
Tools
SVCS
Window
Start/Stop Debug Session Ctrl+F5
Run F5
Step F11
Step Over F10
Step Out of current Function Ctrl+F11
Run to Cursor line Ctrl+F10
Stop Running

ST STM32F105/7 Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_STM32F107.c** file:

- **CAN_CLK** (Peripheral Clock) constant is used to calculate the correct communication speed (baudrate).

```
#define CAN_CLK                36000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL2** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1           1
  #define USE_CAN_CTRL2           1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

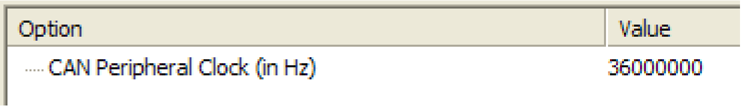
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

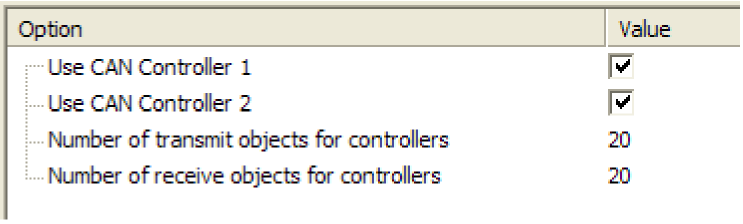
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_STM32F107.c** edited with Configuration Wizard:



Option	Value
CAN Peripheral Clock (in Hz)	36000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

ST STR71x Devices

Configuration

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

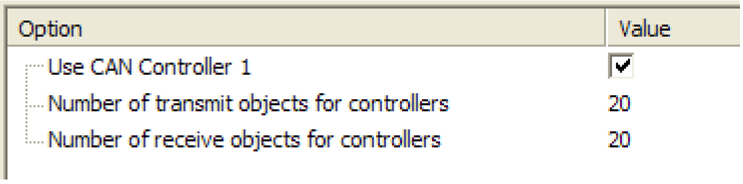
  ```
  #define USE_CAN_CTRL1            1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.
-

-
  ```
  #define CAN_No_SendObjects     20
  ```
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.
-

-
  ```
  #define CAN_No_ReceiveObjects  20
  ```

Using Configuration Wizard

File **CAN_Cfg.h** edited with Configuration Wizard:



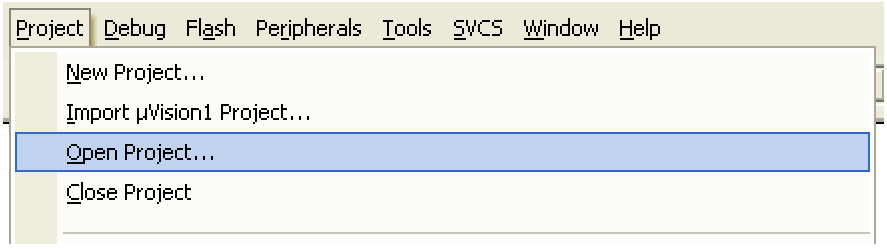
Option	Value
Use CAN Controller 1	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

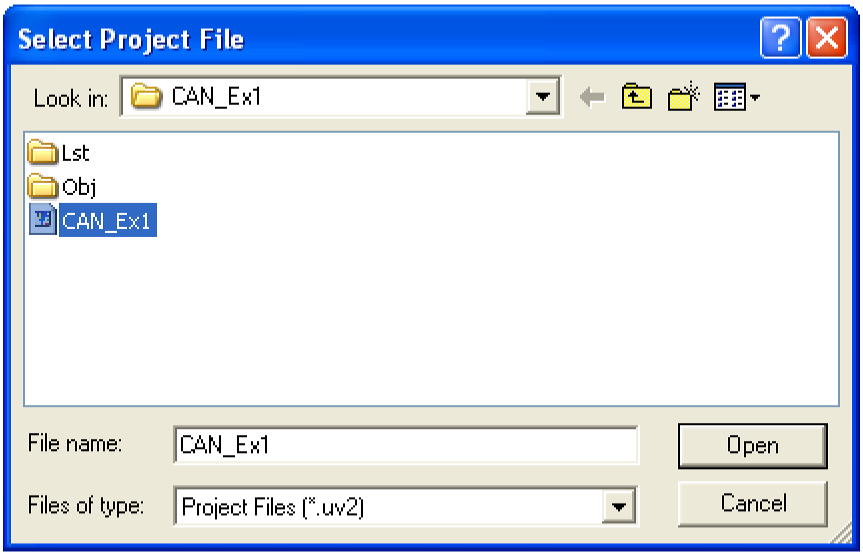
Getting Started

To run this RL-CAN STR71x example on a board, ensure you have:

- an MCBSTR7 Evaluation board from Keil
- a ULINK® USB Interface Adapter from Keil
- a 6 - 9 V power supply.

1. Copy the example from **\Keil\ARM\Boards\Keil\MCBSTR7\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCBSTR7\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 3 IDE (**Project — Open Project...**)



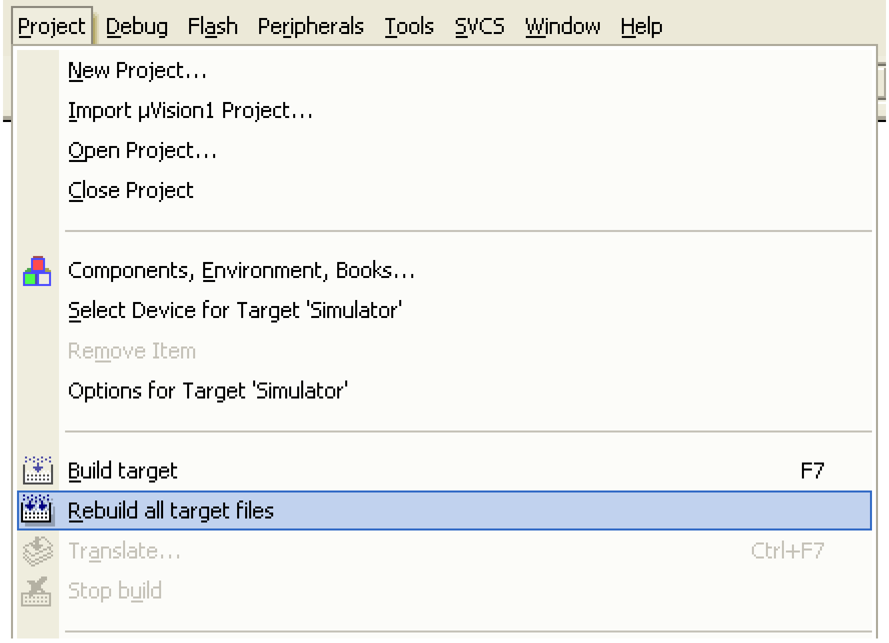


Look in: CAN_Ex1

File name: CAN_Ex1

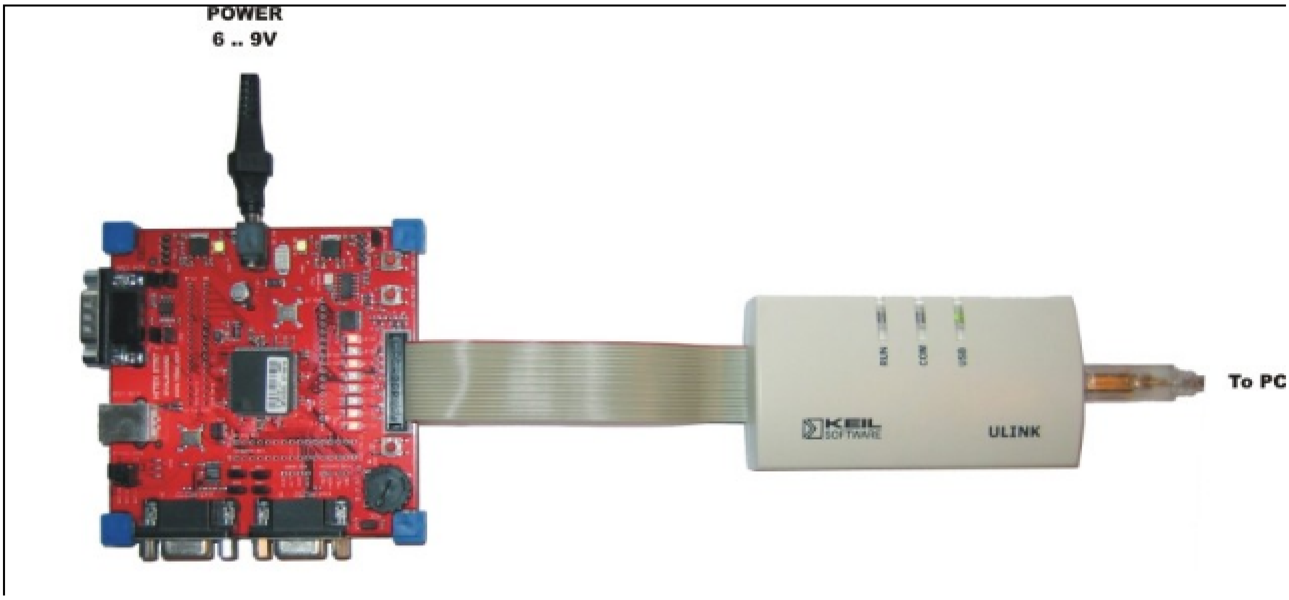
Files of type: Project Files (*.uv2)

3. Rebuild the executable file from source files. Click on **Project — Rebuild all target files** on the menu or click on the toolbar button .

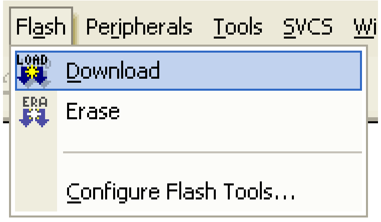


4. Connect the ULINK device to the PC's USB port and to the JTAG Connector on the MCBSTR7

board.
Connect a power cord to the barrel plug port on the MCBSTR7 board.



5. Power-up the MCBSTR7 board.
6. Click on **Flash — Download** on the menu to download the executable file to the target MCBSTR7 board.



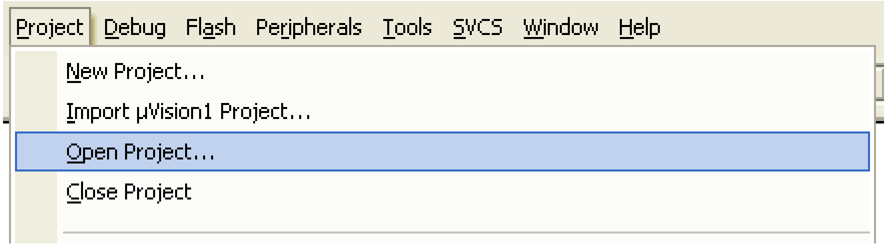
7. Rotate the potentiometer and watch the LEDs change state and flash speed according to the position of the potentiometer.

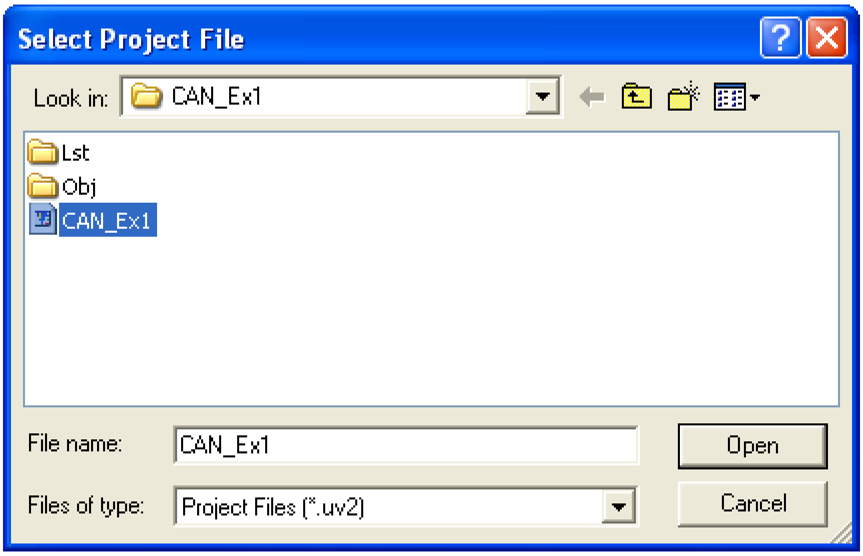
Simulation

You can test the programs you create with the Real-Time Library CAN driver (RL-CAN) using the simulation capabilities of the µVision® IDE. Simulation allows you to test your CAN application before the target hardware is ready.

Follow these steps to test an RL-CAN example application using the µVision Simulator.

1. Copy one of the examples (**\KEIL\ARM\Boards\Keil\MCBSTR7\RL\CAN\CAN_Ex1** or **\KEIL\ARM\Boards\Keil\MCBSTR7\RL\CAN\CAN_Ex2**) to a new folder.
2. Select and load the project file (**CAN_Ex1.UV2** or **CAN_Ex2.UV2**) into µVision. This file is located in the folder copied in step 1. In µVision, use **Project — Open Project...**

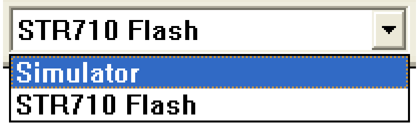




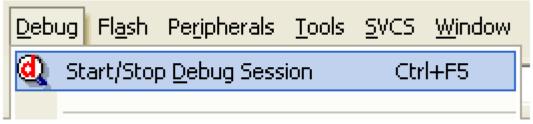
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

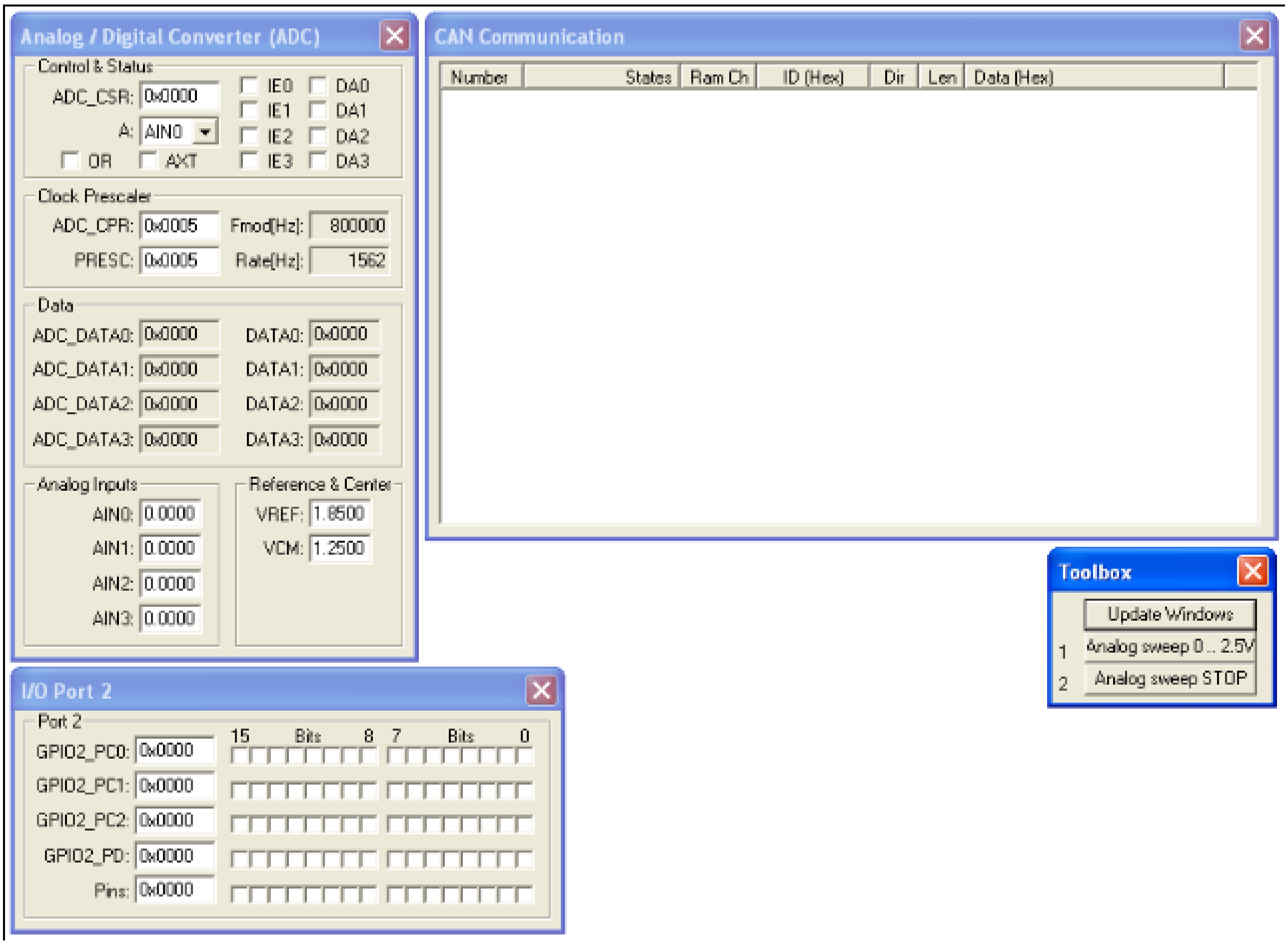
3. Select the target **Simulator**.



4. Click on **Debug — Start/Stop Debug Session** on the menu bar or click the toolbar button to run the simulation.



5. New windows appear as shown below. These windows show the device peripheral functionality.



ADC_CSR: 0x0000 □ IE0 □ DA0
A: AIN0 □ IE1 □ DA1
□ IE2 □ DA2
□ OR □ AXT □ IE3 □ DA3

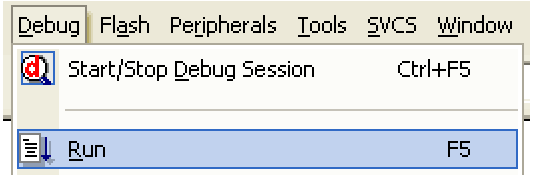
ADC_CPR: 0x0005 Fmod[Hz]: 800000
PRESC: 0x0005 Rate[Hz]: 1562

ADC_DATA0: 0x0000 DATA0: 0x0000
ADC_DATA1: 0x0000 DATA1: 0x0000
ADC_DATA2: 0x0000 DATA2: 0x0000
ADC_DATA3: 0x0000 DATA3: 0x0000

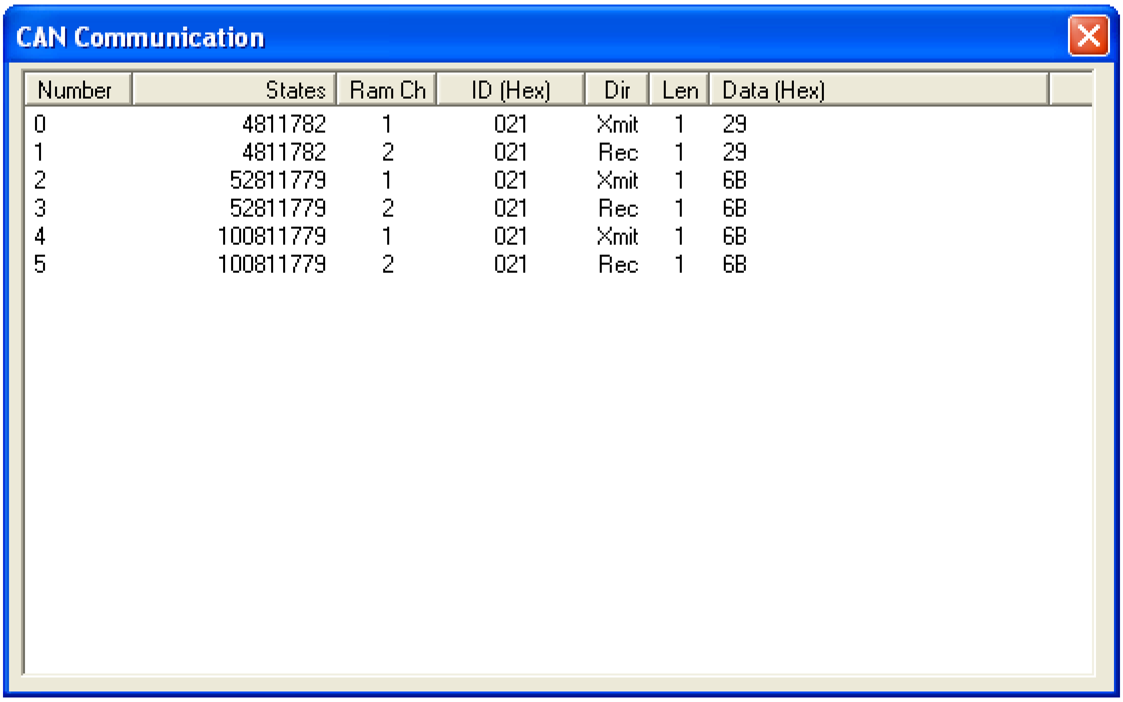
AIN0: 0.0000 VREF: 1.8500
AIN1: 0.0000 VCM: 1.2500
AIN2: 0.0000
AIN3: 0.0000

GPIO2_PC0: 0x0000
GPIO2_PC1: 0x0000
GPIO2_PC2: 0x0000
GPIO2_PD: 0x0000
Pins: 0x0000

6. Click on **Debug — Run** to start the simulation.

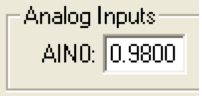


7. The communication messages display in the **CAN Communication** window.



Number	States	Ram Ch	ID (Hex)	Dir	Len	Data (Hex)
0	4811782	1	021	Xmit	1	29
1	4811782	2	021	Rec	1	29
2	52811779	1	021	Xmit	1	6B
3	52811779	2	021	Rec	1	6B
4	100811779	1	021	Xmit	1	6B
5	100811779	2	021	Rec	1	6B

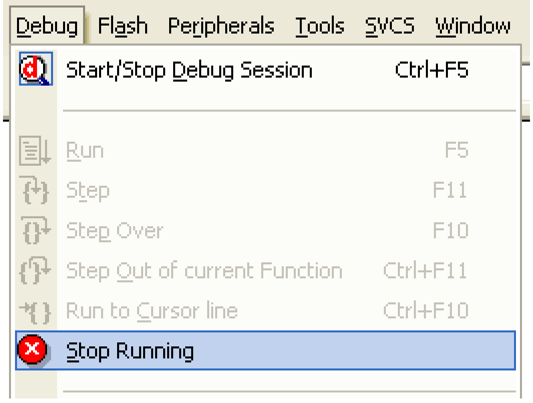
8. Click on the **Analog sweep 0 .. 2.5V** button, on the **Toolbox** window, to start the analog value changing by rising from 0 V to 2.5 V and lowering from 2.5 V to 0 V. You can stop the analog value change by clicking on **Analog sweep STOP**.



Analog Inputs

AIN0: 0.9800

9. Stop the simulation before exiting µVision by clicking on **Debug — Stop** on the Debug menu.


Debug
Flash
Peripherals
Tools
SVCS
Window
Start/Stop Debug Session Ctrl+F5
Run F5
Step F11
Step Over F10
Step Out of current Function Ctrl+F11
Run to Cursor line Ctrl+F10
Stop Running

ST STR73x Devices

Configuration

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL3** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

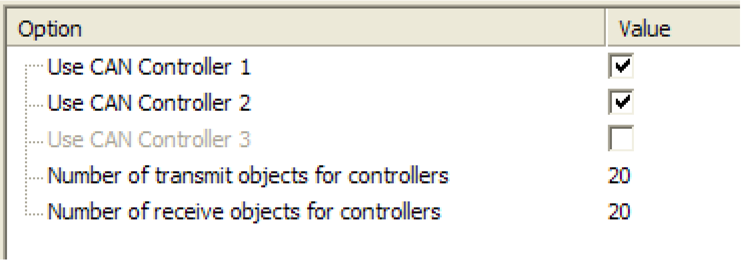
  ```
  #define USE_CAN_CTRL1           1
  #define USE_CAN_CTRL2           1
  #define USE_CAN_CTRL3           0
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

  ```
  #define CAN_No_SendObjects     20
  ```
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

  ```
  #define CAN_No_ReceiveObjects  20
  ```

Using Configuration Wizard

File **CAN_Cfg.h** edited with Configuration Wizard:



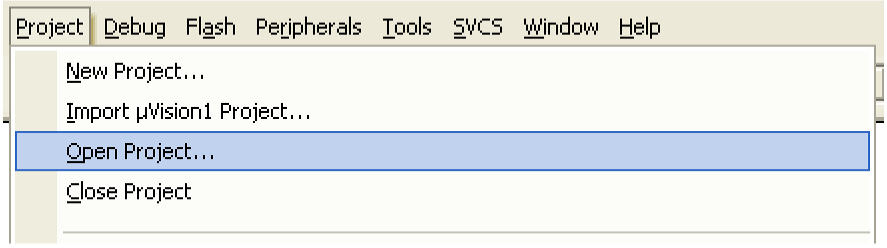
Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	☑
Use CAN Controller 3	☐
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

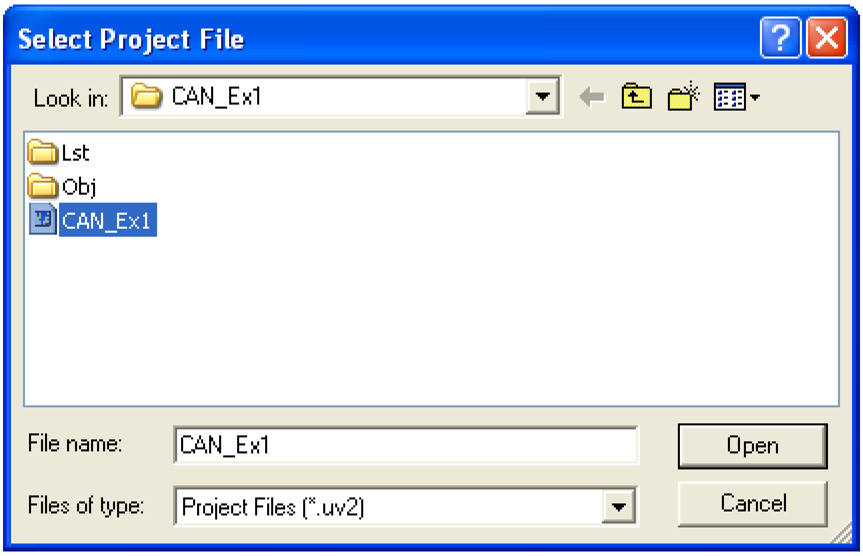
Getting Started

To run this RL-CAN STR73x example on a board, ensure you have:

- an STR730 Evaluation board from Hitex
- a ULINK® USB Interface Adapter from Keil
- a 6 - 9 V power supply
- a loopback cable (DB-9 female to DB-9 female).

1. Copy the example from **\Keil\ARM\Boards\Keil\MCBSTR730\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCBSTR730\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 3 IDE (**Project — Open Project...**)





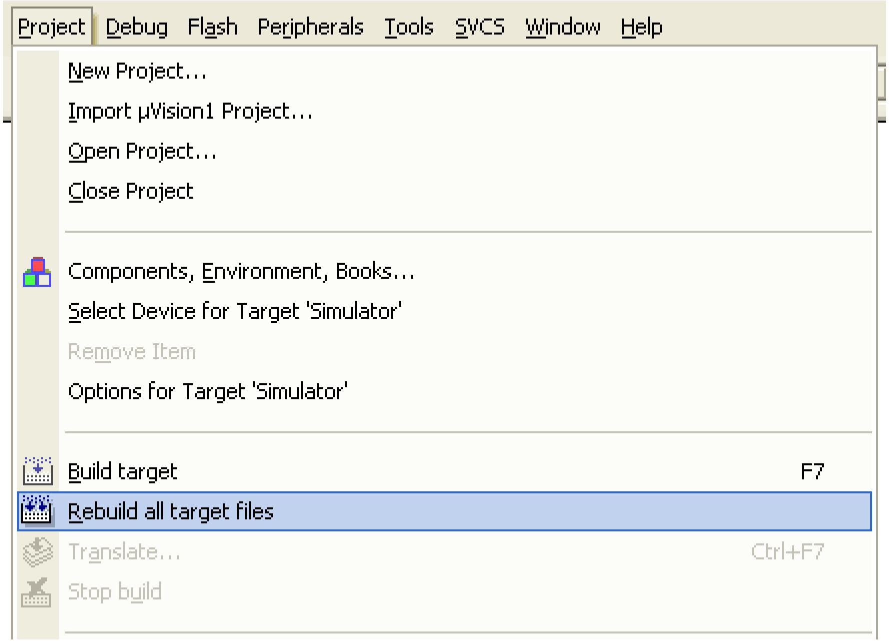
Look in: CAN_Ex1

CAN_Ex1

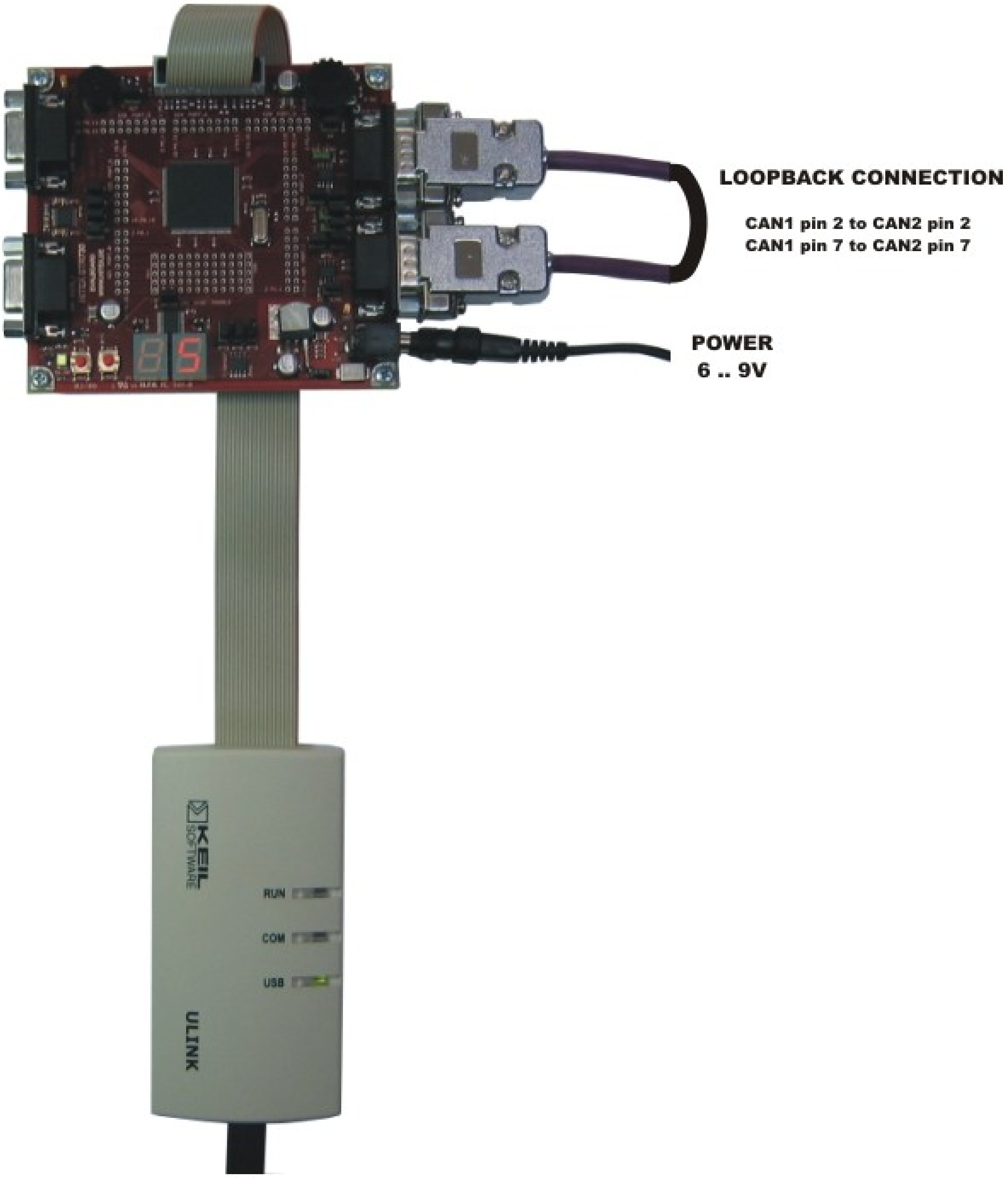
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

3. Rebuild the executable file from source files. Click on **Project — Rebuild all target files** on the menu or click on the toolbar button .

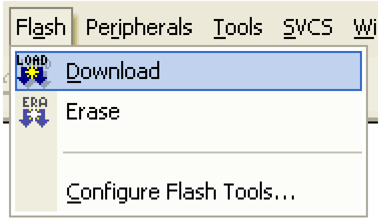


4. Connect the ULINK device to the PC's USB port and to the JTAG Connector on the STR730 board.
   Connect the power cord to the barrel plug port on the STR730 board.



To PC

5. Power-up the STR730 board.
6. Click on **Flash — Download** on the menu to download the executable file to the target STR730 board.



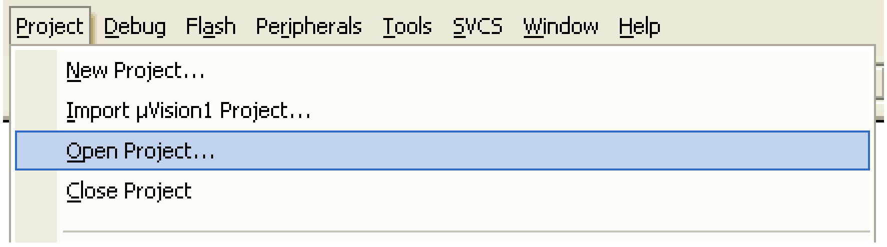
7. Rotate the potentiometer and watch the value change on the 7-segment LCDs according to the position of the potentiometer.

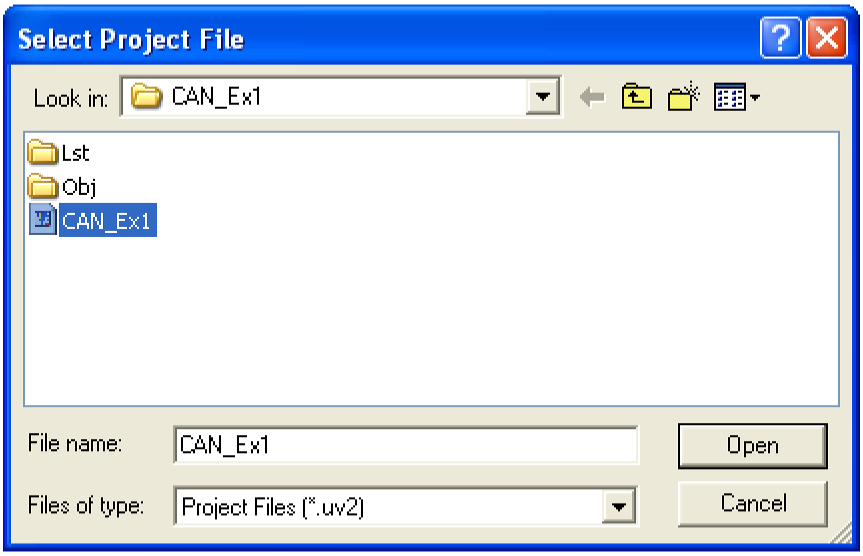
Simulation

You can test the programs you create with the Real-Time Library CAN driver (RL-CAN) using the simulation capabilities of the µVision® IDE. Simulation allows you to test your CAN application before the target hardware is ready.

Follow these steps to test an RL-CAN example application using the µVision Simulator.

1. Copy one of the examples (**\KEIL\ARM\Boards\Keil\MCBSTR730\RL\CAN\CAN_Ex1** or **\KEIL\ARM\Boards\Keil\MCBSTR730\RL\CAN\CAN_Ex2**) to a new folder.
2. Select and load the project file (**CAN_Ex1.UV2** or **CAN_Ex2.UV2**) into µVision. This file is located in the folder copied in step 1. In µVision, use **Project — Open Project...**

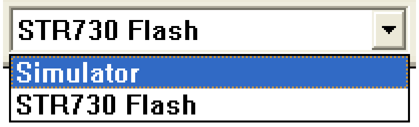




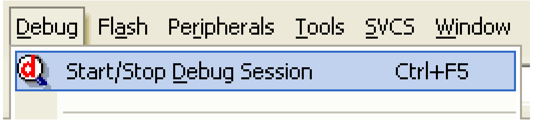
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

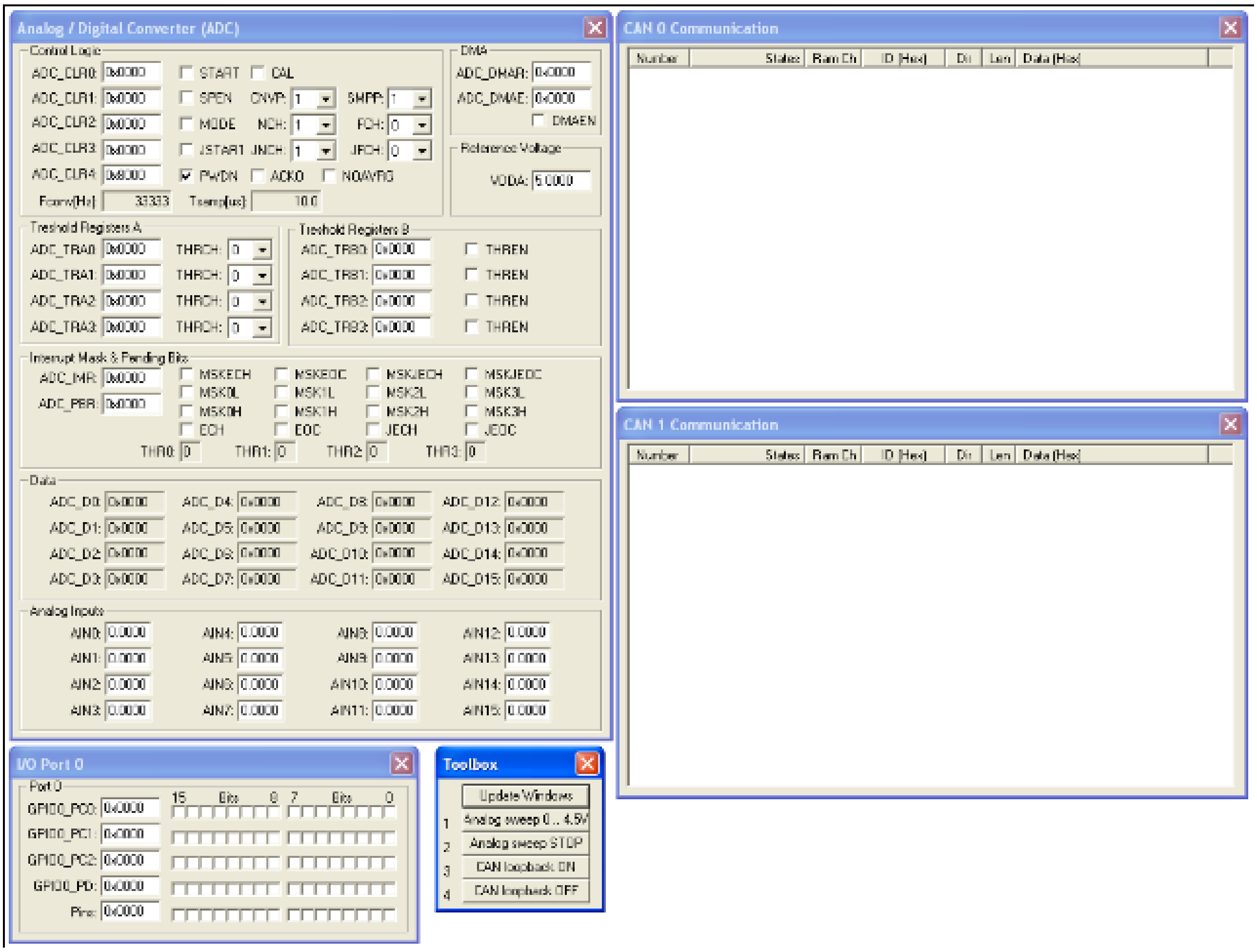
3. Select the target **Simulator**.



4. Click on **Debug — Start/Stop Debug Session** on the menu bar or click the toolbar button to run the simulation.



5. New windows appear as shown below. These windows show the device peripheral functionality.



ADC_CLR0: 0x0000 □ START □ CAL

ADC_CLR1: 0x0000 □ SPEN CNVP: 1 SMPP: 1

ADC_CLR2: 0x0000 □ MODE NCH: 1 FCH: 0

ADC_CLR3: 0x0000 □ JSTART JNCH: 1 JFCH: 0

ADC_CLR4: 0x8000 ☑ PWDN □ ACKO □ NOAVRG

Fconv(Hz) 33333 Tsamp(us) 10.0

ADC_DMAR: 0x0000

ADC_DMAE: 0x0000

□ DMAEN

VDDA: 5.0000

ADC_TRA0: 0x0000 THRCH: 0 ADC_TRB0: 0x0000 □ THREN

ADC_TRA1: 0x0000 THRCH: 0 ADC_TRB1: 0x0000 □ THREN

ADC_TRA2: 0x0000 THRCH: 0 ADC_TRB2: 0x0000 □ THREN

ADC_TRA3: 0x0000 THRCH: 0 ADC_TRB3: 0x0000 □ THREN

ADC_IMR: 0x0000 □ MSKECH □ MSKEOC □ MSKJECH □ MSKJEOC

ADC_PBR: 0x0000 □ MSK0L □ MSK1L □ MSK2L □ MSK3L

□ MSK0H □ MSK1H □ MSK2H □ MSK3H

□ ECH □ EOC □ JECH □ JEOC

THR0: 0 THR1: 0 THR2: 0 THR3: 0

ADC_D0: 0x0000 ADC_D4: 0x0000 ADC_D8: 0x0000 ADC_D12: 0x0000

ADC_D1: 0x0000 ADC_D5: 0x0000 ADC_D9: 0x0000 ADC_D13: 0x0000

ADC_D2: 0x0000 ADC_D6: 0x0000 ADC_D10: 0x0000 ADC_D14: 0x0000

ADC_D3: 0x0000 ADC_D7: 0x0000 ADC_D11: 0x0000 ADC_D15: 0x0000

AIN0: 0.0000 AIN4: 0.0000 AIN8: 0.0000 AIN12: 0.0000

AIN1: 0.0000 AIN5: 0.0000 AIN9: 0.0000 AIN13: 0.0000

AIN2: 0.0000 AIN6: 0.0000 AIN10: 0.0000 AIN14: 0.0000

AIN3: 0.0000 AIN7: 0.0000 AIN11: 0.0000 AIN15: 0.0000

GPIO0_PC0: 0x0000

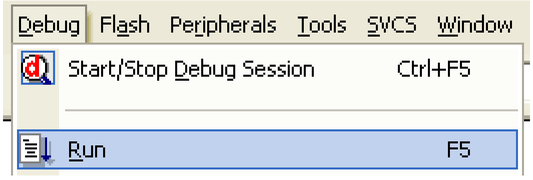
GPIO0_PC1: 0x0000

GPIO0_PC2: 0x0000

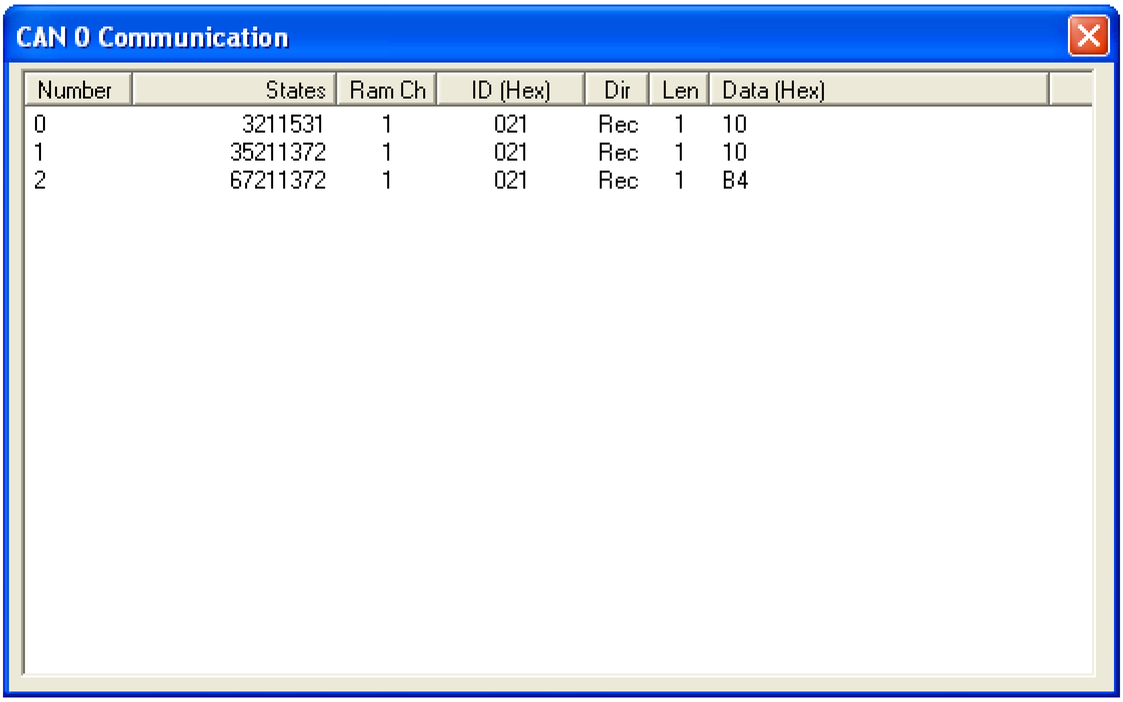
GPIO0_PD: 0x0000

Pins: 0x0000

6. Click on **Debug — Run** to start the simulation.

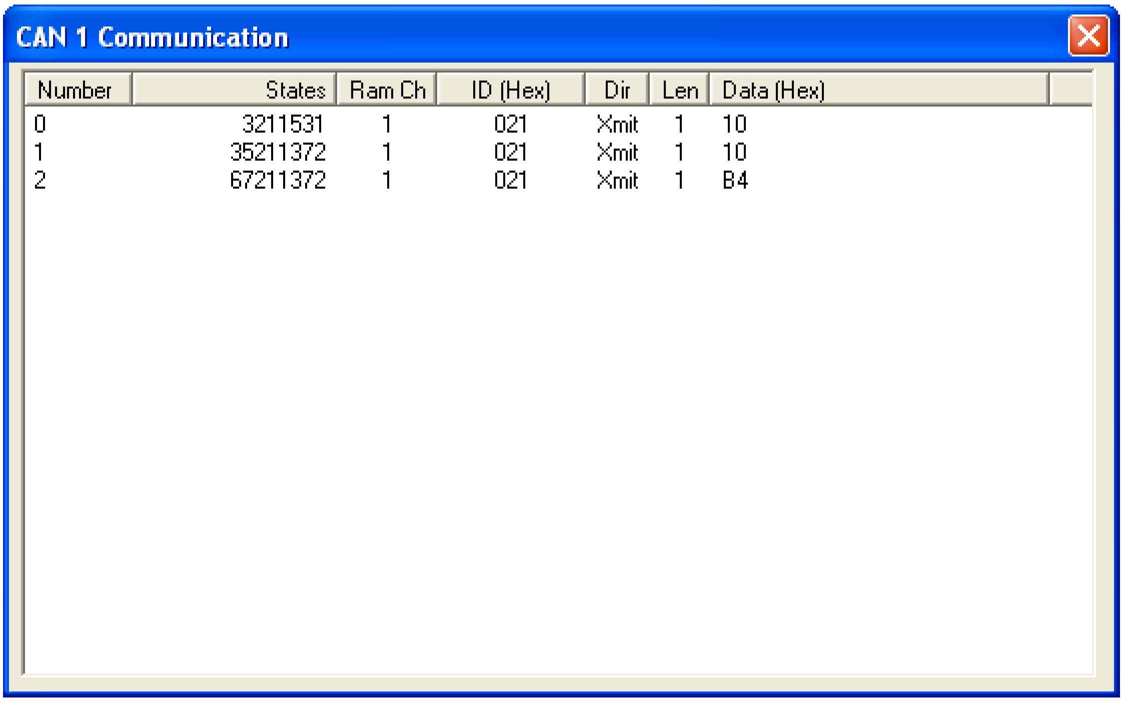


7. The communication messages display in the **CAN 0 Communication** and **CAN 1 Communication** windows.



CAN 0 Communication

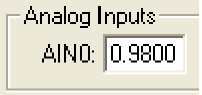
Number	States	Ram Ch	ID (Hex)	Dir	Len	Data (Hex)
0	3211531	1	021	Rec	1	10
1	35211372	1	021	Rec	1	10
2	67211372	1	021	Rec	1	B4



CAN 1 Communication

Number	States	Ram Ch	ID (Hex)	Dir	Len	Data (Hex)
0	3211531	1	021	Xmit	1	10
1	35211372	1	021	Xmit	1	10
2	67211372	1	021	Xmit	1	B4

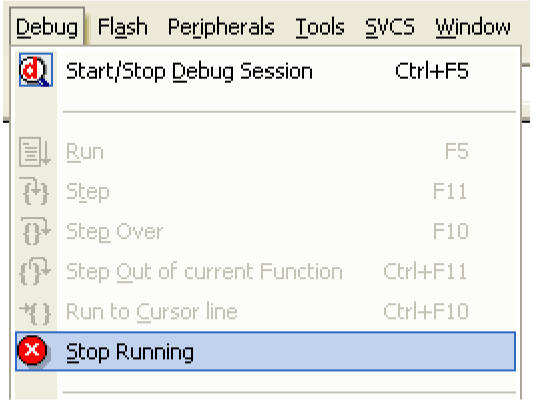
8. Click on the **Analog sweep 0 .. 4.5V** button, on the **Toolbox** window, to start the analog value rising from 0 V to 4.5 V and lowering from 4.5 V to 0 V. You can stop the analog value change by clicking on **Analog sweep STOP**.



Analog Inputs

AIN0: 0.9800

9. Stop the simulation before exiting µVision by clicking on **Debug — Stop** on the Debug menu.


Debug
Flash
Peripherals
Tools
SVCS
Window
Start/Stop Debug Session Ctrl+F5
Run F5
Step F11
Step Over F10
Step Out of current Function Ctrl+F11
Run to Cursor line Ctrl+F10
Stop Running

ST STR91x Devices

Configuration

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

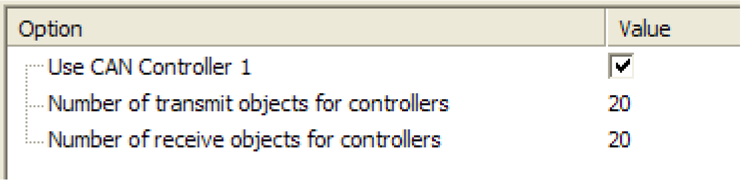
  ```
  #define USE_CAN_CTRL1            1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

  ```
  #define CAN_No_SendObjects     20
  ```
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

  ```
  #define CAN_No_ReceiveObjects  20
  ```

Using Configuration Wizard

File **CAN_Cfg.h** edited with Configuration Wizard:



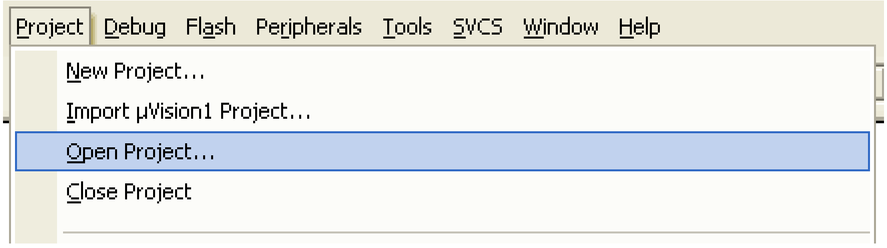
Option	Value
Use CAN Controller 1	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

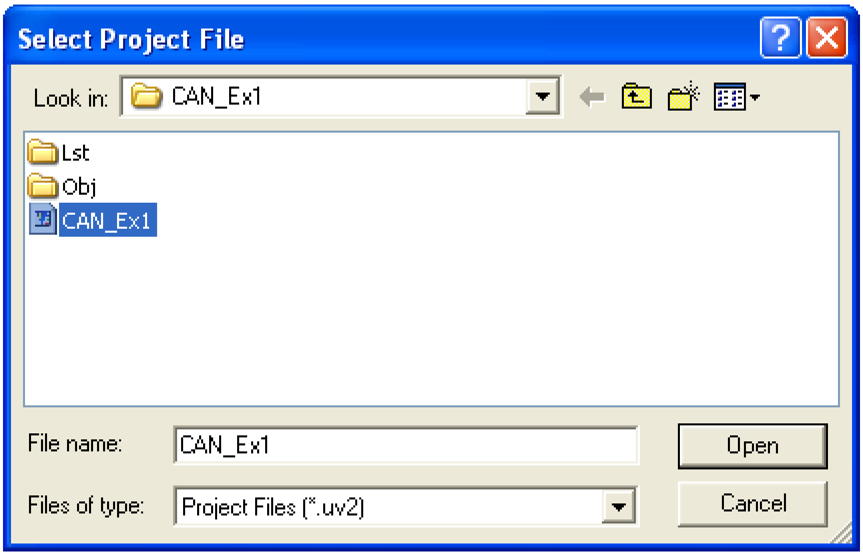
Getting Started

To run this RL-CAN STR91x example on a board, ensure you have:

- an MCBSTR9 Evaluation board from Keil
- a ULINK® USB Interface Adapter from Keil
- a USB cable for 5 V power supply.

1. Copy the example from **\Keil\ARM\Boards\Keil\MCBSTR9\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCBSTR9\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 3 IDE (**Project — Open Project...**)




Select Project File

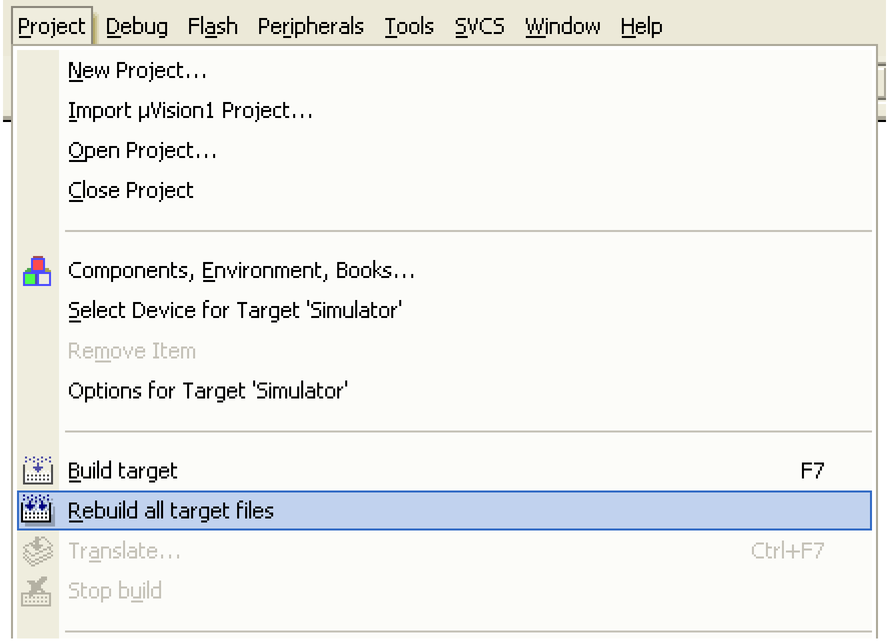
Look in: CAN_Ex1

Lst
Obj
CAN_Ex1

File name: CAN_Ex1

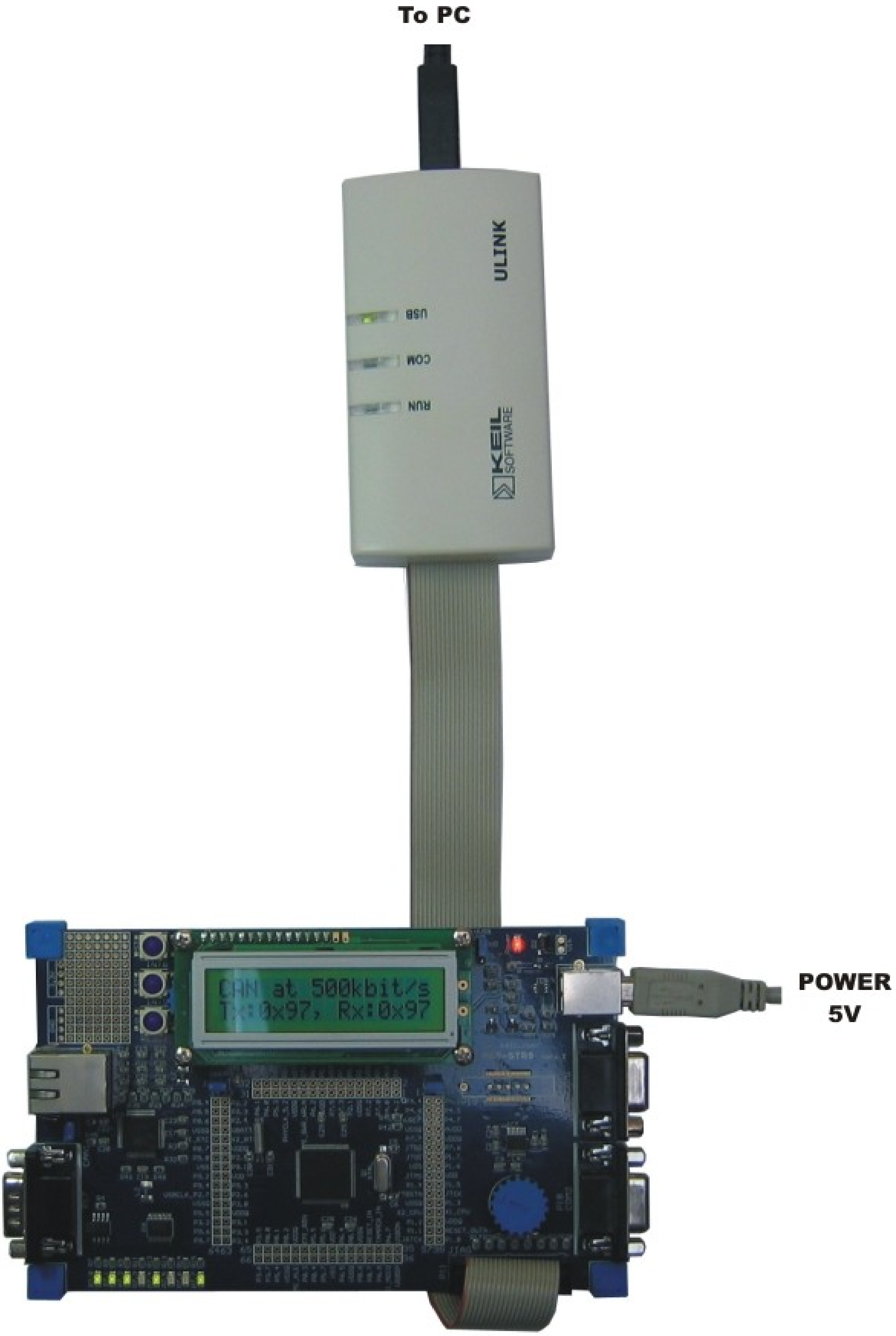
Files of type: Project Files (*.uv2)

3. Rebuild the executable file from source files. Click on **Project —; Rebuild all target files** on the menu or click on the toolbar button .



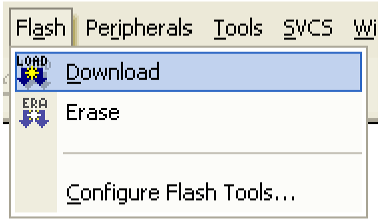
4. Connect the ULINK device to the PC's USB port and to the JTAG Connector on the MCBSTR9

board.



5. Power-up the MCBSTR9 board by connecting the board's USB power input to the PC's USB port.

6. Click on **Flash — Download** on the menu to download the executable file to the STR910 flash target on the MCBSTR9 board.



7. Rotate the potentiometer and watch the Tx and RX values on the LCD, and the state of the LEDs change according to the position of the potentiometer.

Toshiba TMPM36x Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_TMPM36x.c** file:

- **CAN_PERI_FREQ** (**CAN PERI**pheral **FREQ**ency) constant is used to calculate the correct communication speed (baudrate).

```
#define CAN_PERI_FREQ         12000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1            1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

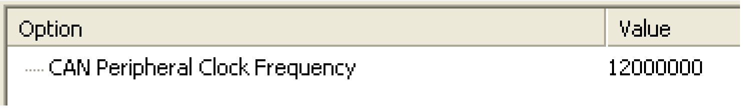
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

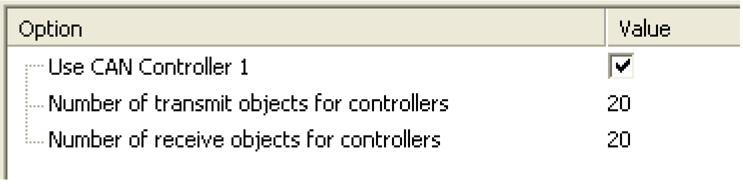
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_TMPM36x.c** edited with Configuration Wizard:



Option	Value
CAN Peripheral Clock Frequency	12000000

File **CAN_Cfg.h** edited with Configuration Wizard:



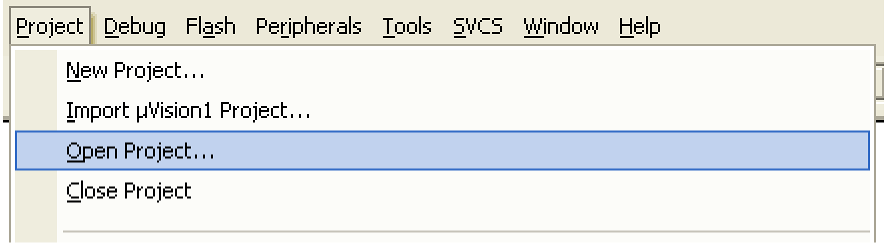
Option	Value
Use CAN Controller 1	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

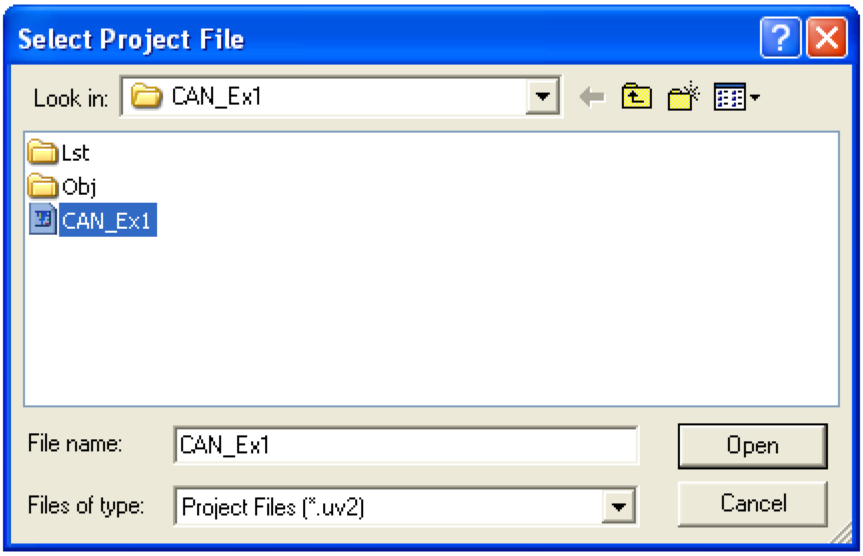
Getting Started

To run this RL-CAN TMPM36x example on a board, ensure you have:

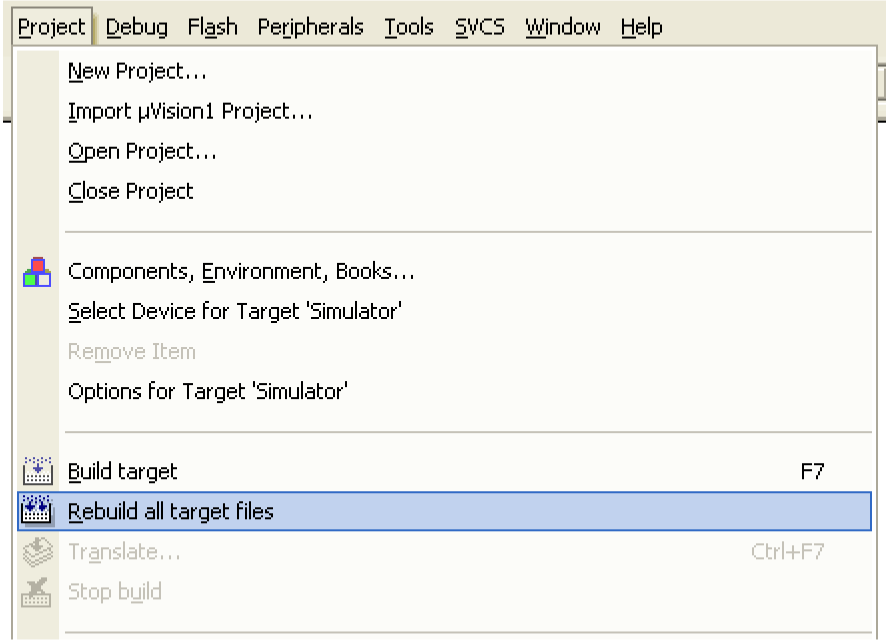
- an MCBTMPM360 Evaluation board from Keil
- a ULINK® USB Interface Adapter from Keil
- a USB cable for 5 V power supply.

1. Copy the example from **\Keil\ARM\Boards\Keil\MCBTMPM360\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Keil\MCBTMPM360\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 4 IDE (**Project —> Open Project...**)

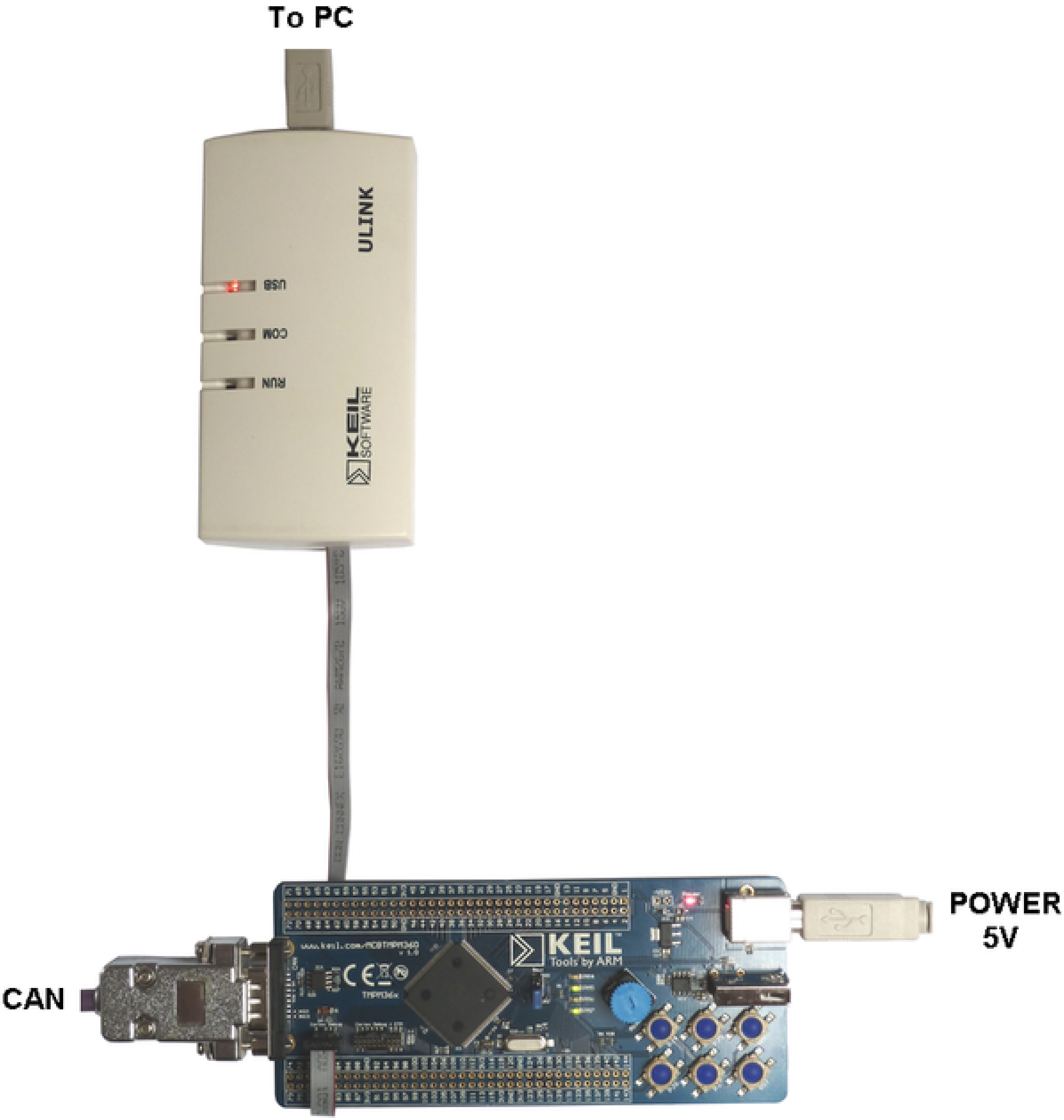




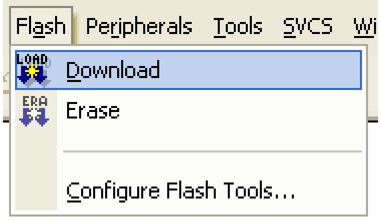
3. Rebuild the executable file from source files. Click on **Project —> Rebuild all target files** on the menu or click on the toolbar button .



4. Connect the ULINK device to the PC's USB port and to the Cortex Debug Connector on the MCBTMPM360 board.



5. Power-up the MCBTMPM360 board by connecting the board's USB power input to the PC's USB port.
6. Click on **Flash —> Download** on the menu to download the executable file to the TMPM364 flash target on the MCBTMPM360 board.



7. Press button PJ4 and keep it pressed while resetting the board (to enable loopback mode in which everything that is sent on CAN is also received).
8. Rotate the potentiometer and watch the the state of the LEDs change according to the position of the potentiometer.

Luminary LM3Sxxxx Devices

Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_LM3Sxxxx.c** file:

- **CAN_CLOCK** (Peripheral clock) constant is used to calculate the correct communication speed (baudrate).

```
#define CAN_CLOCK         8000000
```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1 .. USE_CAN_CTRL3** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1           1
  #define USE_CAN_CTRL2           0
  #define USE_CAN_CTRL3           0
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.

```
#define CAN_No_SendObjects     20
```

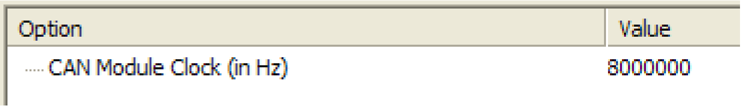
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.

```
#define CAN_No_ReceiveObjects  20
```

Using Configuration Wizard

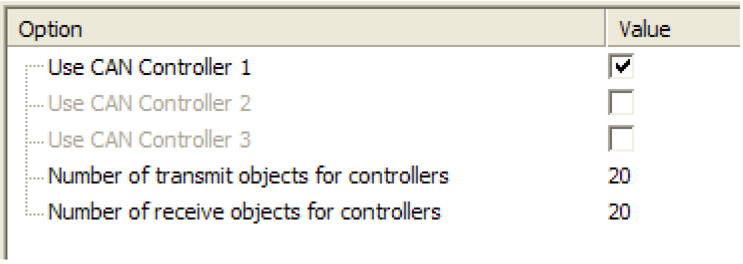
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_LM3Sxxxx.c** edited with Configuration Wizard:



Option	Value
CAN Module Clock (in Hz)	8000000

File **CAN_Cfg.h** edited with Configuration Wizard:



Option	Value
Use CAN Controller 1	☑
Use CAN Controller 2	□
Use CAN Controller 3	□
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

Atmel AT91SAM7X Devices

Hardware Configuration

CAN Hardware Configuration

The following symbols specify CAN Hardware Configuration related parameters and are located in **CAN_SAM7X.c** file:

- **MCK_FREQ** (Master Clock Frequency) constant is used to calculate the correct communication speed (baudrate). The value of this constant must be calculated manually and is dependent on the Power Management Controller (PMC) settings in the SAM7.s startup file.
-

-
  ```
  #define  MCK_FREQ     47923200
  ```

CAN Generic Configuration

The following symbols specify CAN Generic Configuration related parameters and are located in **CAN_Cfg.h** file:

- **USE_CAN_CTRL1** enables the CAN controller that will be used. To enable the CAN controller, set this value to 1.

  The RL-CAN uses this information to reserve the memory pool and the memory for software FIFO buffers, as well as to enable interrupts for handling transmission and reception of CAN messages on the specified CAN controller.

  ```
  #define USE_CAN_CTRL1           1
  ```
- **CAN_No_SendObjects** specifies the size of the software message FIFO buffers for message sending. Each CAN controller reserves a specified size of software buffer to send messages.
-

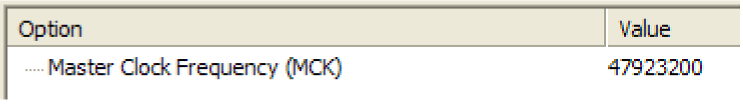
-
  ```
  #define CAN_No_SendObjects     20
  ```
- **CAN_No_ReceiveObjects** specifies the size of the software message FIFO buffers for message reception. Each CAN controller reserves a specified size of software buffer to receive messages.
-

-
  ```
  #define CAN_No_ReceiveObjects  20
  ```

Using Configuration Wizard

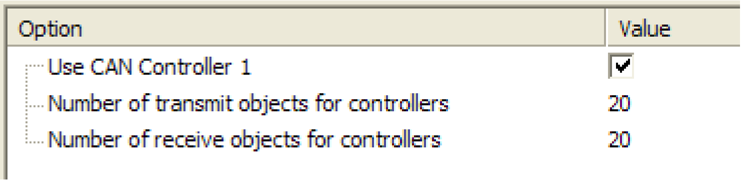
You can use the Configuration Wizard to select the parameters as shown in the picture below.

File **CAN_SAM7X.c** edited with Configuration Wizard:



Option	Value
Master Clock Frequency (MCK)	47923200

File **CAN_Cfg.h** edited with Configuration Wizard:



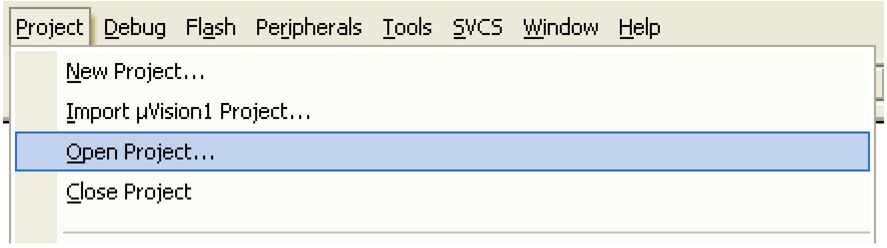
Option	Value
Use CAN Controller 1	☑
Number of transmit objects for controllers	20
Number of receive objects for controllers	20

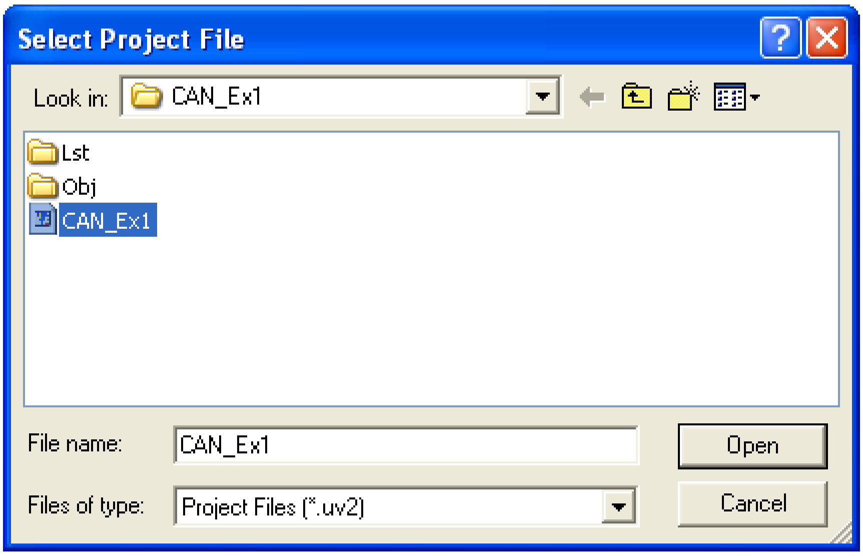
Getting Started

To run this RL-CAN AT91SAM7X example on a board, ensure you have:

- an AT91SAM7X-EK Evaluation board from Atmel
- a ULINK® USB Interface Adapter from Keil
- a USB cable for 5 V power supply.

1. Copy the example from **\Keil\ARM\Boards\Atmel\AT91SAM7X-EK\RL\CAN\CAN_Ex1** or from **\Keil\ARM\Boards\Atmel\AT91SAM7X-EK\RL\CAN\CAN_Ex2** to any directory you want to use.
2. Load the project CAN_Ex1.uv2 or CAN_Ex2.uv2 file from the copied directory into µVision® 3 IDE (**Project — Open Project...**)

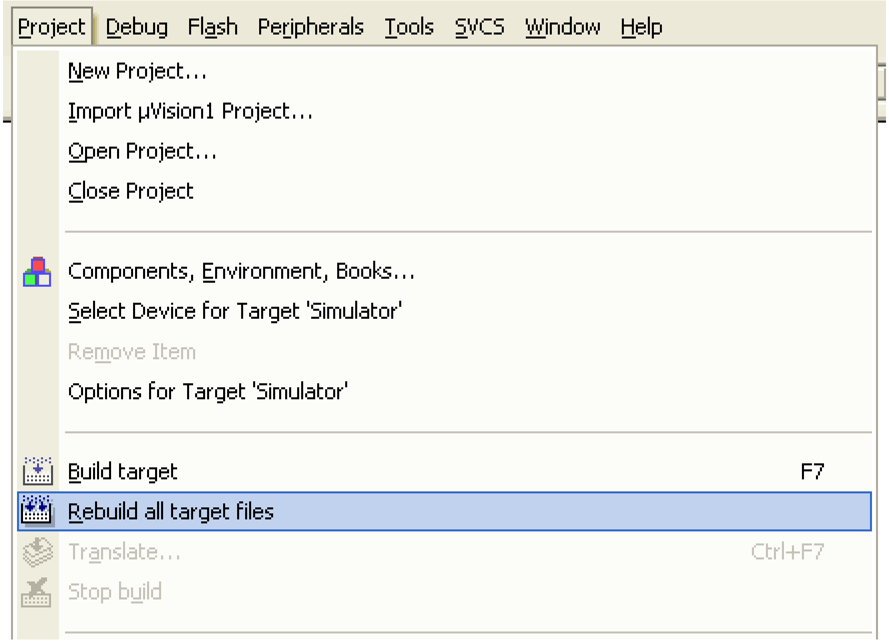




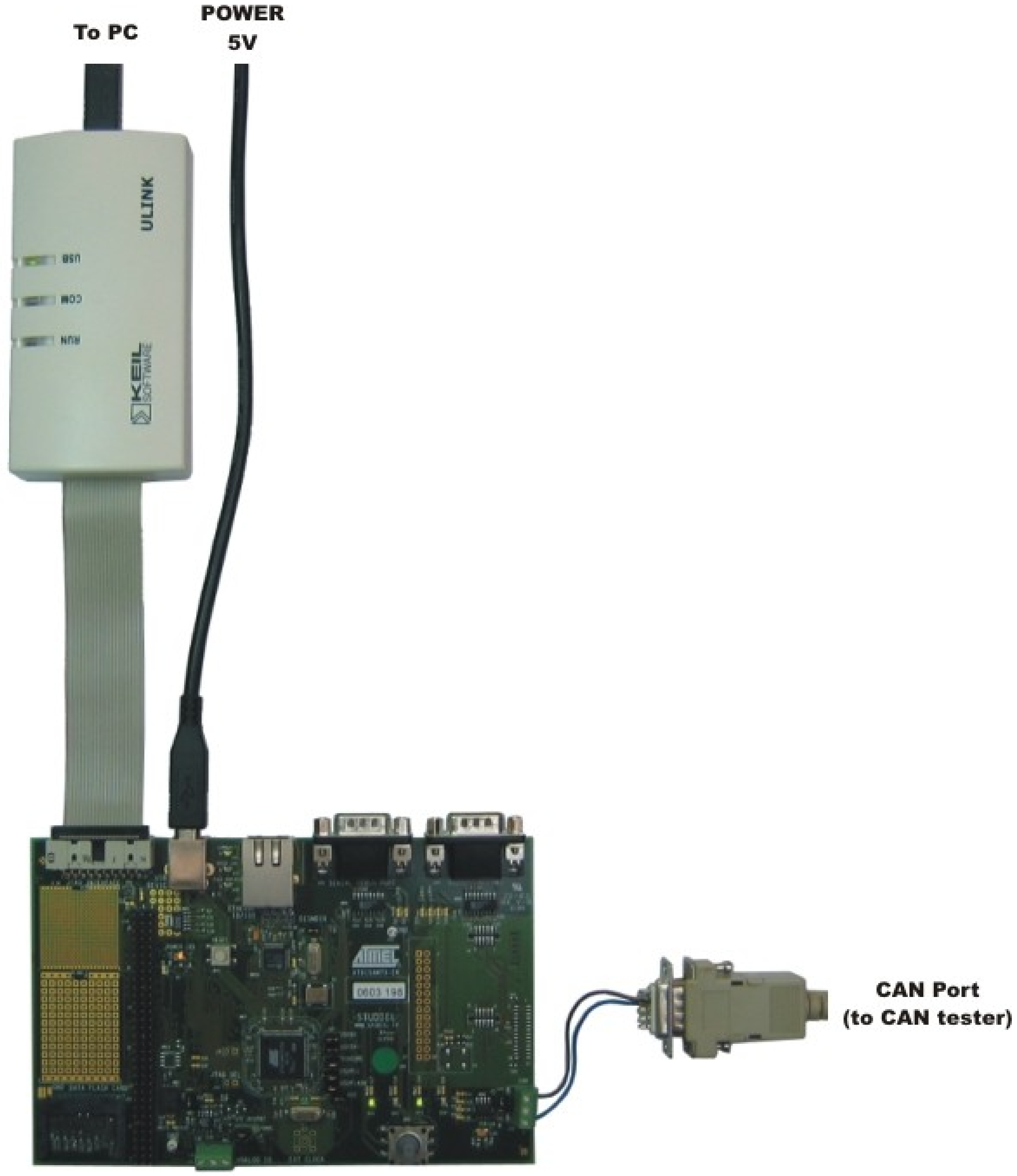
File name: CAN_Ex1

Files of type: Project Files (*.uv2)

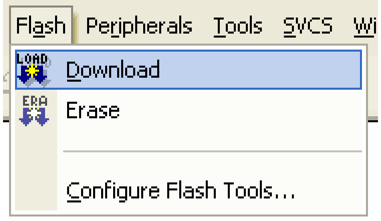
3. Rebuild the executable file from source files. Click on **Project — Rebuild all target files** on the menu or click on the toolbar button .



4. Connect the ULINK device to the PC's USB port and to the JTAG Connector on the AT91SAM7X-EK board.



5. Power-up the AT91SAM7X board by connecting the board's USB power input to the PC's USB port.
6. Click on **Flash — Download** on the menu to download the executable file to the SAM7X flash target on the AT91SAM7X-EK board.



7. The AT91SAM7X microcontroller has only one CAN port, and it does not have loopback capability. Hence, to demonstrate the functionality of the example, you must connect another CAN device that can send and receive CAN messages from this AT91SAM7X-EK

board. After connecting another CAN device, move the joystick on the AT91SAM7X-EK board and see what CAN message has been sent. You can also see the state of the 4 most significant bits of the received message in the 4 LEDs.

Initialization

On its first invocation, the **CAN_init** function does two things:

- It initializes the hardware of all CAN controllers. This includes setting up dedicated pin functions for CAN transmit and receive and assigning interrupt routine addresses and priorities to the CAN transmit and receive interrupts according to the parameters in **CAN_CFG.H**.
- It initializes the memory pool used for CAN transmit and receive message buffers.

  The size of the memory pool is determined by constants in **RTX_CAN.H** and **CAN_CFG.H**. These parameters are constants CAN_CTRL_MAX_NUM, CAN_No_SendObjects and CAN_No_ReceiveObjects.

  CAN_CTRL_MAX_NUM is maximum index of the hardware CAN controller used, this constant is set automatically by preprocessor inside RTX_CAN.H file.

  CAN_No_SendObjects is the number of messages available for usage in the transmit software message FIFO buffer.

  CAN_No_ReceiveObjects is the number of messages available for usage in the receive software message FIFO buffer.

  The memory pool uses (CAN_CTRL_MAX_NUM * (CAN_No_SendObjects + CAN_No_ReceiveObjects) * 4 + 3) * 4 of bytes of memory.

Note

- Mailbox size is specified in the **CAN_CFG.H** file using the **CAN_No_SendObjects** and **CAN_No_ReceiveObjects** manifest constants.

Example Projects

Example Projects for the RL-CAN are in the folder **\Keil\ARM\Boards\<vendor>\<board>\RL\CAN\**.

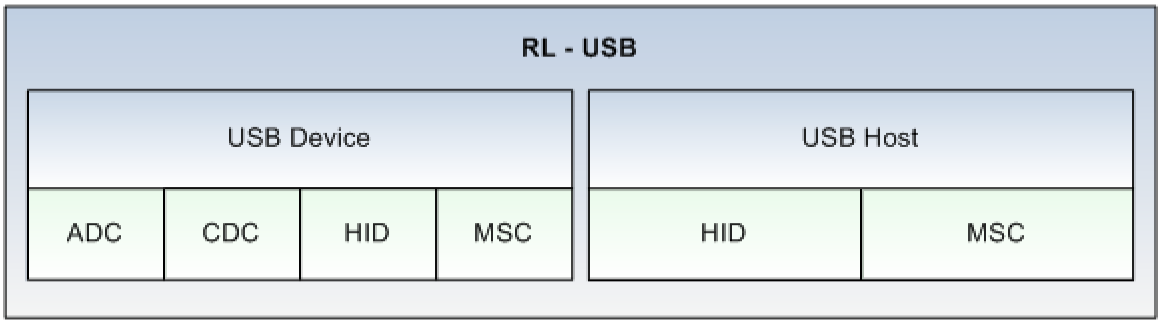
Project	Description
CAN_Ex1	Shows CAN message sending and receiving with different fixed CAN message IDs.
CAN_Ex2	Shows CAN message sending and receiving with remote frames and the usage of CAN message masks.

RL-USB

RL-USB describes the **RL-USB Library** designed to create **USB Device** and **USB Host** applications. **RL-USB** is integrated in the Real-Time Library (RL-ARM).

The **RL-USB Library** offers configurable functions to quickly design an application for a USB Device or USB Host. The library handles the low-level USB requests without the need to write the hardware layer code. Developers can focus on the application's request rather than concentrating on the specialties of the USB protocol.

From an application point of view, the library can be split into two parts, each supporting specific characteristics: RL-USB Device Features and RL-USB Host Features.



The following chapters are included:

RL-USB for USB Device Applications

Describes the library features, software stack, the source files, and device functions. Explains how to create a USB Device applications with the RL-USB Library.

RL-USB for USB Host Applications

Describes the library features, software stack, the source files, and host functions. Explains how to create a USB Host applications with the RL-USB Library.

USB Concepts

Gives a general introduction to the USB network, protocol, communication, and descriptors. This chapter is not strictly related to the RL-USB Library.

RL-USB for USB Device Applications

This chapter contains the following sections:

RL-USB Device Library

Describes the RL-USB Library features, software stack, source files, and functions when designing a USB Device application.

Create USB Device Applications

Describes the steps to create various applications for USB Devices using the RL-USB Library.

Test USB Device Applications

Lists utilities to test applications for USB Devices.

RL-USB Device Library

The **RL-USB Library** for USB Devices is an easy-to-use software collection providing a common API for a broad range of USB Device controllers. The RL-USB Library can be used standalone or with the RTX-RTOS. The library uses the native device drivers of the Windows operating system and supports ADC, CDC, HID, MSC, and composite devices.

This chapter contains the sections:

RL-USB Device Features

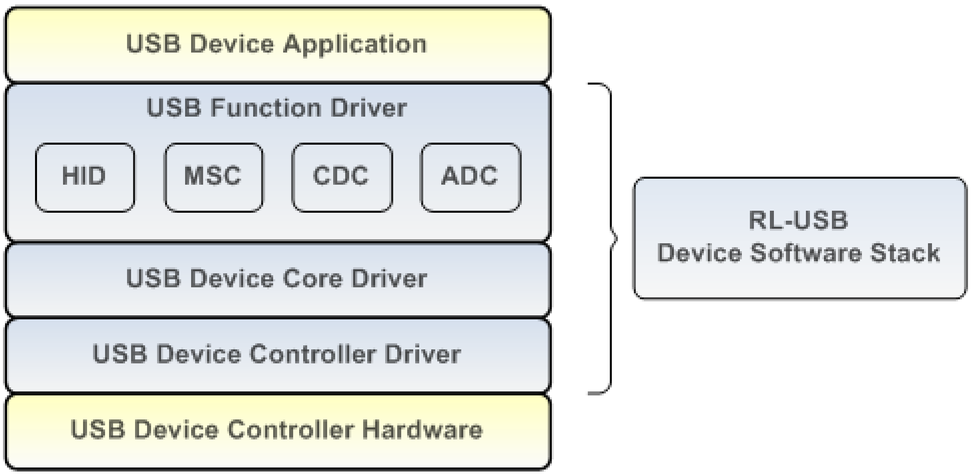
The RL-USB Library enables the developer to create USB Device applications that:

- Comply with the USB 2.0 specification (High-Speed USB).
- Comply with USB 1.1 (Low-Speed and Full-Speed USB) specification.
- Work with the RTX-RTOS.
- Support control, interrupt, bulk, and isochronous endpoints.
- Support composite USB devices.
- Support the device classes:
  - Audio Device (ADC) - to exchange streaming audio data.
  - Communication Device (CDC) - to realize a virtual COM port.
  - Human Interface Device (HID) - to exchange control and configuration data.
  - Mass Storage Device (MSC) - to store and access data.

RL-USB Device Software Stack

The **Software Stack** of the RL-USB Device Library consists of several layers:

- USB Function Driver
- USB Device Core Driver
- USB Device Controller Driver



The **USB Function Driver** is a hardware independent layer containing USB functions that are class specific and application dependent. The following USB classes are supported:

- Audio Device Class (ADC)
- Communication Device Class (CDC)
- Human Interface Device Class (HID)
- Mass Storage Device Class (MSC)

The **USB Device Core Driver** is a hardware independent layer and contains functions that implement the USB device core. This layer is an interface between the USB Device Controller Driver and the USB Function Driver layer.

The **USB Device Controller Driver** is the interface between the USB Device Controller Hardware and the USB Device Core Driver. It contains routines to read and write to the USB Device Controller Hardware.

RL-USB Device Functions

RL-USB Device Functions can be used for creating USB Device applications. The functions are categorized into **global functions** and ***class specific* functions.**

- **Global functions**.

  These RL-USB functions initialize and connect a USB Device. Use these functions prior to using any other class specific function.

Function Name	Description
usbd_init	Initializes the USB Device core and controller hardware. Use this function before using any other USB Device function. No code changes required.
usbd_connect	Connects or disconnects the USB Device. No code changes required.
usbd_reset_core	Resets the USB Device core. No code changes required.

- **Audio Device Class (ADC) functions**.

  Use the following class specific functions to customize audio device functionality. Adapt these functions in the file **usbd_user_adc.c**.

Function Name	Description
usbd_adc_init	Initializes the USB Device user audio functionality. Modify the code to the application needs.
	Custom functions to provide required USB Audio Device functionality. (For example timer interrupt routine to output audio samples on speaker)

- **Communication Device Class (CDC) functions**.

  Use these class specific functions to customize a serial communication device functionality. Adapt these functions in the file **usbd_user_cdc.c**.

Function Name	Description
usbd_cdc_init	Initializes the USB Device user communication functionality. Modify the code to the application needs.
usbd_cdc_ser_availchar	Checks whether data are available at the serial interface. Modify the code to the application needs.
usbd_cdc_ser_closeport	Closes the serial communication port.
usbd_cdc_ser_initport	Initializes the data structures and serial port. Modify the code to the application needs.
usbd_cdc_ser_linestate	Checks the line-state of the port. Modify the code to the application needs.
usbd_cdc_ser_openport	Opens the serial communication port.
usbd_cdc_ser_read	Reads data from the serial port. Modify the code to the application needs.
usbd_cdc_ser_write	Writes data to the serial port. Modify the code to the application needs.
	Custom functions to provide required USB Communication Device functionality. (For example serial interrupt routine to input or output characters on serial port)
usbd_vcom_chkserstate	Checks the status of the USB virtual COM port and if state changes prepares notification. No code changes required.
usbd_vcom_serial2usb	Writes data from the serial port to the USB virtual COM Port. No code changes required.
usbd_vcom_usb2serial	Writes data from the USB virtual COM port to the serial port. No code changes required.

- **Human Interface Device Class (HID) functions**.

  Use the following class specific functions to customize a human interface device functionality. Adapt these functions in the file **usbd_user_hid.c**.

Function Name	Description
usbd_hid_init	Initializes the user USB Device user HID functionality. Modify the code to the application needs.
usbd_hid_getinreport	Prepares USB HID IN report data to be sent to the USB Host. Modify the code to the application needs.
usbd_hid_setoutreport	Processes the USB HID OUT report data that was received. Modify the code to the application needs.

- **Mass Storage Device Class (MSC) functions**.

  Use the following class specific functions to customize a mass storage device functionality. Adapt these functions in the file **usbd_user_msc.c**.

Function Name	Description
usbd_msc_init	Initializes the USB Device user Mass Storage functionality. Modify the code to the application needs.
usbd_msc_read_sect	Handles the read request of a USB Mass Storage Device. Modify the code to the application needs.
usbd_msc_write_sect	Handles the write request of a USB Mass Storage Device. Modify the code to the application needs.

RL-USB Device Source Files

RL-USB Device Source Files for creating USB Device applications with the RL-USB Library can be found in the folders:

Folder Name	Description
\ARM\RV31\INC	Contains include files, header files, and configuration files.
\ARM\RV31\LIB	Contains the library files **USB_ARM_L.LIB** and **USB_CM3.LIB**.
\ARM\RL\USB\Drivers	Contains USB Device Controller Driver modules.
\ARM\Boards\*Vendor*\Board\RL\USB\Device	Contains example applications built with the RL-USB Library. Use the projects as templates to create new USB Device applications.
\ARM\Boards\*Vendor*\Board\RL\USB\Device\RTX	Contains example applications built with the RL-USB Library and the RTX-RTOS. Use the projects as templates to create new USB Device applications.

RL-USB Device files in **\ARM\RV31\INC**:

File Name	File Type	Layer	Description
usb_lib.c	Module	All layers	Packs configuration settings and provides them to the library. The file is included from the module file USB_CONFIG.C. No code changes are required.
rl_usb.h	Header	All layers	Contains prototypes of functions exported from the library or imported to the library. No code changes are required.
usb.h	Header	Core Driver	Main RL-USB header file used to include other USB header files.
usb_adc.h	Header	Core Driver	Contains definitions and structures used by the audio device class such as endpoint, subclasses, audio data formats, audio processing unit, request codes, ...
usb_cdc.h	Header	Core Driver	Contains definitions and structures used by the communication device class such as endpoint, subclasses, sub-types, request codes, ...
usb_hid.h	Header	Core Driver	Contains definitions and structures used by the human interface device class such as subclasses, protocol codes, descriptor types, reports, ...
usb_msc.h	Header	Core Driver	Contains definitions and structures used by the mass storage device class such as subclasses, protocol codes, request codes, SCSCI commands, ...
usb_def.h	Header	Core Driver	Contains definitions for device classes, descriptors, pipes.
usbd.h	Header	Core Driver	Contains variable declarations used by the library.
usbd_desc.h	Header	Core Driver	Contains USB Device descriptor macros.
usbd_event.h	Header	Core Driver	Contains USB Device event definitions and callback function prototypes.
usbd_hw.h	Header	Device Controller Driver	Contains USB Device hardware driver function prototypes.

RL-USB Device Library files in **\ARM\RV31\LIB**:

File Name	File Type	Layer	Description
USB_ARM_L.lib	Library	All layers	RL-USB library for ARM7 and ARM9 devices - Little Endian.
USB_CM3.lib	Library	All layers	RL-USB library for Cortex-M3 devices - Little Endian.

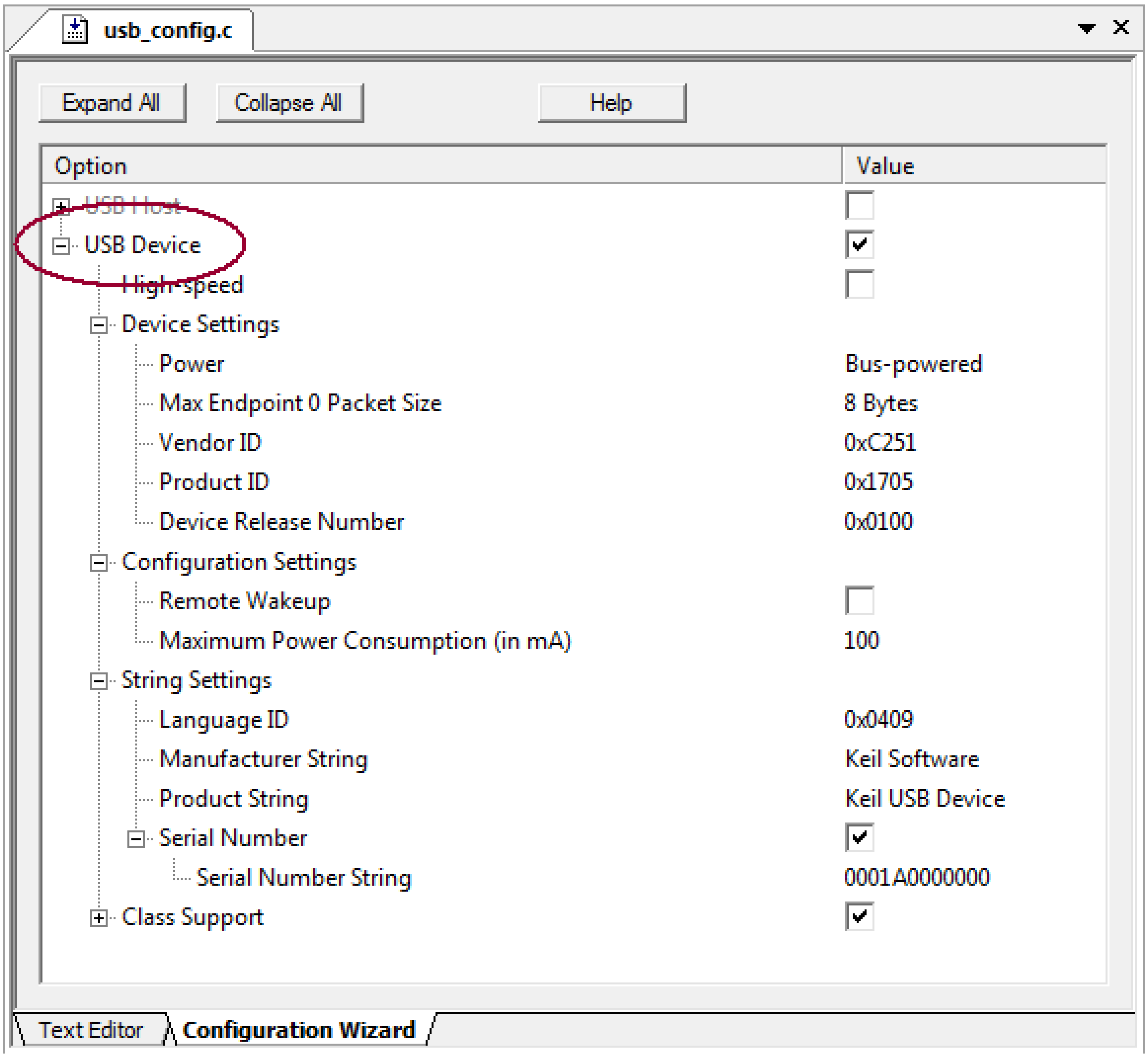
RL-USB Device Application files in **\ARM\Boards\*Vendor*\Board\RL\USB\Device** or **\ARM\Boards\Vendor\Board\RL\USB\Device\RTX:**

File Name	File Type	Layer	Description
usbd_user_*xxx*.c	Module	Application	Contains the user code for appropriate class handling.
usb_config.c	Module	All layers	Configures the USB Device Controller and USB

File Name	File Type	Layer	Description
			Device Classes at compile time. Settings for the USB Device Controller Driver and USB Device Class Drivers are available. Modify this file to suit the application requirements.
usbd_*device family*.c	Module	Device Controller Driver	Provides the hardware specific driver functions. Standardized modules are available in the folder **\ARM\RL\USB\Drivers**. Modify this file, if no module is provided for the used device.

RL-USB Device Configuration

RL-USB Device Configuration explains the options offered by the RL-USB Library for configuring USB Devices. Options are set in the file **usb_config.c** directly or using the µVision Configuration Wizard.



Option	Value
⊞ USB Host	□
⊟ USB Device	☑
High-speed	□
⊟ Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
⊟ Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
⊟ String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
⊟ Serial Number	☑
Serial Number String	0001A0000000
⊞ Class Support	☑

Where

- **USB Device** - enables the USB Device functionality. This option corresponds to *#define USBD_ENABLE*.

```
#define USBD_ENABLE                   1
```

- **High Speed** - enables high speed support, if supported by the microcontroller. This option corresponds to *#define USBD_HS_ENABLE*.

```
#define USBD_ENABLE                   0
```

Device Settings
configure the settings of the USB Device that affect Device Descriptor.

- **Power** sets default power source, *Bus-powered* or *Self-powered*. This option corresponds to *#define USBD_POWER*.

```
#define USBD_POWER                 0                          // 0 =
Bus-powered;  1 = Self-powered;
```

- **Max Endpoint 0 Packet Size** sets the maximum packet size of endpoint 0 (bMaxPacketSize0). Values of 8, 16, 32, and 64 Bytes are possible. This option corresponds to *#define USBD_MAX_PACKET0*.

```
#define USBD_MAX_PACKET0          8                         // 8, 16, 32, and 64 Bytes
```

- **Vendor ID** sets the vendor identification number (idVendor) assigned by the USB-IF. Values between 0x0000-0xFFFF are valid. This option corresponds to *#define USBD_DEVDESC_IDVENDOR*.

```
#define USBD_DEVDESC_IDVENDOR     0xC251                    // ask USB-IF for a valid ID
```

- **Product ID** sets the product identification number (idProduct) assigned by the manufacturer. Values between 0x0000-0xFFFF are valid. This option corresponds to *#define USBD_DEVDESC_IDPRODUCT*.

```
#define USBD_DEVDESC_IDPRODUCT    0x1705                    // ask the manufacturer for a valid ID
```

- **Device Release Number** sets the product release number (bcdDevice) assigned by the manufacturer. Values between 0x0000-0xFFFF are valid. This option corresponds to *#define USBD_DEVDESC_BCDDEVICE*.

```
#define USBD_DEVDESC_BCDDEVICE    0x0100                    // ask manufacturer for release number
```

Configuration Settings
configure the settings of the USB Device that affect Configuration Descriptor.

- **Remote Wakeup** sets the wakeup attribute (D5: of bmAttributes). This option corresponds to *#define USBD_CFGDESC_BMATTRIBUTES*.

```
#define USBD_CFGDESC_BMATTRIBUTES 0xA0
```

- **Maximum Power Consumption (in mA)** sets the maximum power consumption (bMaxPower) allowed when the device is fully operational. Values between 0..510 are possible. This option corresponds to *#define USBD_CFGDESC_BMAXPOWER*.

```
#define USBD_CFGDESC_BMAXPOWER    0x32                      // value range 0..510
```

String Settings
configure the language, manufacturer name, product name, and product serial number.

- **Language ID** sets the language. Numbers between 0x0000-0xFCFF are valid. This option corresponds to *#define USBD_STRDESC_LANGID*.

```
#define USBD_STRDESC_LANGID       0x0409                    // 0x0409 - English (United States)
```

- **Manufacturer String** sets the manufacturer name. The string can have 126 characters. This option corresponds to *#define USBD_STRDESC_MAN*.

```
#define USBD_STRDESC_MAN          L"Keil Software"          // max. 126 characters
```

- **Product String** sets the product name. The string can have 126 characters. This option corresponds to *#define USBD_STRDESC_PROD*.

```
#define USBD_STRDESC_PROD         L"Keil USB Device"        // max. 126 characters
```

- **Serial Number** enables the serial number string. This option corresponds to *#define USBD_STRDESC_SER_ENABLE*.

```
#define USBD_STRDESC_SER_ENABLE   1                         // 1=enabled; 0=disabled;
```

- **Serial Number String** sets the serial number. This option corresponds to *#define USBD_STRDESC_SER*.

```
#define USBD_STRDESC_SER          L"0001A0000000"           // max. 126
```

```
    characters
```

Class Support
enables class specific requests. This option corresponds to *#define USBD_CLASS_ENABLE*.

```
#define USBD_CLASS_ENABLE             1                            // 1=enabled;
0=disabled;
```

The USB Device class options are explained in:

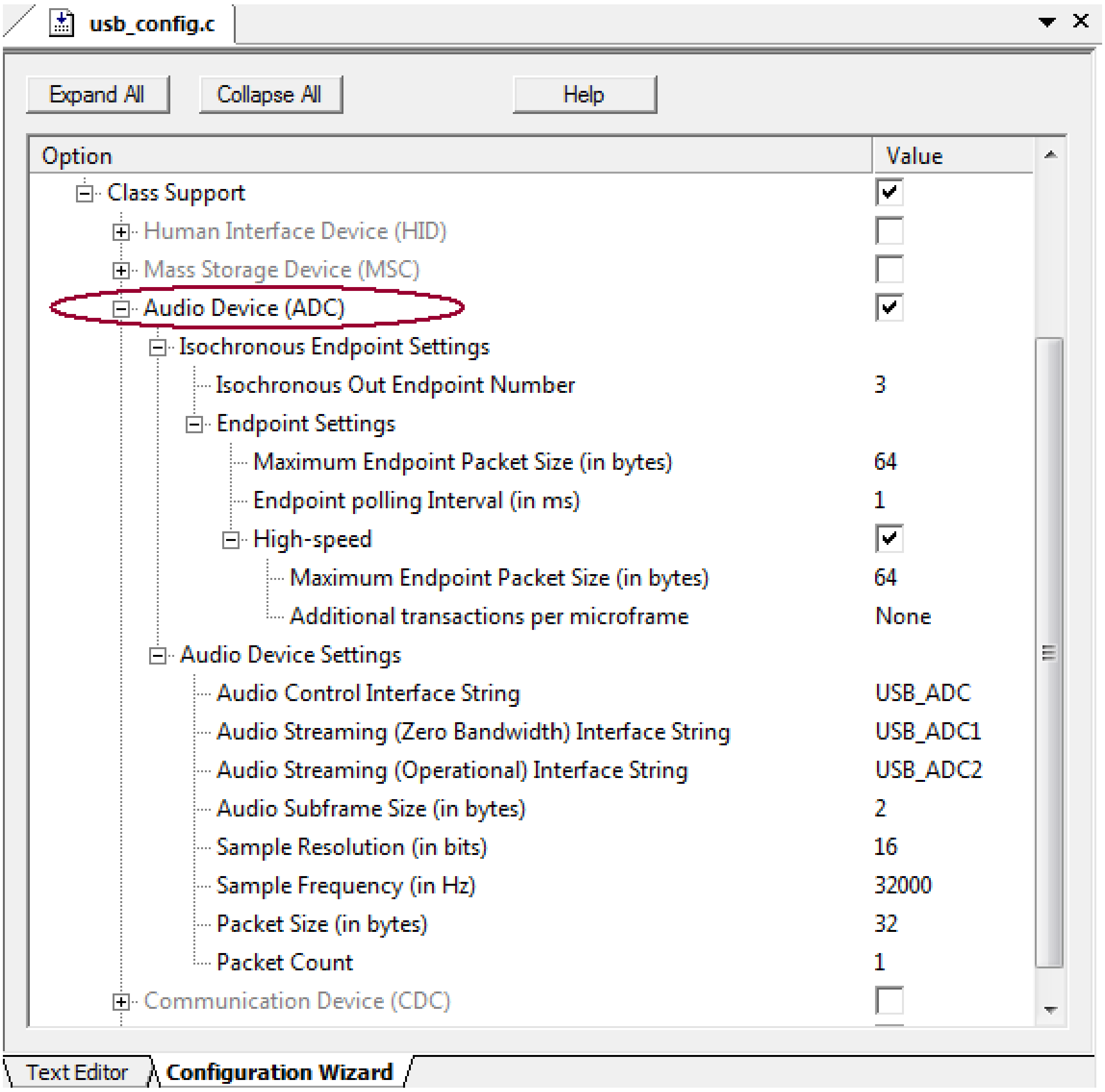
- Audio Device (ADC) Options
- Communication Device (CDC) Options
- Human Interface Device (HID) Options
- Mass Storage Device (MSC) Options
- **Vendor Specific Device** enables vendor specific requests. This category is optional and might not be available for every microcontroller.

Note

- Refer to Create USB Device Applications to build an USB Device application.

Audio Device (ADC) Options

Audio Device (ADC) Options explains the configuration options for an audio device. The options can be edited in the file **usb_config.c** directly or using the µVision Configuration Wizard.



Option	Value
⊟ Class Support	☑
⊞ Human Interface Device (HID)	□
⊞ Mass Storage Device (MSC)	□
⊟ Audio Device (ADC)	☑
⊟ Isochronous Endpoint Settings	
Isochronous Out Endpoint Number	3
⊟ Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	64
Endpoint polling Interval (in ms)	1
⊟ High-speed	☑
Maximum Endpoint Packet Size (in bytes)	64
Additional transactions per microframe	None
⊟ Audio Device Settings	
Audio Control Interface String	USB_ADC
Audio Streaming (Zero Bandwidth) Interface String	USB_ADC1
Audio Streaming (Operational) Interface String	USB_ADC2
Audio Subframe Size (in bytes)	2
Sample Resolution (in bits)	16
Sample Frequency (in Hz)	32000
Packet Size (in bytes)	32
Packet Count	1
⊞ Communication Device (CDC)	□

Where

- **Audio Device (ADC)** - enables the USB audio functionality of the device. This option corresponds to *#define USBD_ADC_ENABLE*.

```
#define USBD_ADC_ENABLE                 1                    // 1=enabled; 0=disabled
```

Isochronous Endpoint Settings
configure the endpoint characteristics and affect Endpoint Descriptor.

- **Isochronous Out Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_ADC_EP_ISOOUT*.

```
#define USBD_ADC_EP_ISOOUT           0                    // value range 1..15
```

- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size. Values between 0..1024 are allowed. This option corresponds to *#define USBD_ADC_WMAXPACKETSIZE*.

```
#define USBD_ADC_WMAXPACKETSIZE      64                   // value range 0..1024
```

- **Endpoint polling Interval (in ms)** sets the data transfer polling interval. Discrete values can be selected. This option corresponds to *#define USBD_ADC_BINTERVAL*.

```
#define USBD_ADC_BINTERVAL          1                    // value range 0..1024
```

- **High-speed** enables the characteristics when using high speed transfer rates. Enable the **High Speed** option under USB Device. This option corresponds to *#define USBD_ADC_HS_ENABLE*.

```
#define USBD_ADC_HS_ENABLE          1                    // 1=enabled; 0=disabled
```

- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size for high speed transfers. Values between 0..1024 are allowed. This option corresponds to *#define USBD_ADC_HS_WMAXPACKETSIZE*.

```
#define USBD_ADC_HS_WMAXPACKETSIZE   64                   // next option also configures this macro
```

- **Additional transactions per microframe** sets the additional transaction packets for high speed transfers. Discrete settings can be selected. This option also corresponds to *#define USBD_ADC_HS_WMAXPACKETSIZE*.

```
#define USBD_ADC_HS_WMAXPACKETSIZE   64                    // prev. option also configures this macro
```

Audio Device Settings
configure device specific options.

- **Audio Control Interface String** sets the audio control string identifier. 126 characters are allowed. This option corresponds to *#define USBD_ADC_CIF_STRDESC*.

```
#define USBD_ADC_CIF_STRDESC         L"USB_ADC"           // up to 126 characters
```

- **Audio Streaming (Zero Bandwidth) Interface String** sets the zero bandwidth streaming control string identifier. 126 characters are allowed. This option corresponds to *#define USBD_ADC_SIF1_STRDESC*.

```
#define USBD_ADC_SIF1_STRDESC        L"USB_ADC1"          // up to 126 characters
```

- **Audio Streaming (Operational) Interface String** sets the operational streaming control string identifier. 126 characters are allowed. This option corresponds to *#define USBD_ADC_SIF2_STRDESC*.

```
#define USBD_ADC_SIF2_STRDESC        L"USB_ADC2"          // up to 126 characters
```

- **Audio Subframe Size (in bytes)** sets the audio subframe size. Values between 0..255 are allowed. This option corresponds to *#define USBD_ADC_BSUBFRAMESIZE*.

```
#define USBD_ADC_BSUBFRAMESIZE       2                    // value range 0..255
```

- **Sample Resolution (in bits)** sets the bit resolution size. Values between 0..255 are allowed. This option corresponds to *#define USBD_ADC_BBITRESOLUTION*.

```
#define USBD_ADC_BBITRESOLUTION      16                   // value range 0..255
```

- **Sample Frequency (in Hz)** sets the sample frequency. Values between 0..16777215 are allowed. This option corresponds to *#define USBD_ADC_TSAMFREQ*.

```
#define USBD_ADC_TSAMFREQ            32000                // value range 0..16777215
```

- **Packet Size (in bytes)** sets the size of data packets. Values between 1..256 are allowed. This option corresponds to *#define USBD_ADC_CFG_P_S*.

```
#define USBD_ADC_CFG_P_S              32                    // value range 1..256
```

- **Packet Count** sets the number of data packets. Values between 1..16 are allowed. This option corresponds to *#define USBD_ADC_CFG_P_C*.

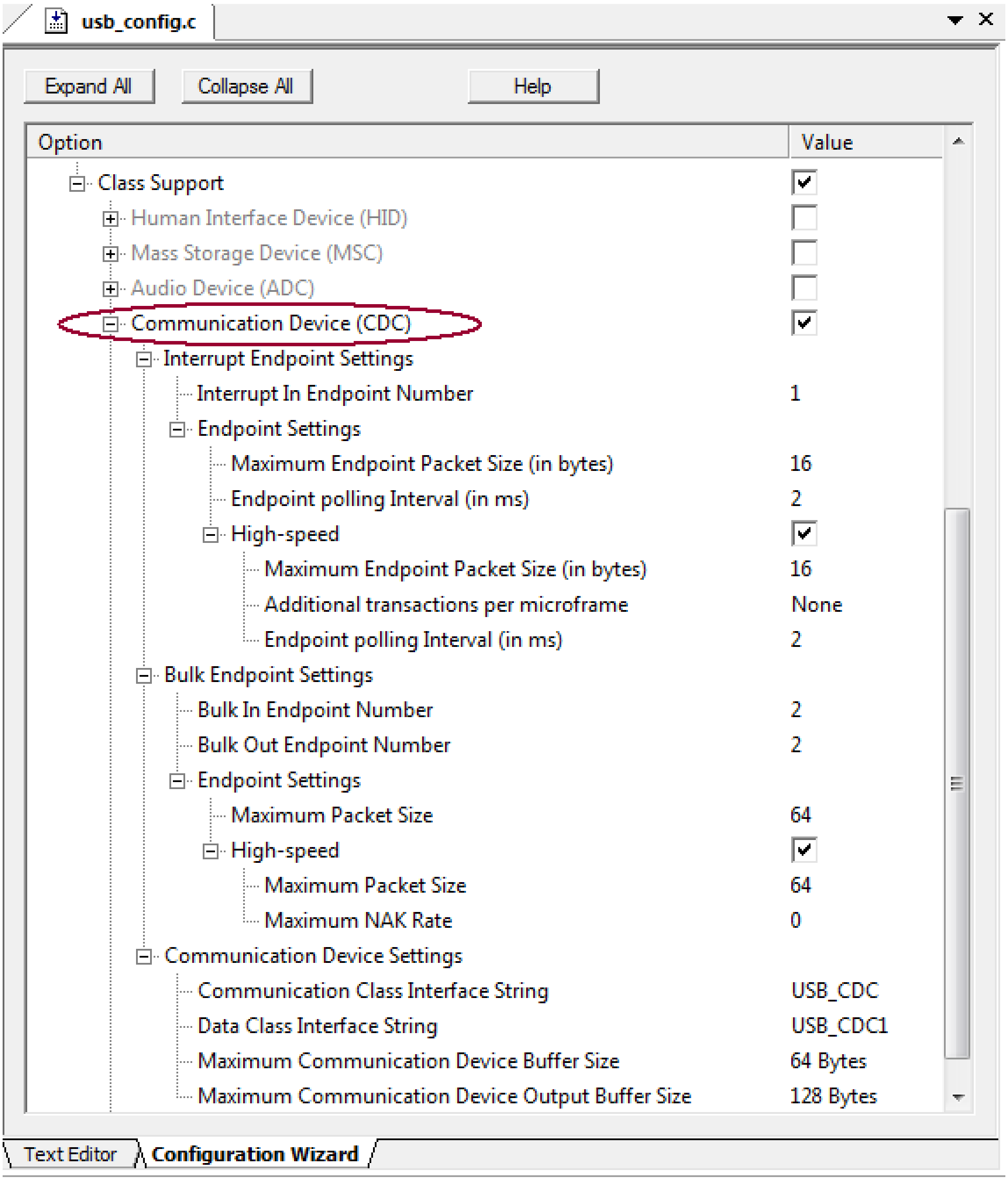
```
#define USBD_ADC_CFG_P_C              1                     // value range 1..16
```

Note

- The USB Device Controller hardware might impose restrictions on the use of endpoints.
- Refer to Create ADC Applications for a quick-start on programming an audio device.

Communication Device (CDC) Options

Communication Device (CDC) Options explains the configuration options for an communication device. The options can be edited in the file **usb_config.c** directly or using the µVision Configuration Wizard.



Option	Value
Class Support	☑
Human Interface Device (HID)	□
Mass Storage Device (MSC)	□
Audio Device (ADC)	□
Communication Device (CDC)	☑
Interrupt Endpoint Settings	
Interrupt In Endpoint Number	1
Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	16
Endpoint polling Interval (in ms)	2
High-speed	☑
Maximum Endpoint Packet Size (in bytes)	16
Additional transactions per microframe	None
Endpoint polling Interval (in ms)	2
Bulk Endpoint Settings	
Bulk In Endpoint Number	2
Bulk Out Endpoint Number	2
Endpoint Settings	
Maximum Packet Size	64
High-speed	☑
Maximum Packet Size	64
Maximum NAK Rate	0
Communication Device Settings	
Communication Class Interface String	USB_CDC
Data Class Interface String	USB_CDC1
Maximum Communication Device Buffer Size	64 Bytes
Maximum Communication Device Output Buffer Size	128 Bytes

Where

- **Communication Device (CDC)** - enables class support for communication devices. This option corresponds to *#define USBD_CDC_ENABLE*.

```
#define USBD_CDC_ENABLE                    1                          //
1=enabled; 0=disabled
```

Interrupt Endpoint Settings
configure the endpoint characteristics and affect the Endpoint Descriptor.

- **Interrupt In Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_CDC_EP_INTIN*.

```
#define USBD_CDC_EP_INTIN              1                         // value range
```

```
1..15
```

- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size. Values between 0..1024 are allowed. This option corresponds to *#define USBD_CDC_WMAXPACKETSIZE*.
-

-
  ```
  #define USBD_CDC_WMAXPACKETSIZE      16                    // value range 0..1024
  ```
- **Endpoint polling Interval (in ms)** sets the data transfer polling interval. Values from 0..255 are allowed. This option corresponds to *#define USBD_CDC_BINTERVAL*.
-

-
  ```
  #define USBD_CDC_BINTERVAL           2                     // value range 0..255
  ```
- **High-speed** enables the characteristics when using high speed transfer rates. Enable the **High Speed** option under USB Device. This option corresponds to *#define USBD_CDC_HS_ENABLE*.
-

-
  ```
  #define USBD_CDC_HS_ENABLE           1                    // 1=enabled; 0=disabled
  ```
- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size for high speed transfers. Values between 0..1024 are allowed. This option corresponds to *#define USBD_CDC_HS_WMAXPACKETSIZE*.
-

-
  ```
  #define USBD_CDC_HS_WMAXPACKETSIZE   16                   // next option also configures this macro
  ```
- **Additional transactions per microframe** sets the additional transaction packets for high speed transfers. Discrete settings can be selected. This option also corresponds to *#define USBD_CDC_HS_WMAXPACKETSIZE*.
-

-
  ```
  #define USBD_CDC_HS_WMAXPACKETSIZE   16                   // prev. option also configures this macro
  ```
- **Endpoint polling Interval (in ms)** sets the data transfer polling interval for high speed transfers. Discrete values can be selected. This option corresponds to *#define USBD_CDC_HS_BINTERVAL*.
-

-
  ```
  #define USBD_CDC_HS_BINTERVAL        2                    // value range 0..32768
  ```

Bulk Endpoint Settings
configure the endpoint characteristics and affect the Endpoint Descriptors.

- **Bulk In Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_CDC_EP_BULKIN*.
-

-
  ```
  #define USBD_CDC_EP_BULKIN           2                    // value range 1..15
  ```
- **Bulk Out Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_CDC_EP_BULKOUT*.
-

-
  ```
  #define USBD_CDC_EP_BULKOUT          2                    // value range 1..15
  ```
- **Maximum Packet Size** sets the maximal packet size, in bytes. Values between 0..1024 are allowed. This option corresponds to *#define USBD_CDC_WMAXPACKETSIZE1*.
-

-
  ```
  #define USBD_CDC_WMAXPACKETSIZE1     64                   // value range 0..1024
  ```
- **High-speed** enables the characteristics when using high speed transfer rates. Enable the **High Speed** option under USB Device. This option corresponds to *#define USBD_CDC_HS_ENABLE1*.
-

-
  ```
  #define USBD_CDC_HS_ENABLE1          1                    // 1=enabled; 0=disabled
  ```

- **Maximum Packet Size** sets the maximal packet size, in bytes, for high speed transfers. Values between 0..1024 are allowed. This option corresponds to *#define USBD_CDC_HS_WMAXPACKETSIZE1*.

```
#define USBD_CDC_HS_WMAXPACKETSIZE1  64                    // value range
0..1024
```

- **Maximum NAK Rate** sets the number for maximum not acknowledged transfers. values between 0..255 are allowed. This option also corresponds to *#define USBD_CDC_HS_BINTERVAL1*.

```
#define USBD_CDC_HS_BINTERVAL1       0                     // value range
0..255
```

Communication Device Settings
configure device specific options.

- **Communication Class Interface String** sets the communication interface string identifier. 126 characters are allowed. This option corresponds to *#define USBD_CDC_CIF_STRDESC*.

```
#define USBD_CDC_CIF_STRDESC         L"USB_CDC"            // up to 126
characters
```

- **Data Class Interface String** sets the data interface string identifier. 126 characters are allowed. This option corresponds to *#define USBD_CDC_DIF_STRDESC*.

```
#define USBD_CDC_DIF_STRDESC         L"USB_CDC1"           // up to 126
characters
```

- **Maximum Communication Device Buffer Size** sets the maximal communication buffer size, in bytes. Discrete values can be selected. This option corresponds to *#define USBD_CDC_BUFSIZE*.

```
#define USBD_CDC_BUFSIZE             64                    // possible
values: 8, 16, 32, 64, 128
```

- **Maximum Communication Device Output Buffer Size** sets the maximal data buffer size, in bytes. Discrete values can be selected. This option corresponds to *#define USBD_CDC_OUTBUFSIZE*.

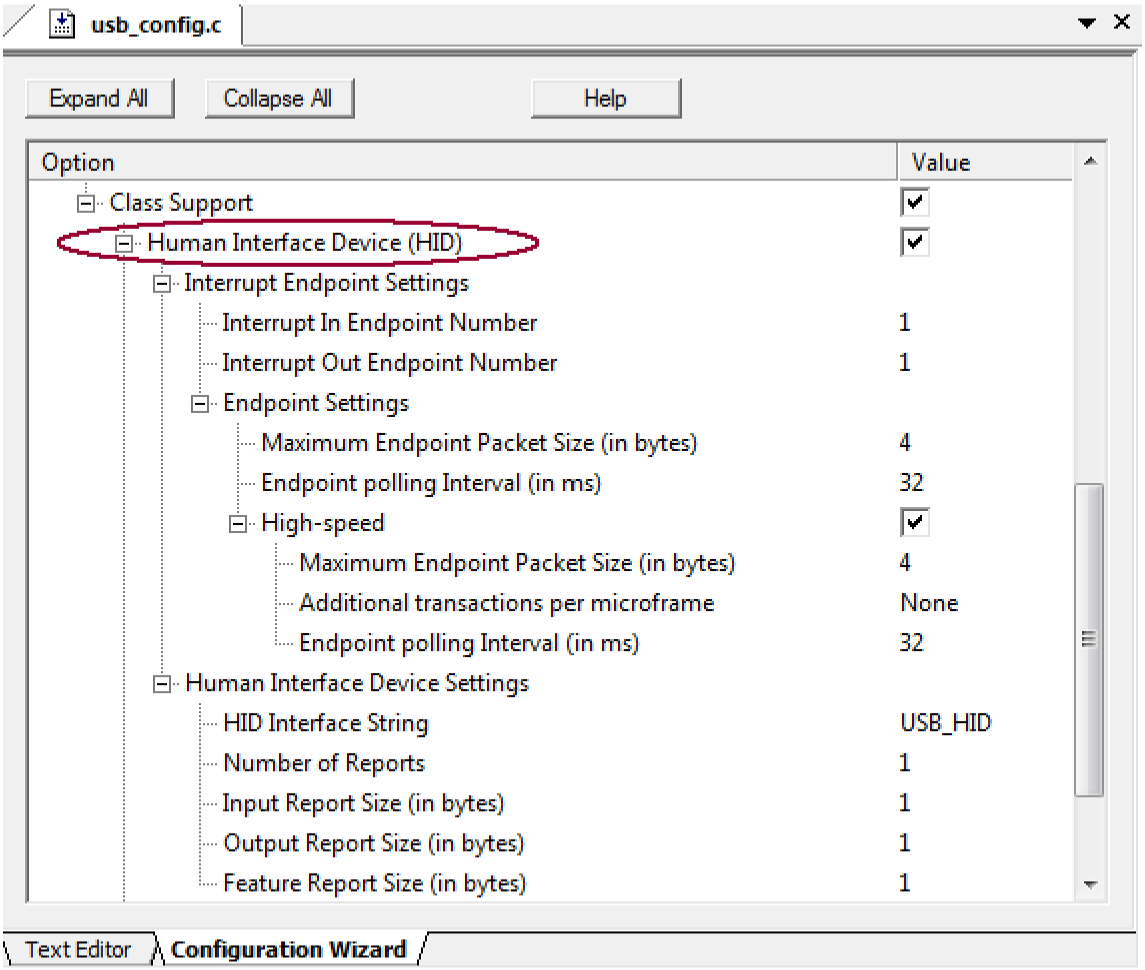
```
#define USBD_CDC_OUTBUFSIZE          128                   // possible
values: 8, 16, 32, 64, 128
```

Note

- The USB Device Controller hardware might impose restrictions on the use of endpoints.
- Refer to Create CDC Applications for a quick-start on programming a USB CDC Device.

Human Interface Device (HID) Options

Human Interface Device (HID) Options explains the configuration options for creating an human interface application. The options can be edited in the file **usb_config.c** directly or using the µVision Configuration Wizard.



Option	Value
Class Support	☑
Human Interface Device (HID)	☑
Interrupt Endpoint Settings	
Interrupt In Endpoint Number	1
Interrupt Out Endpoint Number	1
Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	4
Endpoint polling Interval (in ms)	32
High-speed	☑
Maximum Endpoint Packet Size (in bytes)	4
Additional transactions per microframe	None
Endpoint polling Interval (in ms)	32
Human Interface Device Settings	
HID Interface String	USB_HID
Number of Reports	1
Input Report Size (in bytes)	1
Output Report Size (in bytes)	1
Feature Report Size (in bytes)	1

Where

- **Human Interface Device (HID)** - enables class support for human interface devices. This option corresponds to *#define USBD_HID_ENABLE*.

```
#define USBD_HID_ENABLE               1                     // 1=enabled; 0=disabled
```

Interrupt Endpoint Settings
configure the endpoint characteristics and affect the Endpoint Descriptors.

- **Interrupt In Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_HID_EP_INTIN*.

```
#define USBD_HID_EP_INTIN             1                     // value range 1..15
```

- **Interrupt Out Endpoint Number** sets the endpoint number. Values between 0..15 are allowed; 0-disables the endpoint for messages sent by the host. This option corresponds to *#define USBD_HID_EP_INTOUT*.

```
#define USBD_HID_EP_INTOUT            1                     // value range 0..15; 0=not used;
```

- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size. Values between 0..64 are allowed. This option corresponds to *#define USBD_HID_WMAXPACKETSIZE*.

```
#define USBD_HID_WMAXPACKETSIZE       4                     // value range
```

```
0..64
```

- **Endpoint polling Interval (in ms)** sets the data transfer polling interval. Values from 0..255 are allowed. This option corresponds to *#define USBD_HID_BINTERVAL*.

```
#define USBD_HID_BINTERVAL          32                     // value range
0..255
```

- **High-speed** enables the characteristics when using high speed transfer rates. Enable the **High Speed** option under USB Device. This option corresponds to *#define USBD_HID_HS_ENABLE*.

```
#define USBD_HID_HS_ENABLE          1                      // 1=enabled;
0=disabled
```

- **Maximum Endpoint Packet Size (in bytes)** sets the maximal packet size for high speed transfers. Values between 0..1024 are allowed. This option corresponds to *#define USBD_HID_HS_WMAXPACKETSIZE*.

```
#define USBD_HID_HS_WMAXPACKETSIZE  4                      // next option
also configures this macro
```

- **Additional transactions per microframe** sets the additional transaction packets for high speed transfers. Discrete settings can be selected. This option also corresponds to *#define USBD_HID_HS_WMAXPACKETSIZE*.

```
#define USBD_HID_HS_WMAXPACKETSIZE  4                      // prev. option
also configures this macro
```

- **Endpoint polling Interval (in ms)** sets the data transfer polling interval for high speed transfers. Discrete values can be selected. This option corresponds to *#define USBD_HID_HS_BINTERVAL*.

```
#define USBD_HID_HS_BINTERVAL       6                      // value range
1..32768
```

Human Interface Device Settings
configure device specific options.

- **HID Interface String** sets the interface string identifier. 126 characters are allowed. This option corresponds to *#define USBD_HID_CIF_STRDESC*.

```
#define USBD_HID_CIF_STRDESC        L"USB_HID"             // up to 126
characters
```

- **Number of Reports** sets the amount of reports. Values from 1..255 are allowed. This option corresponds to *#define USBD_HID_CFG_REPORTNUM*.

```
#define USBD_HID_CFG_REPORTNUM      1                      // value range
0..255
```

- **Input Report Size (in bytes)** sets the size for reports sent to the host. Values from 1..64 are allowed. This option corresponds to *#define USBD_HID_INREPORT_BYTES*.

```
#define USBD_HID_INREPORT_BYTES     1                      // value range
1..64
```

- **Output Report Size (in bytes)** sets the size for reports received from the host. Values from 1..64 are allowed. This option corresponds to *#define USBD_HID_OUTREPORT_BYTES*.

```
#define USBD_HID_OUTREPORT_BYTES    1                      // value range
1..64
```

- **Feature Report Size (in bytes)** sets the size for control reports. Values from 1..64 are allowed. This option corresponds to *#define USBD_HID_FEATREPORT_BYTES*.

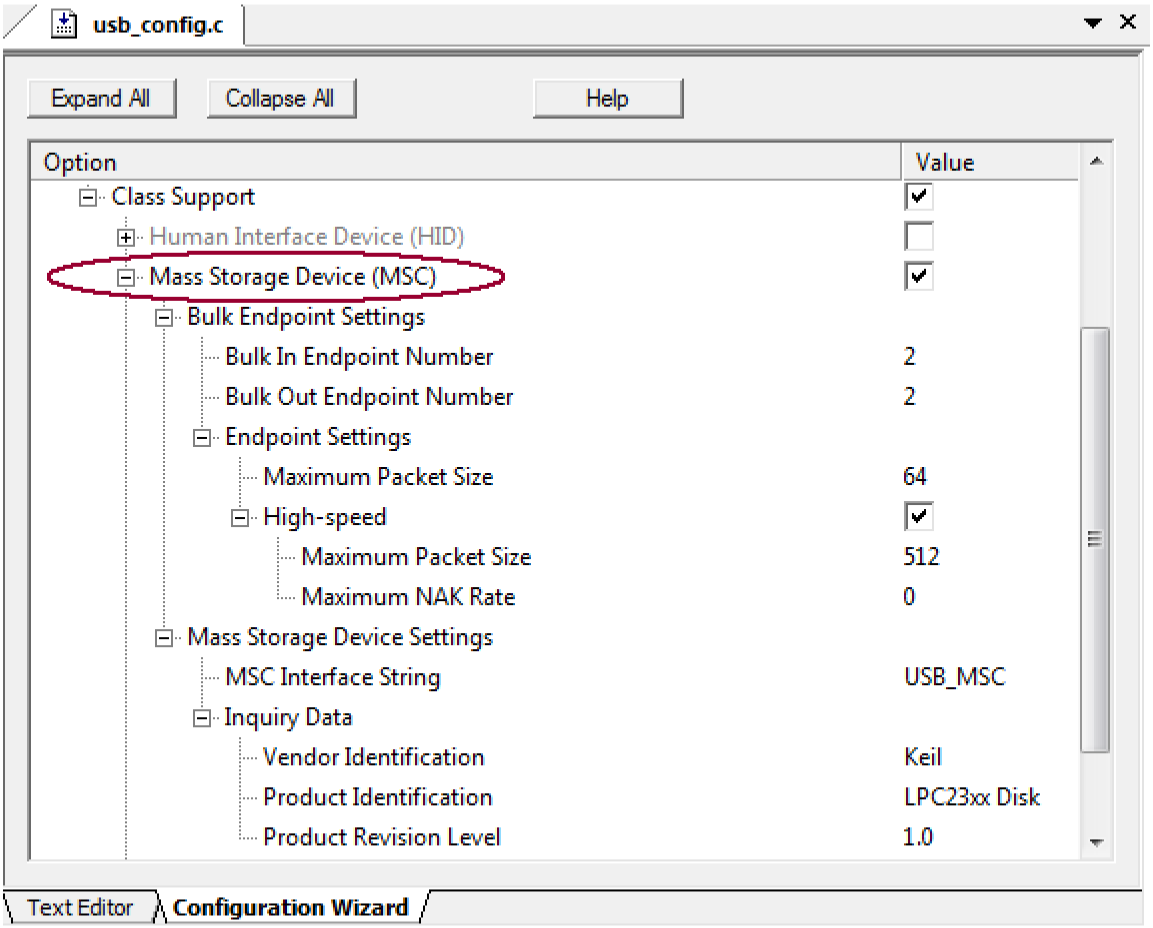
```
#define USBD_HID_FEATREPORT_BYTES   1                      // value range
1..64
```

Note

- The USB Device Controller hardware might impose restrictions on the use of endpoints.
- Refer to Create HID Applications for a quick-start on programming a USB HID Device.

Mass Storage Device (MSC) Options

Mass Storage Device (MSC) Options explains the configuration options for an mass storage device. The options can be edited in the file **usb_config.c** directly or using the µVision Configuration Wizard.



Option	Value
⊟ Class Support	☑
⊞ Human Interface Device (HID)	□
⊟ Mass Storage Device (MSC)	☑
⊟ Bulk Endpoint Settings	
Bulk In Endpoint Number	2
Bulk Out Endpoint Number	2
⊟ Endpoint Settings	
Maximum Packet Size	64
⊟ High-speed	☑
Maximum Packet Size	512
Maximum NAK Rate	0
⊟ Mass Storage Device Settings	
MSC Interface String	USB_MSC
⊟ Inquiry Data	
Vendor Identification	Keil
Product Identification	LPC23xx Disk
Product Revision Level	1.0

Where

- **Mass Storage Device (MSC)** - enables class support for mass storage devices. This option corresponds to *#define USBD_MSC_ENABLE*.

```
#define USBD_MSC_ENABLE                1                        // 1=enabled; 0=disabled
```

Bulk Endpoint Settings
configure the endpoint characteristics and affect the Endpoint Descriptors.

- **Bulk In Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_MSC_EP_BULKIN*.

```
#define USBD_MSC_EP_BULKIN          2                     // value range 1..15
```

- **Bulk Out Endpoint Number** sets the endpoint number. Values between 1..15 are allowed. This option corresponds to *#define USBD_MSC_EP_BULKOUT*.

```
#define USBD_MSC_EP_BULKOUT         2                     // value range 1..15
```

- **Maximum Packet Size** sets the maximal packet size, in bytes. Values between 1..1024 are allowed. This option corresponds to *#define USBD_MSC_WMAXPACKETSIZE*.

```
#define USBD_MSC_WMAXPACKETSIZE     64                     // value range 1..1024
```

- **High-speed** enables the characteristics when using high speed transfer rates. Enable the

High Speed option under USB Device. This option corresponds to *#define USBD_MSC_HS_ENABLE*.

```
#define USBD_MSC_HS_ENABLE           1                    // 1=enabled; 0=disabled
```

- **Maximum Packet Size** sets the maximal packet size for high speed transfers. Values between 1..1024 are allowed. This option corresponds to *#define USBD_MSC_HS_WMAXPACKETSIZE*.

```
#define USBD_MSC_HS_WMAXPACKETSIZE   512                  // value range 1..1024
```

- **Maximum NAK Rate** sets the maximum not acknowledge packets for high speed transfers. Values between 0..255 are allowed. This option also corresponds to *#define USBD_MSC_HS_BINTERVAL*.

```
#define USBD_MSC_HS_BINTERVAL        0                    // value range 0..255
```

Mass Storage Device Settings
configure device specific options.

- **MSC Interface String** sets the mass storage interface string identifier. 126 characters are allowed. This option corresponds to *#define USBD_MSC_STRDESC*.

```
#define USBD_MSC_STRDESC             L"USB_MSC"           // up to 126 characters
```

- **Vendor Identification** sets the vendor string identifier. 8 characters are allowed. This option corresponds to *#define USBD_MSC_INQUIRY_DATA*.

```
#define USBD_MSC_INQUIRY_DATA        L"KEIL    "          // see next option; 8 characters
```

- **Product Identification** sets the product string identifier. 16 characters are allowed. This option corresponds to *#define USBD_MSC_INQUIRY_DATA*.

```
#define USBD_MSC_INQUIRY_DATA        L"LPC23xx Disk    "  // see next+prev. option; 16 characters
```

- **Product Revision Level** sets the product revision string identifier. 4 characters are allowed. This option corresponds to *#define USBD_MSC_INQUIRY_DATA*.

```
#define USBD_MSC_INQUIRY_DATA        L"1.0 "              // see prev. option; 4 characters
```

Note

- The USB Device Controller hardware might impose restrictions on the use of endpoints.
- Refer to Create MSC Applications for a quick-start on programming a USB MSC Device.

Create USB Device Applications

Several USB Device applications can be built with the RL-USB Library. Each example demonstrates the use of another endpoint and transfer type and are located in the folder **\Keil\ARM\Boards\ *Vendor*\ *Board*\RL\USB\Device**.

Create ADC Applications

Creates an audio applications demonstrating isochronous endpoints. The evaluation board is configured as a sound card, which can be connected to a computer using a USB cable.

Create CDC Applications

Creates a serial interface device application demonstrating interrupt and bulk endpoints via a USB Virtual COM Port.

Create HID Applications

Creates an human interface application demonstrating interrupt endpoints. The µVision examples control the LEDs of the evaluation board.

Create MSC Applications

Creates a mass storage application demonstrating bulk endpoints for transferring large volumes of data between a computer and the device.

Create Composite Applications

Creates a combined application using existing HID and MSC applications.

Create ADC Applications

Create ADC Applications explains the steps to program a USB Device for audio streaming using the RL-USB Library. Audio applications control speakers, microphones, or other audio devices.

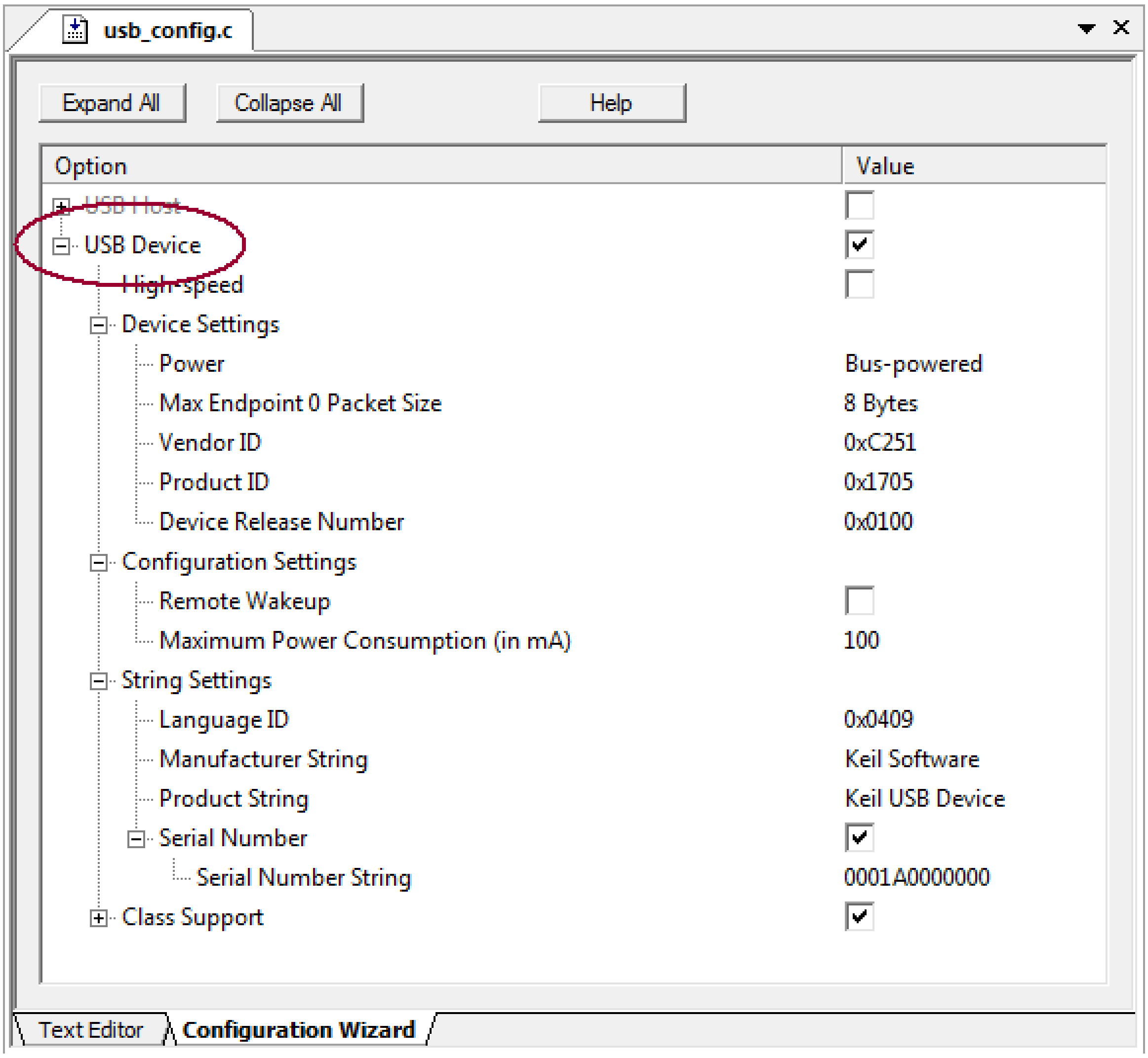
Include into the project and configure the following RL-USB Device Source Files:

1. The library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usb_config.c** - to configure the USB system.
3. **usbd_*device family*.c** - to configure the device hardware layer.
4. **usbd_user_adc.c** - to adapt the code to the application needs.
5. **USBD_Demo.c** - to initialize and connect the USB Device from *main()*.

```

..
usbd_init();
usbd_connect(__TRUE);
..
```

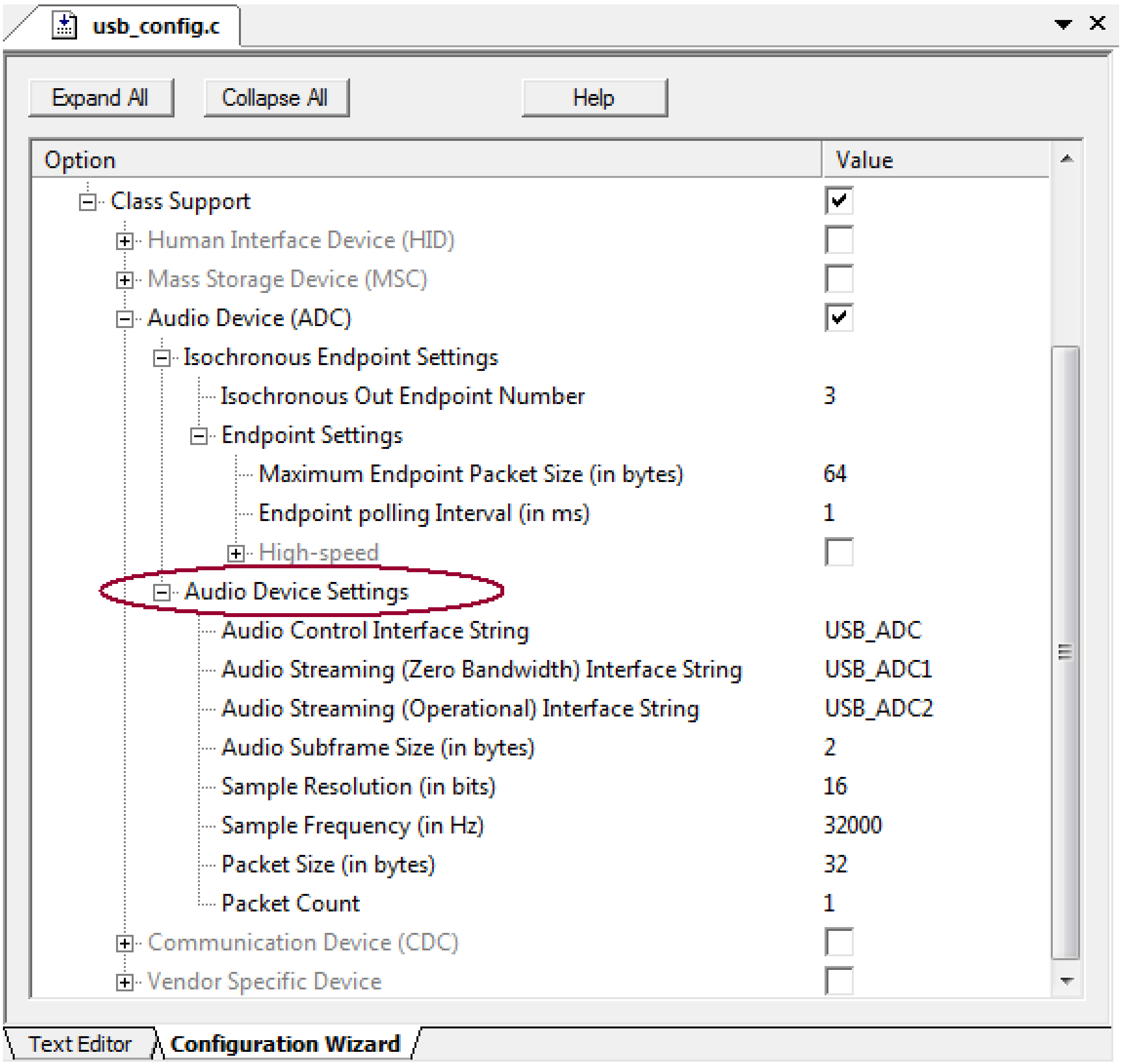
Applications can be created using existing µVision audio projects. The RL-USB audio examples have a simple function that receives an audio data stream from the host and sends it to a speaker.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\BoardName\RL\USB\Device\*Audio*** to a new folder and open the project ***.uvproj** with µVision. **RTX** projects are using the RTX-RTOS, whereas simple **Audio** projects work without an RTOS. However, the USB configuration does not differ.
2. Open the file **usb_config.c** and configure the USB device with the Configuration Wizard.
3. Enable **USB Device** and set the device characteristics.



Option	Value
⊞ ~~USB Host~~	□
⊟ USB Device	☑
High-speed	□
⊟ Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
⊟ Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
⊟ String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
⊟ Serial Number	☑
Serial Number String	0001A0000000
⊞ Class Support	☑

4. Enable **Class Support** and **Audio Device (ADC)** and set the characteristics.



Option	Value
Class Support	☑
Human Interface Device (HID)	□
Mass Storage Device (MSC)	□
Audio Device (ADC)	☑
Isochronous Endpoint Settings	
Isochronous Out Endpoint Number	3
Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	64
Endpoint polling Interval (in ms)	1
High-speed	□
Audio Device Settings	
Audio Control Interface String	USB_ADC
Audio Streaming (Zero Bandwidth) Interface String	USB_ADC1
Audio Streaming (Operational) Interface String	USB_ADC2
Audio Subframe Size (in bytes)	2
Sample Resolution (in bits)	16
Sample Frequency (in Hz)	32000
Packet Size (in bytes)	32
Packet Count	1
Communication Device (CDC)	□
Vendor Specific Device	□

5. Optionally, adapt the source files **usbd_*device family*.c** that contain hardware dependent code and must be adapted to the USB controller.
6. Modify the file **usbd_user_adc.c** to adapt the code to the application needs.

Note

- The options are explained in Audio Device (ADC) Options.

Create CDC Applications

Create CDC Applications explains how to program a USB Communication Device using the RL-USB Library. Communication applications control networks, serial and wireless communication interfaces, telecom devices, modems, printers, scanners.

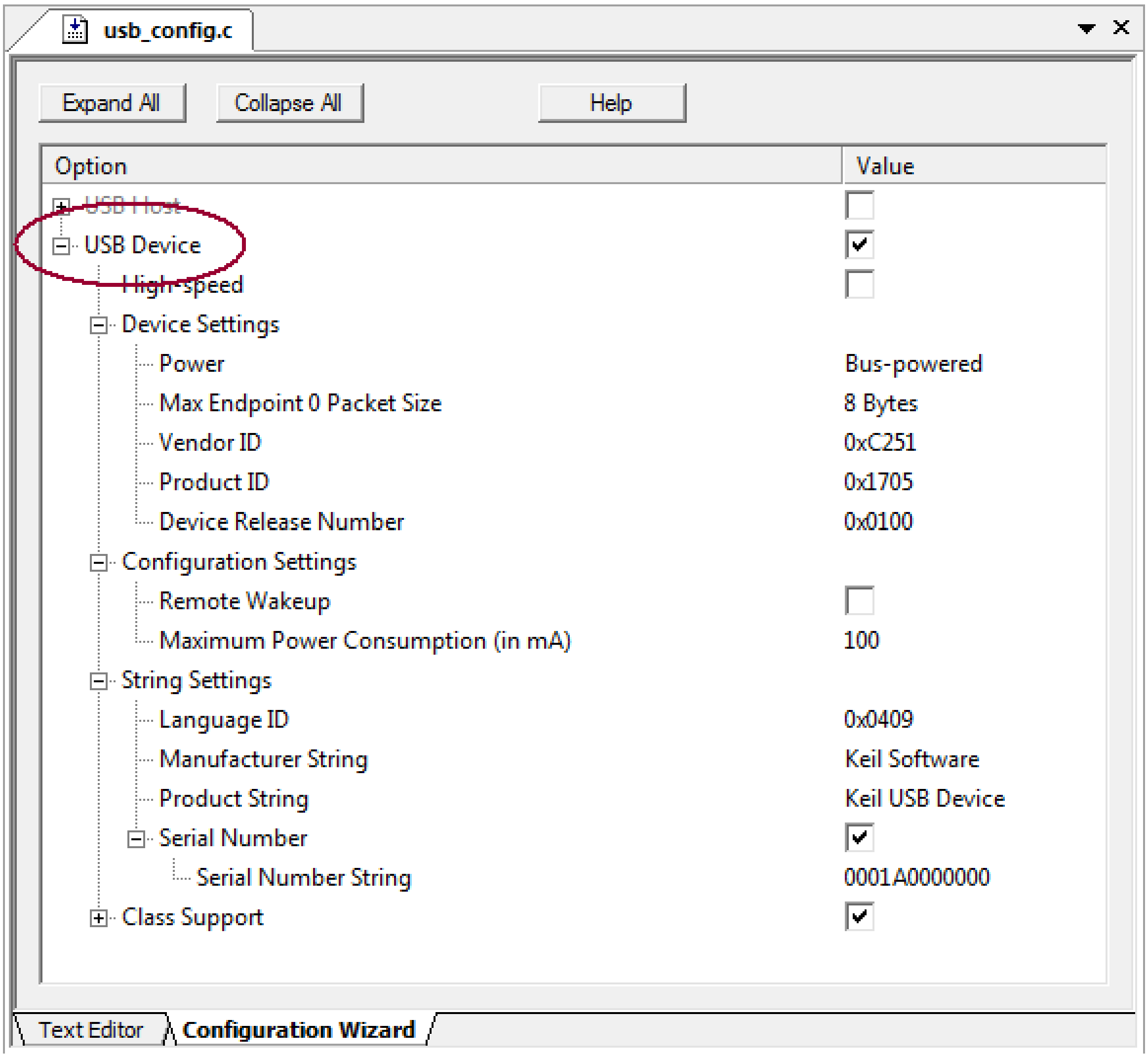
Include into the project and configure the following RL-USB Device Source Files:

1. The library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usb_config.c** - to configure the USB system.
3. ***board_name*-vcom.inf** - to configure the driver information file.
4. **usbd_*device family*.c** - to configure the device hardware layer.
5. **usbd_user_cdc.c** - to adapt the code to the application needs.
6. **USBD_Demo.c** - to initialize and connect the USB Device from *main()*.

```

..
usbd_init();                              // USB Initialization
usbd_connect(__TRUE);                     // USB Connect

while (1) {                               // Loop forever
  usbd_vcom_serial2usb();                 // write to USB VCOM Port
  usbd_vcom_chkserstate();                // check status
  usbd_vcom_usb2serial();                 // write to Serial Port
}
..
```

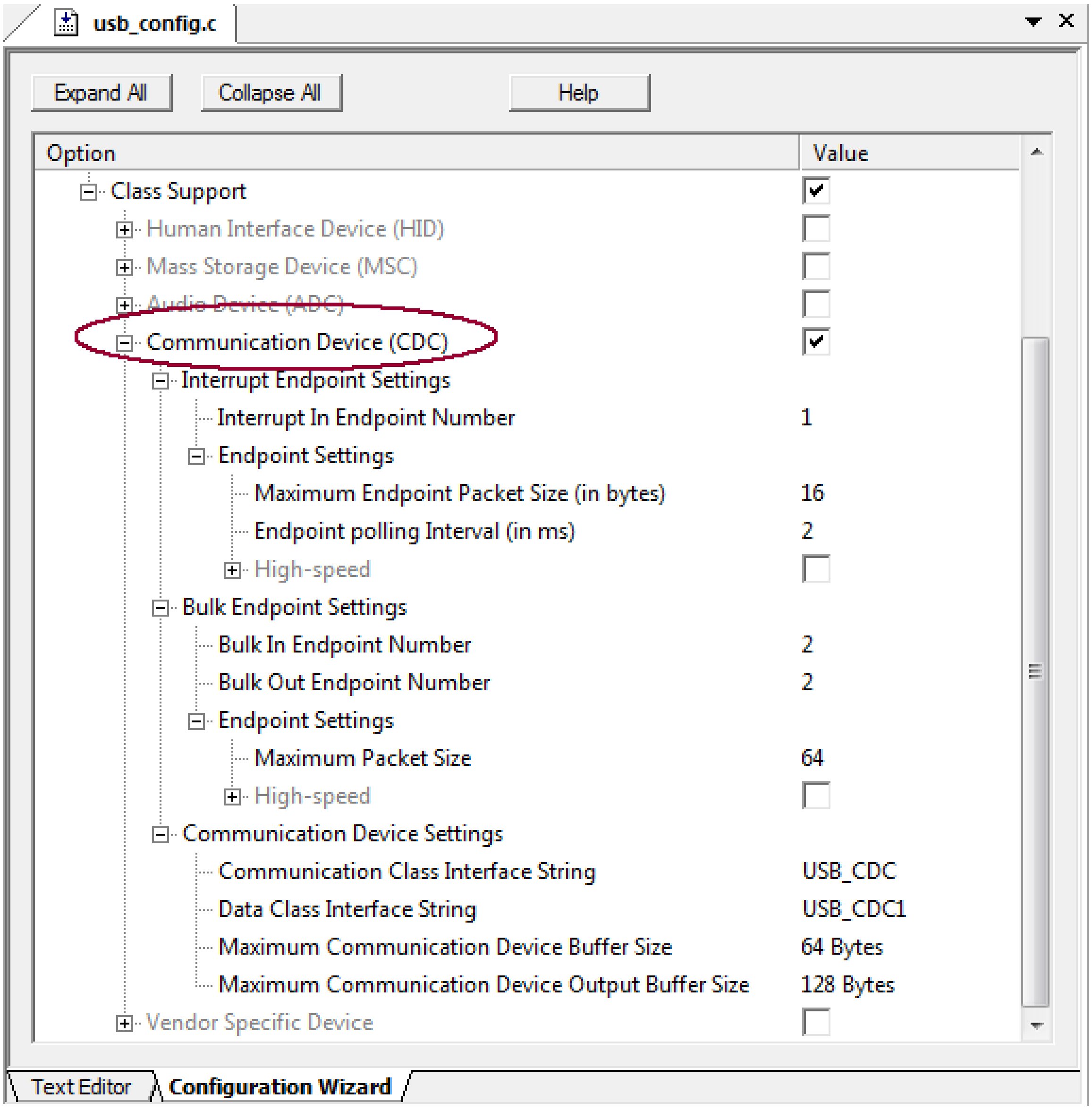
Applications can be created using existing µVision CDC projects. The RL-USB communication examples use two simple functions that send the signals from the USB virtual COM port to the PC serial COM port and vice versa.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\BoardName\RL\USB\Device\*CDC*** to a new folder and open the project ***.uvproj** with µVision. **RTX** projects are using the RTX-RTOS, whereas simple **CDC** projects work without an RTOS. However, the USB configuration does not differ.
2. Open the file **usb_config.c** and configure the USB device with the Configuration Wizard.
3. Enable **USB Device** and set the device characteristics.



Option	Value
⊞ USB Host	□
⊟ USB Device	☑
High-speed	□
⊟ Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
⊟ Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
⊟ String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
⊟ Serial Number	☑
Serial Number String	0001A0000000
⊞ Class Support	☑

4. Adapt the file ***board_name*-vcom.inf** whenever the *Vendor ID* and *Product ID* has been changed. (See picture above).
5. Enable **Class Support** and **Communication Device (CDC)** and set the characteristics.



Option	Value
Class Support	☑
Human Interface Device (HID)	□
Mass Storage Device (MSC)	□
Audio Device (ADC)	□
Communication Device (CDC)	☑
Interrupt Endpoint Settings	
Interrupt In Endpoint Number	1
Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	16
Endpoint polling Interval (in ms)	2
High-speed	□
Bulk Endpoint Settings	
Bulk In Endpoint Number	2
Bulk Out Endpoint Number	2
Endpoint Settings	
Maximum Packet Size	64
High-speed	□
Communication Device Settings	
Communication Class Interface String	USB_CDC
Data Class Interface String	USB_CDC1
Maximum Communication Device Buffer Size	64 Bytes
Maximum Communication Device Output Buffer Size	128 Bytes
Vendor Specific Device	□

6. Optionally, adapt the source file **usbd_*device family*.c** that contains hardware dependent code.
7. Modify the file **usbd_user_cdc.c** to adapt the code to the application needs.

Note

- The options are explained in Communication Device (CDC) Options.

Create HID Applications

Create HID Applications explain how to program a USB Human Interface Device using the RL-USB Library. HID applications control keyboards, mice, and other I/O devices.

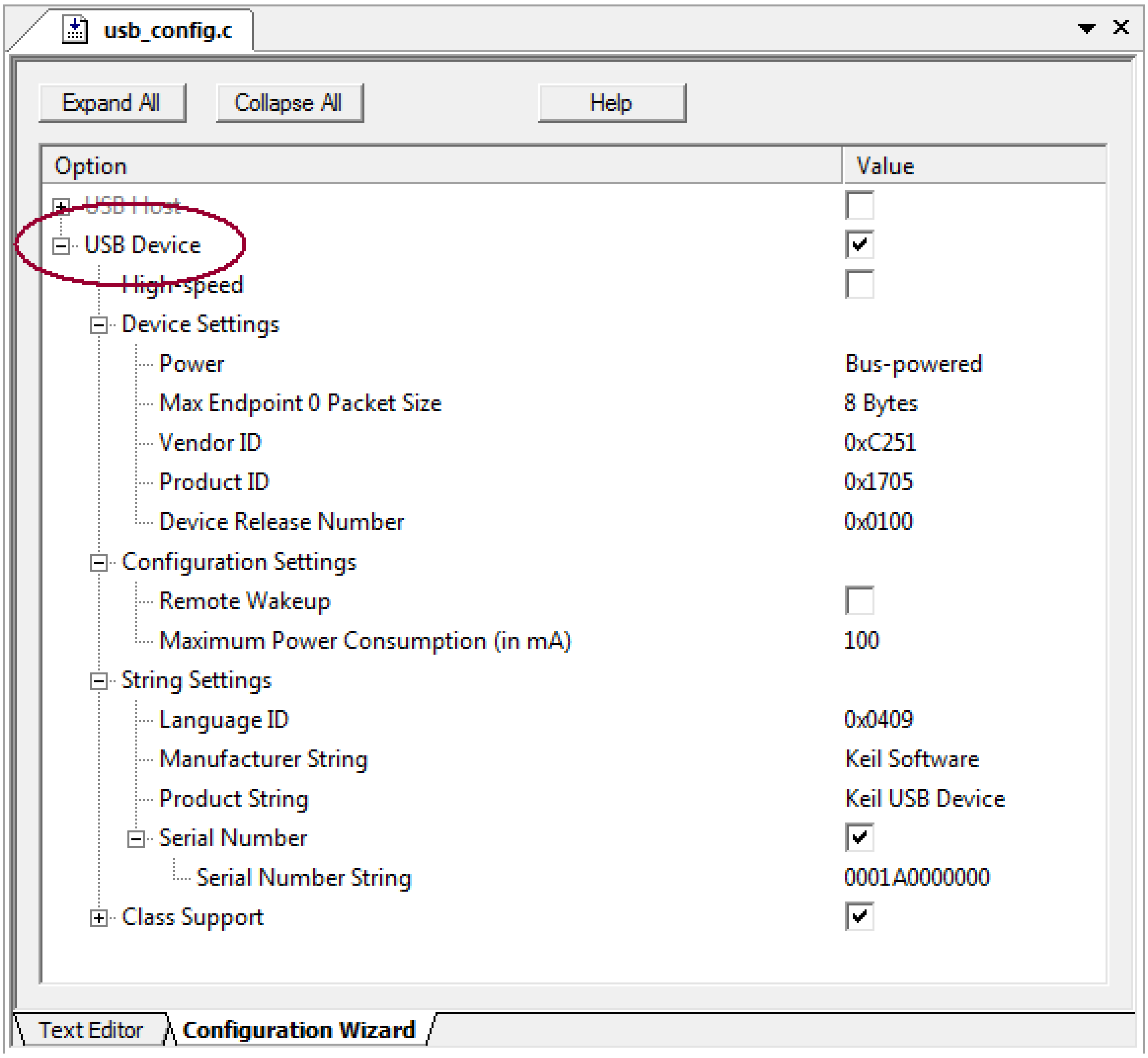
Include into the project and configure the following RL-USB Device Source Files:

1. The library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usb_config.c** - to configure the USB system.
3. **usbd_*device family*.c** - to configure the device hardware layer.
4. **usbd_user_hid.c** - to adapt the code to the application needs.
5. **USBD_Demo.c** - to initialize and connect the USB Device from *main()*.

```

..
usbd_init();                          // USB Initialization
usbd_connect(__TRUE);                 // USB Connect
..
```

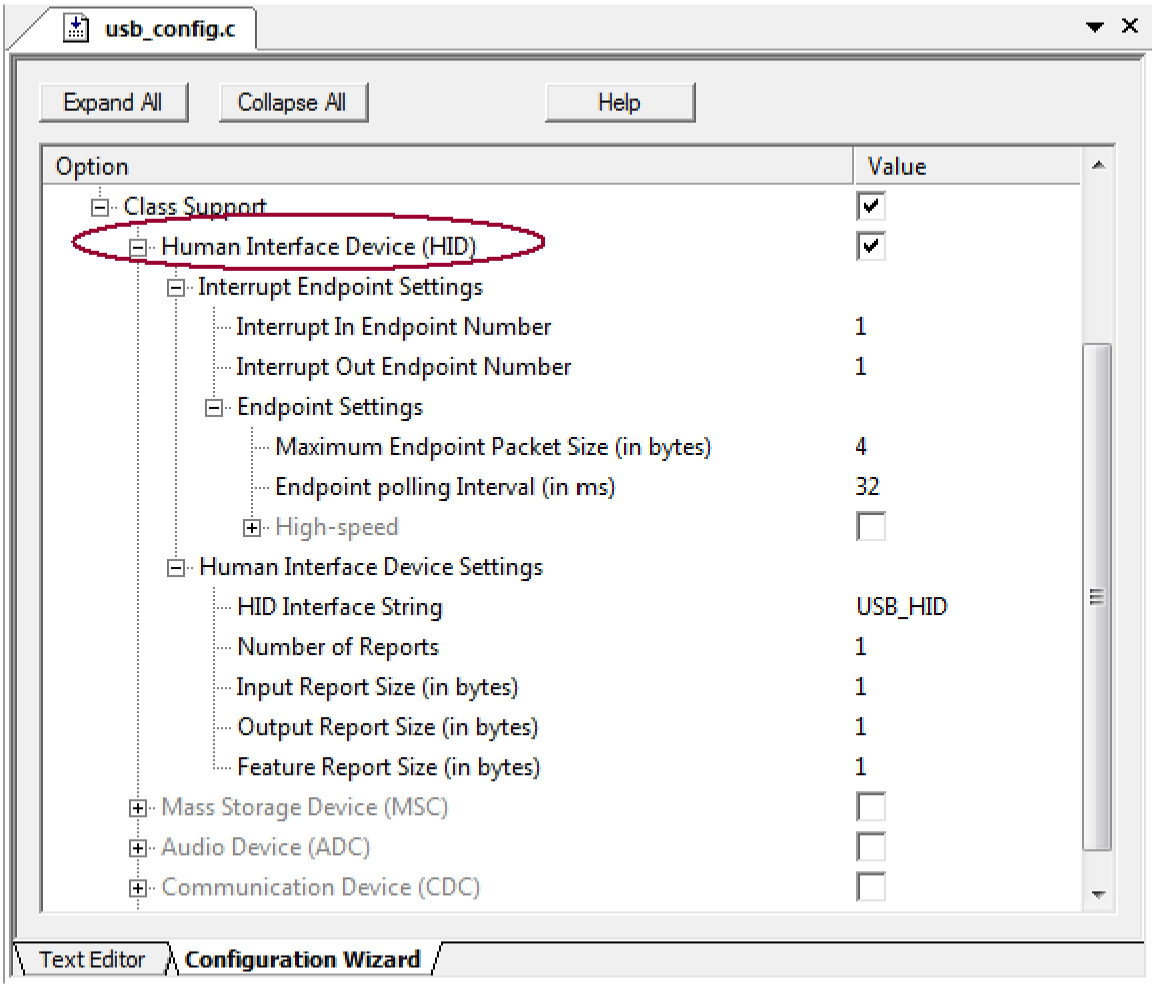
Applications can be created using existing µVision HID projects.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\BoardName\RL\USB\Device\*HID*** to a new folder and open the project ***.uvproj** with µVision. **RTX** projects are using the RTX-RTOS, whereas simple **HID** projects work without an RTOS. However, the USB configuration does not differ.
2. Open the file **usb_config.c** and configure the USB device with the Configuration Wizard.
3. Enable **USB Device** and set the device characteristics.



Option	Value
USB Host	□
USB Device	☑
High-speed	□
Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
Serial Number	☑
Serial Number String	0001A0000000
Class Support	☑

4. Enable **Class Support** and **Human Interface (HID)** and set the characteristics.



Option	Value
Class Support	☑
Human Interface Device (HID)	☑
Interrupt Endpoint Settings	
Interrupt In Endpoint Number	1
Interrupt Out Endpoint Number	1
Endpoint Settings	
Maximum Endpoint Packet Size (in bytes)	4
Endpoint polling Interval (in ms)	32
High-speed	□
Human Interface Device Settings	
HID Interface String	USB_HID
Number of Reports	1
Input Report Size (in bytes)	1
Output Report Size (in bytes)	1
Feature Report Size (in bytes)	1
Mass Storage Device (MSC)	□
Audio Device (ADC)	□
Communication Device (CDC)	□

5. Optionally, adapt the source file **usbd_*device family*.c** that contains hardware dependent code.
6. Modify the file **usbd_user_hid.c** to adapt the code to the application needs.

Note

- The options are explained in Human Interface Device (HID) Options.
- Use the Test HID Client application for testing the HID applications that interact with the LEDs and push buttons of the board.

Create MSC Applications

Create MSC Applications explains how to program a USB Mass Storage Device using the RL-USB Library. Mass Storage applications control USB sticks, cameras, or other external storage devices.

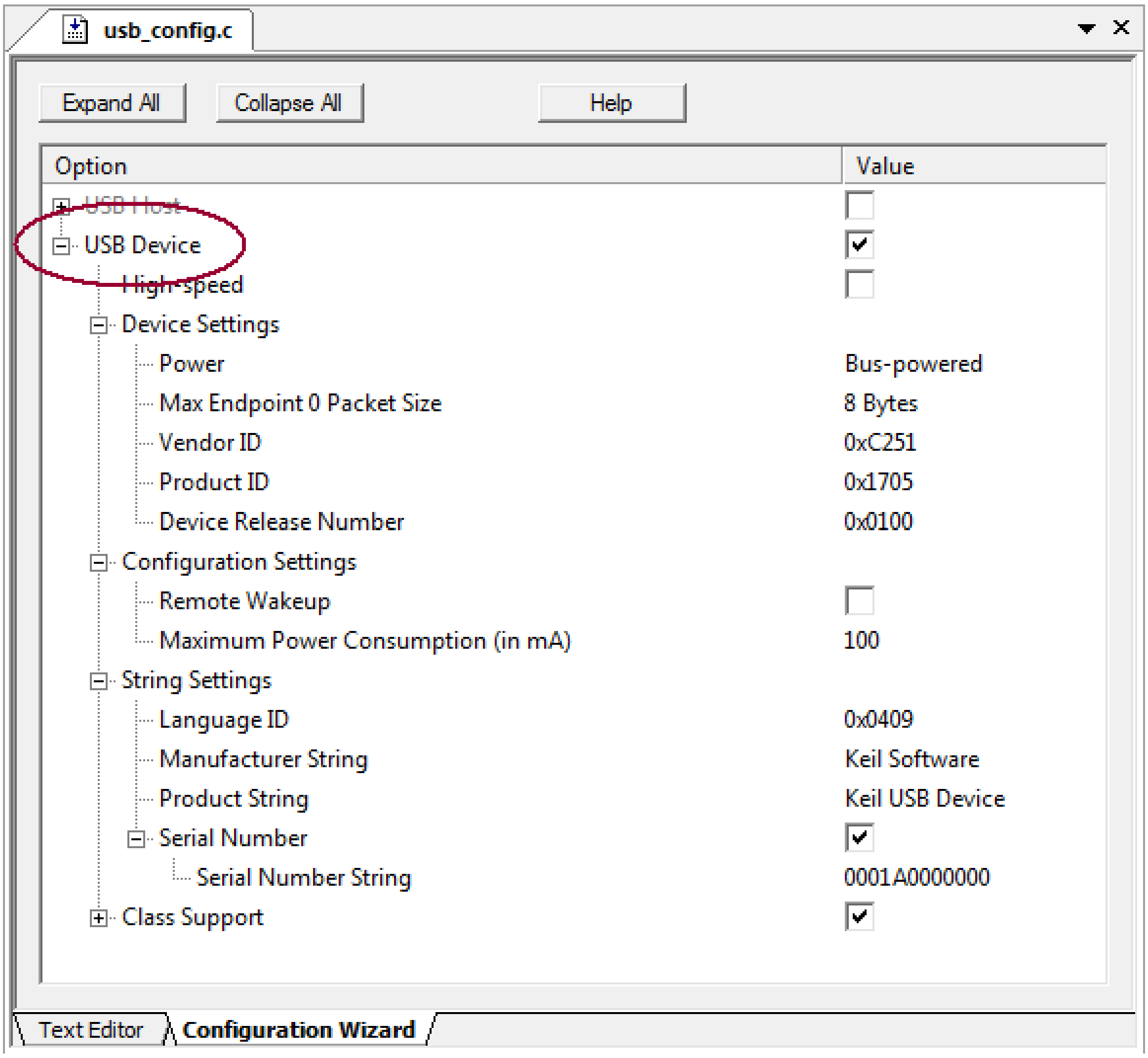
Include into the project and configure the following RL-USB Device Source Files:

1. The library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usb_config.c** - to configure the USB system.
3. **usbd_*device family*.c** - to configure the device hardware layer.
4. **DiskImg.c** - to provide the disk image.
5. **usbd_user_msc.c** - to adapt the code to the application needs.
6. **USBD_Demo.c** - to initialize and connect the USB Device from *main()*.

```

..
usbd_init();                         // USB Initialization
usbd_connect(__TRUE);                // USB Connect
..
```

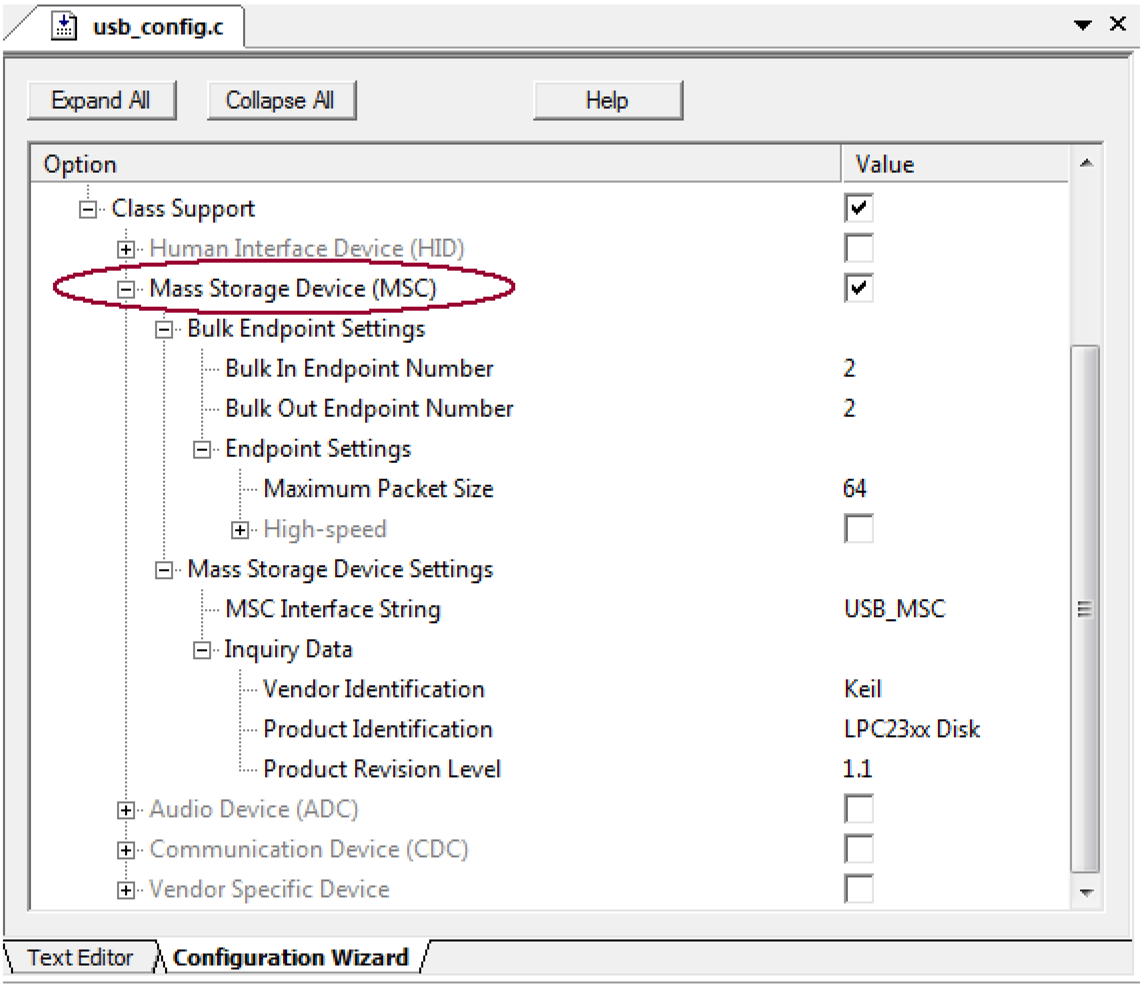
Applications can be created using existing µVision MSC projects.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\BoardName\RL\USB\Device\*Memory*** to a new folder and open the project ***.uvproj** with µVision. **RTX** projects are using the RTX-RTOS, whereas simple **Memory** projects work without an RTOS. However, the USB configuration does not differ.
2. Open the file **usb_config.c** and configure the USB device with the Configuration Wizard.
3. Enable **USB Device** and set the device characteristics.



Option	Value
USB Host	□
USB Device	☑
High-speed	□
Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
Serial Number	☑
Serial Number String	0001A0000000
Class Support	☑

4. Enable **Class Support** and **Mass Storage Device (MSC)** and set the characteristics.



```
Option                                      Value
Class Support                               ☑
  Human Interface Device (HID)              ☐
  Mass Storage Device (MSC)                 ☑
    Bulk Endpoint Settings
      Bulk In Endpoint Number               2
      Bulk Out Endpoint Number              2
      Endpoint Settings
        Maximum Packet Size                 64
        High-speed                          ☐
    Mass Storage Device Settings
      MSC Interface String                  USB_MSC
      Inquiry Data
        Vendor Identification               Keil
        Product Identification              LPC23xx Disk
        Product Revision Level              1.1
  Audio Device (ADC)                        ☐
  Communication Device (CDC)                ☐
  Vendor Specific Device                    ☐
```

5. Optionally, adapt the source file **usbd_*device family*.c** that contains hardware dependent code.
6. Optionally, modify the file **DiskImg.c**.
7. Modify the file **usbd_user_msc.c** to adapt the code to the application needs.

Note

- The options are explained in Mass Storage Device (MSC) Options.

Create Composite Applications

Create Composite Applications explains how to program a USB Device that supports more class functions. Composite applications combine various USB functionalities into one device.

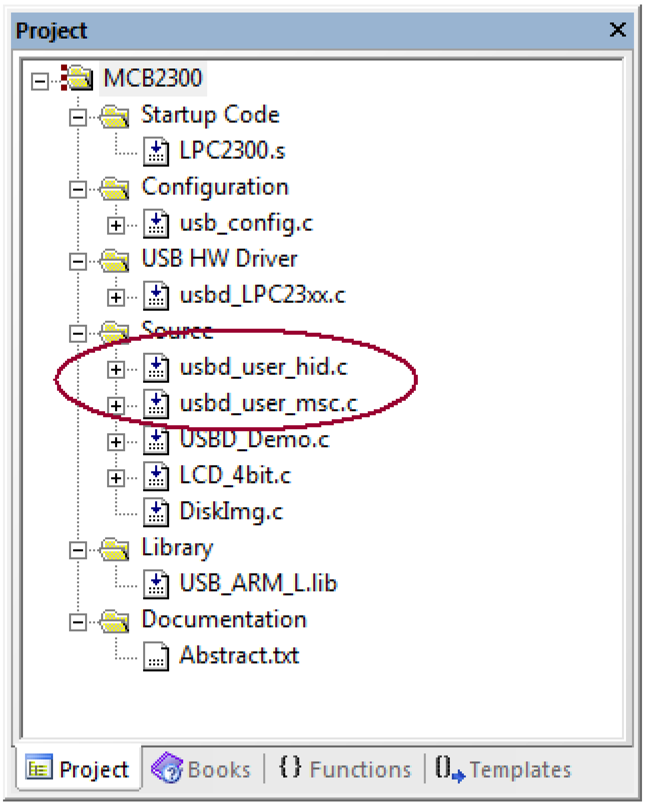
Include into the project and configure the following RL-USB Device Source Files:

1. The library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usb_config.c** - to configure the USB system.
3. ***special files*** - provide special files. For example, disk image file - for MSC support, driver information file - for CDC support.
4. **usbd_*device family*.c** - to configure the device hardware layer.
5. **usbd_user_*xxx*.c** - to adapt the code to the application needs.
6. **USBD_Demo.c** - to initialize and connect the USB Client device from *main()*.

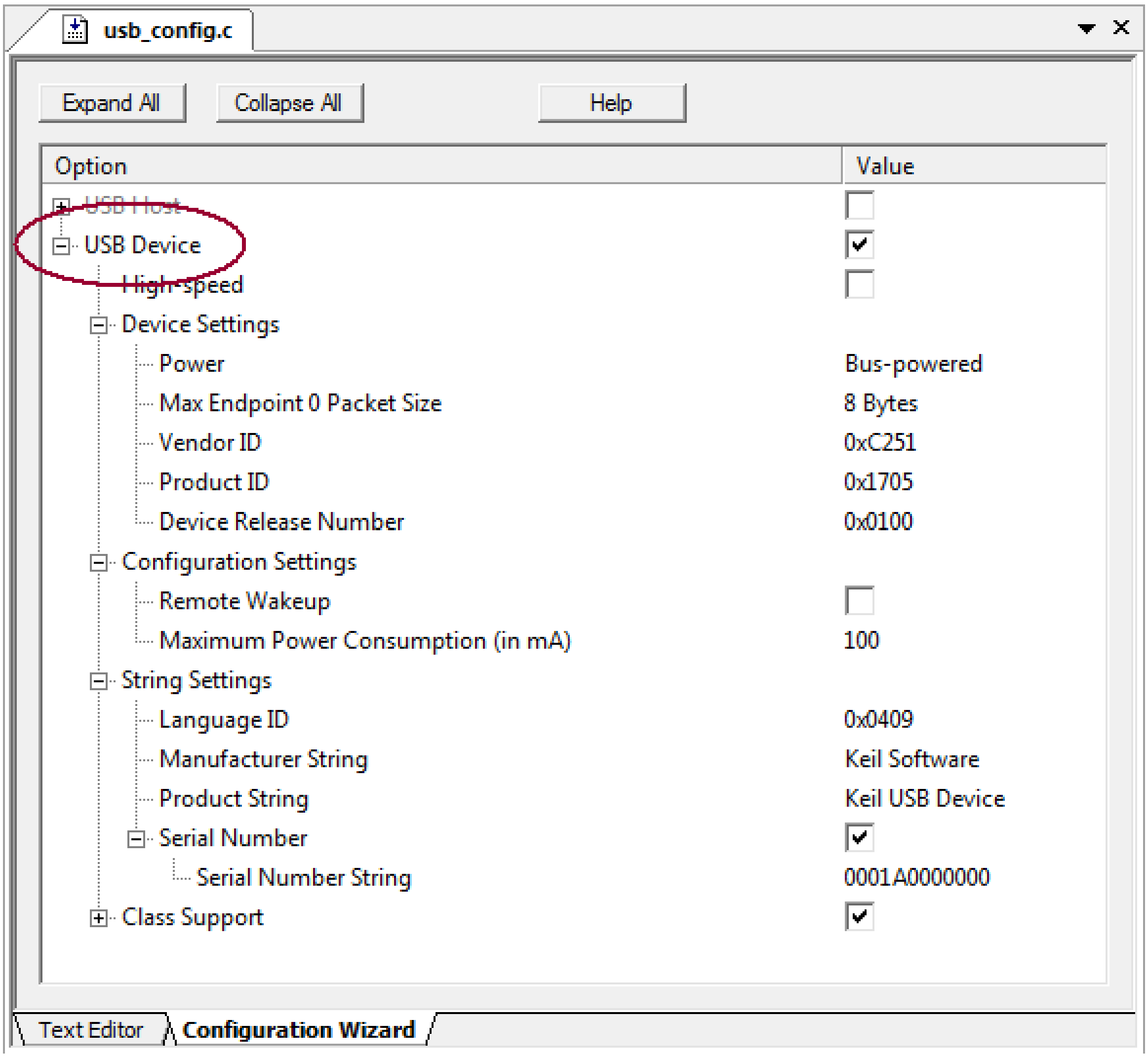
```

..
usbd_init();                         // USB Initialization
usbd_connect(__TRUE);                // USB Connect
..
```

Applications can be created using existing µVision projects. For example, to create a HID and MSC device:

1. Copy all files from any folder **\ARM\Boards\*Vendor*\BoardName\RL\USB\Device\*Memory*** to a new folder and open the project ***.uvproj** with µVision.
2. Copy the file **usbd_user_hid.c** from an existing HID project and include the file into this new project. The project should now contain the files **usbd_user_hid.c** and **usbd_user_msc.c**.

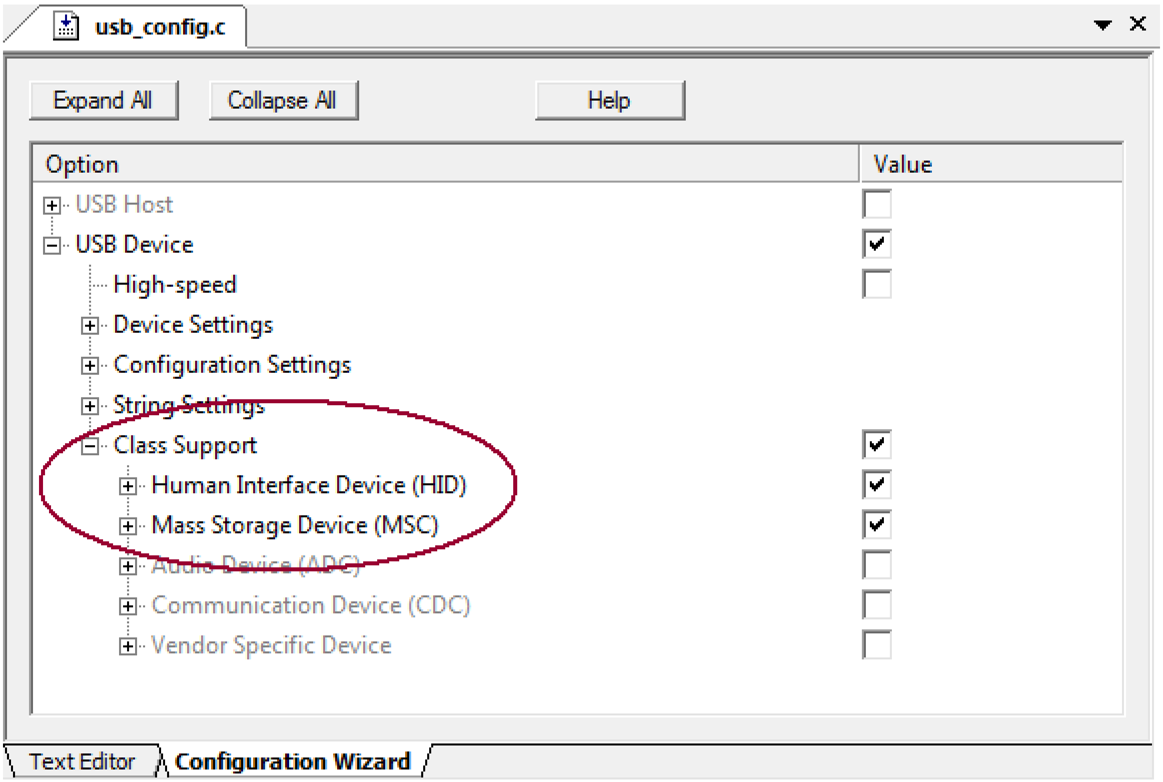


3. Open the file **usb_config.c** and configure the USB device with the Configuration Wizard.
4. Enable **USB Device** and set the device characteristics.



Option	Value
⊞ USB Host	□
⊟ USB Device	☑
High-speed	□
⊟ Device Settings	
Power	Bus-powered
Max Endpoint 0 Packet Size	8 Bytes
Vendor ID	0xC251
Product ID	0x1705
Device Release Number	0x0100
⊟ Configuration Settings	
Remote Wakeup	□
Maximum Power Consumption (in mA)	100
⊟ String Settings	
Language ID	0x0409
Manufacturer String	Keil Software
Product String	Keil USB Device
⊟ Serial Number	☑
Serial Number String	0001A0000000
⊞ Class Support	☑

5. Enable **Class Support**, **Human Interface Device (HID)**, **Mass Storage Device (MSC)**, and set the characteristics.



Option	Value
USB Host	□
USB Device	☑
High-speed	□
Device Settings	
Configuration Settings	
String Settings	
Class Support	☑
Human Interface Device (HID)	☑
Mass Storage Device (MSC)	☑
Audio Device (ADC)	□
Communication Device (CDC)	□
Vendor Specific Device	□

6. Optionally, adapt the source file **usbd_*device family*.c** that contains hardware dependent code.
7. Adapt the code in **usbd_user_hid.c** and **usbd_user_msc.c** to the application needs.

Note

- The configuration options are explained in RL-USB Device Configuration.

Test USB Device Applications

USB peripherals have to pass certain tests in order to gain certification. The links below offer the necessary test utilities:

Compliance Tests

Test the application with a **USB Command Verifier** to ensure compliance with the USB Standard.

Test HID Device Applications

Test a USB HID Device application with the **HID Client** utility.

Compliance Tests

Before releasing a USB device, ensure that the device fully meets the USB specifications. A suite of USB compliance test tools can be downloaded from the USB Implementers Forum. The type of software needed is called **USB Command Verifier**.

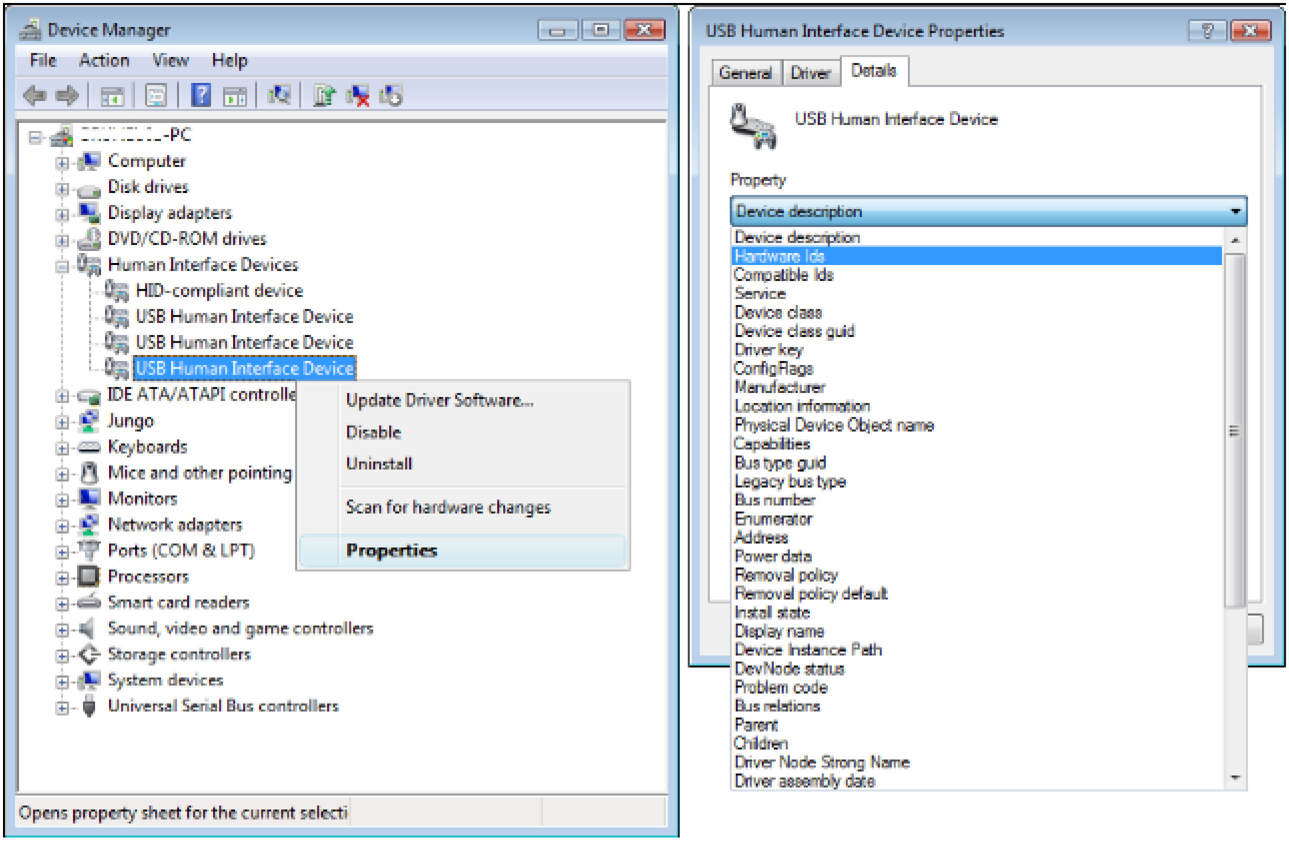
It is recommended to perform as much plug and play testing with different hosts, hubs, and operating systems as you would expect to find in the field.

Test HID Client Application

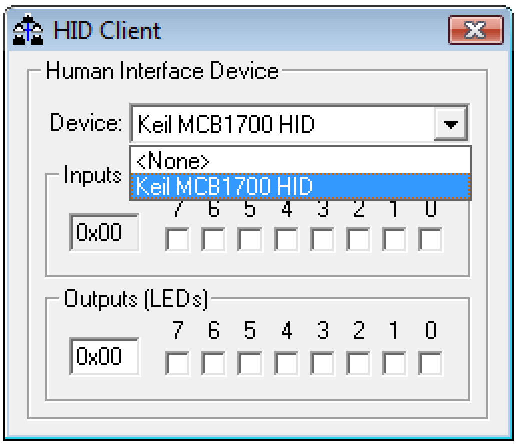
After the USB HID application has been downloaded to the device, test the application with the **HIDClient.exe** utility located in **C:\Keil\ARM\Utilities\HID_Client\Release**.

1. Verify the jumper settings of the board.
2. Connect the board to the PC using a USB cable.

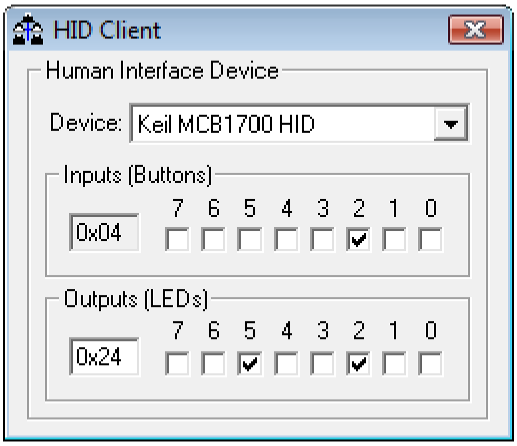
   (The Windows operating system installs the Human Interface driver automatically.)
3. Optionally, ensure that the device has been recognized. Launch the **Device Manager** from the command line with **devmgmt.msc**.
   - Scan for new hardware if necessary.
   - Use the **Property - Details** page to find the device identifiers.



4. Launch **C:\Keil\ARM\Utilities\HID_Client\Release\HIDClient.exe**.
5. Select the **Device** to establish the communication channel.



6. Test the application.
   - Move the joystick (on some boards press buttons) to enable the check-boxes **Inputs (Buttons)**.
   - Check the check-boxes **Outputs (LEDs)** to turn on the LEDs on the board.



Human Interface Device

Device: Keil MCB1700 HID

Inputs (Buttons)

	7	6	5	4	3	2	1	0
0x04	□	□	□	□	□	☑	□	□

Outputs (LEDs)

	7	6	5	4	3	2	1	0
0x24	□	□	☑	□	□	☑	□	□

RL-USB for USB Host Applications

This chapter contains the following sections:

RL-USB Host Library

Describes the RL-USB Host Library features, software stack, source files, supported class functions when designing a USB Host application.

Create USB Host Applications

Describes the steps to create new USB Host applications using the RL-USB Library.

RL-USB Host Library

The **RL-USB Host Library** is an easy-to-use collection of functions providing a common API for designing USB Host applications. The RL-USB Host Library can be used standalone or with the RTX-RTOS. The RL-USB Host Library supports the human interface device class (HID) and mass storage device class (MSC).

This chapter contains the sections:

RL-USB Host Features

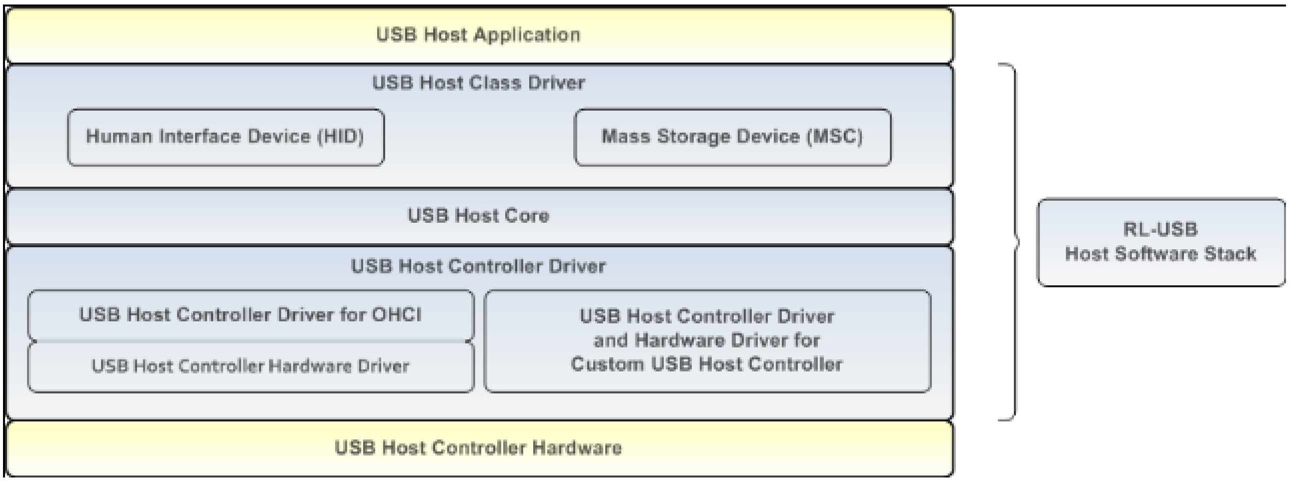
The RL-USB Host Software library can be used standalone or with the RTX-RTOS and enables the developer to create applications that:

- Comply with the USB 1.1 (Low-Speed and Full-Speed USB) specification.
- Support control, interrupt, and bulk data transfer types.
- Support the following device classes:
  - Human Interface Device (HID)
  - Mass Storage Device (MSC)

RL-USB Host Software Stack

The **RL-USB Host Software Stack** consists of the following layers:

- **USB Host Class Driver**
- **USB Host Core**
- **USB Host Controller Driver**


USB Host Application
USB Host Class Driver
Human Interface Device (HID)
Mass Storage Device (MSC)
USB Host Core
USB Host Controller Driver
USB Host Controller Driver for OHCI
USB Host Controller Hardware Driver
USB Host Controller Driver and Hardware Driver for Custom USB Host Controller
USB Host Controller Hardware
RL-USB Host Software Stack

The **USB Host Class Driver** is a hardware independent layer containing functions that are class specific and application dependent. The following USB classes are supported:

- Human Interface Device (HID)
- Mass Storage Device (MSC)

The **USB Host Core** is a hardware independent layer and contains the functions to manage USB peripherals and the USB bus. This layer is an interface between the USB Host Controller Driver and the USB Host Class Driver. The most important functions in this layer are:

- *usbh_init()* which initializes the USB Host.
- *usbh_engine()* which enables the USB Host Core to react to USB bus events. This function must be called periodically from the main application.

The **USB Host Controller Driver** is the layer between the USB Host Controller Hardware and the USB Host Core. This layer handles hardware specific interfaces. Two types of USB Host Controller Drivers are available:

1. The **Host Controller Driver for OHCI** is packed in the library, whereas the **USB Host Controller Hardware Driver** is delivered as a source module. However, the two parts are used as an entity by devices having OHCI USB Host Controllers.
2. The **USB Host Controller Driver and Hardware Driver for Custom USB Host Controller** is packed in the library and is included for devices that do not support the USB OHCI standard (this is not refering to EHCI).

RL-USB Host Functions

The following **RL-USB Host Functions** are important to understand the RL-USB Host Software Stack. RL-USB Host functions are categorized into:

- **Core functions**

  initialize and run the USB Host Core. These functions are part of the RL-USB Library.

Function Name	Description
usbh_connected	Checks whether a device is present on the USB bus.
usbh_engine	Handles USB Host events and executes enumeration.
usbh_init	Initializes the USB Host Stack and USB Host Controller Hardware.
usbh_uninit	Uninitializes the USB Host Stack and USB Host Controller Hardware.

- **Hardware layer functions**

  interact with the USB Host Controller Hardware. In most cases, these functions are provided in an external module for devices that have an OHCI compliant USB Host Controller. For devices without this module, adapt the functions manually. Provided modules are located in the folder **\ARM\RL\USB\Drivers** and have the name **usbh_ohci_*device family name*.c**.

Function Name	Description
usbh_ohci_hw_delay	Delays execution for a specified time.
usbh_ohci_hw_init	Initializes the hardware. For example, clocks or pins.
usbh_ohci_hw_irq_dis	Disables the USB Host OHCI Controller interrupt.
usbh_ohci_hw_irq_en	Enables the USB Host OHCI Controller interrupt.
usbh_ohci_hw_power	Supplies or removes power on the USB Host OHCI Controller port.
usbh_ohci_hw_reg_rd	Reads data from the OHCI register.
usbh_ohci_hw_reg_wr	Writes data to the OHCI register.
usbh_ohci_hw_uninit	Uninitializes the hardware. For example, clocks or pins.

- **HID Class specific functions**

  enable the USB Host to support USB Human Interface Devices.

Function Name	Description
usbh_hid_kbd_getkey	Retrieves the signals sent by a USB keyboard.
usbh_hid_mouse_getdata	Reads the signals sent by a USB mouse.

- **MSC Class specific functions**

  enable the USB Host to support USB Mass Storage Devices.

Function Name	Description
usbh_msc_read_config	Reads the USB MSC device configuration and checks whether the device is compatible with the Flash File System.
usbh_msc_read	Reads USB MSC device sector data into a buffer.
usbh_msc_status	Checks whether the USB MSC device is connected and ready.
usbh_msc_write	Writes data from a buffer to USB MSC device sectors.

RL-USB Host Source Files

RL-USB Host Source Files for creating USB Host applications with the RL-USB Library can be found in the folders:

Folder Name	Description
\ARM\RV31\INC	Contains include files, header files, and configuration files.
\ARM\RV31\LIB	Contains the library files **USB_ARM_L.LIB** and **USB_CM3.LIB**.
\ARM\RL\USB\Drivers	Contains USB Host Controller Driver modules.
\ARM\Boards\*Vendor*\Board\RL\USB\Host	Contains example applications built with the RL-USB Library. Use the projects as templates to create new USB Host applications.

RL-USB Host files in **\ARM\RV31\INC**:

File Name	File Type	Layer	Description
usb_lib.c	Module	All layers	Packs configuration settings and provides them to the library. The file is included from the module file USB_CONFIG.C. No code changes are required.
rl_usb.h	Header	All layers	Contains prototypes of functions exported from the library or imported to the library. No code changes are required.
usb.h	Header	Core Driver	Main RL-USB header file used to include other USB header files.
usb_adc.h	Header	Core Driver	Contains definitions and structures used by the audio device class such as endpoint, subclasses, audio data formats, audio processing unit, request codes, ...
usb_cdc.h	Header	Core Driver	Contains definitions and structures used by the communication device class such as endpoint, subclasses, sub-types, request codes, ...
usb_hid.h	Header	Core Driver	Contains definitions and structures used by the human interface device class such as subclasses, protocol codes, descriptor types, reports, ...
usb_msc.h	Header	Core Driver	Contains definitions and structures used by the mass storage device class such as subclasses, protocol codes, request codes, SCSCI commands, ...
usb_def.h	Header	Core Driver	Contains definitions for device classes, descriptors, pipes.

RL-USB Host Library files in **\ARM\RV31\LIB**:

File Name	File Type	Layer	Description
USB_ARM_L.lib	Library	All layers	RL-USB library for ARM7 and ARM9 devices - Little Endian.
USB_CM3.lib	Library	All layers	RL-USB library for Cortex-M3 devices - Little Endian.

RL-USB Host Application files in **\ARM\Boards\*Vendor*\Board\RL\USB\Host**:

File Name	File Type	Layer	Description
usb_config.c	Module	All layers	Configures the USB Host Controller and USB Host Classes at compile time. Settings for the USB Host Controller Driver and USB Host Class Drivers are available. Modify this file to suit the application requirements.
usbh_ohci_*device family*.c	Module	Host Controller Driver	Provides the OHCI hardware specific driver functions. Standardized modules are available in the folder **\ARM\RL\USB\Drivers**. Adapt this file, if no module is provided for the used device.

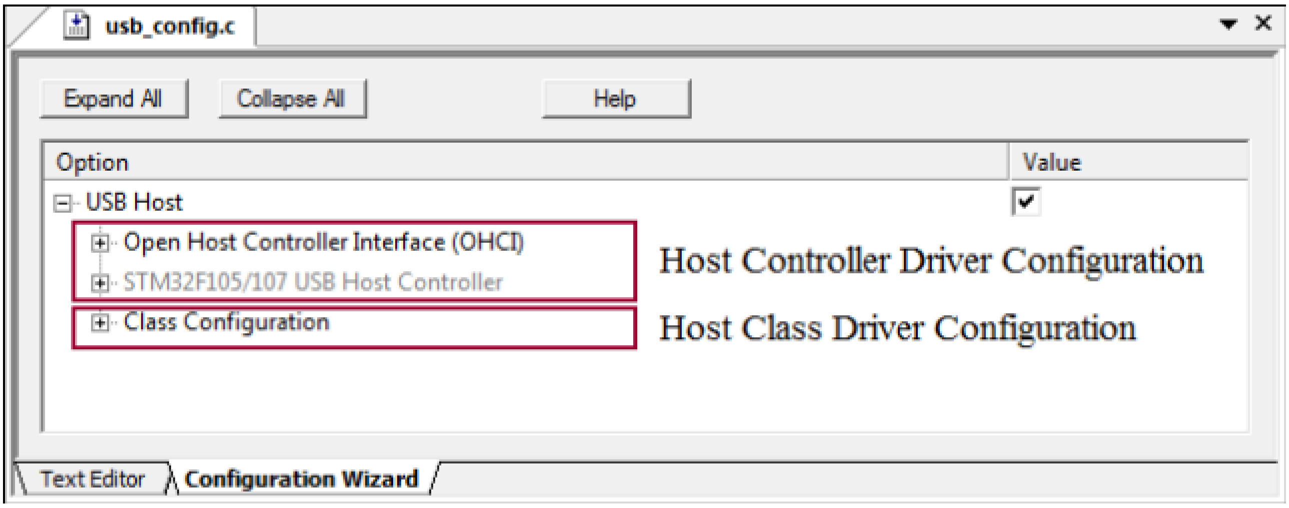
Source files required to support **MSC devices**. Use these files in addition to the RL-USB Host source files outlined in the tables above.

	File Type	Layer	Description
File_Config.c	Module	Application	Provides configuration options for FlashFS. Options to define the USB drive letter and cache size are

	File Type	Layer	Description
			available. Modify the options to suit the application needs.
fs_usbh_msc.c	Module	Application	Provides the USB Host functions to support MSC devices. Modify the code to suit the application needs.
FS_ARM_L.lib	Library	Application	Flash File System library for ARM7 and ARM9 devices - Little Endian.
FS_CM3.lib	Library	Application	Flash File System library for Cortex-M3 devices - Little Endiand.

RL-USB Host Configuration

RL-USB Host Configuration explains the options offered by the RL-USB Library for configuring USB Host devices. The options can be configured in the file **usb_config.c** directly or using the µVision Configuration Wizard.


usb_config.c
Expand All
Collapse All
Help
Option
Value
USB Host
☑
Open Host Controller Interface (OHCI)
STM32F105/107 USB Host Controller
Class Configuration
Host Controller Driver Configuration
Host Class Driver Configuration
Text Editor
Configuration Wizard

USB Host enables the USB Host functionlity. This options corresponds to *#define USBH_ENABLE*.

```
#define USBH_ENABLE                    1                              // enable
HC; (0=disabled)
```

The USB Host Controller configuration can be split into two main groups:

1. Host Controller Driver Configuration, which offers:
   - Options to configure a OHCI compliant Host Controller.
   - Options to configure a custom USB Host Controller.
2. Host Class Driver Configuration, which defines the supported class functions.

Host Controller Driver Configuration

The section explains the configuration options for the USB Host Controller Driver layer, which supports the OHCI standard and the custom USB Host Controller Driver for the ST STM32F105/107 device series. The configuration options are set in the file **usb_config.c**. The following option blocks are available:

- Open Host Controller Interface (OHCI) for NXP devices.
- STM32F105/107 USB Host Controller (Custom driver) for ST STM32 devices.

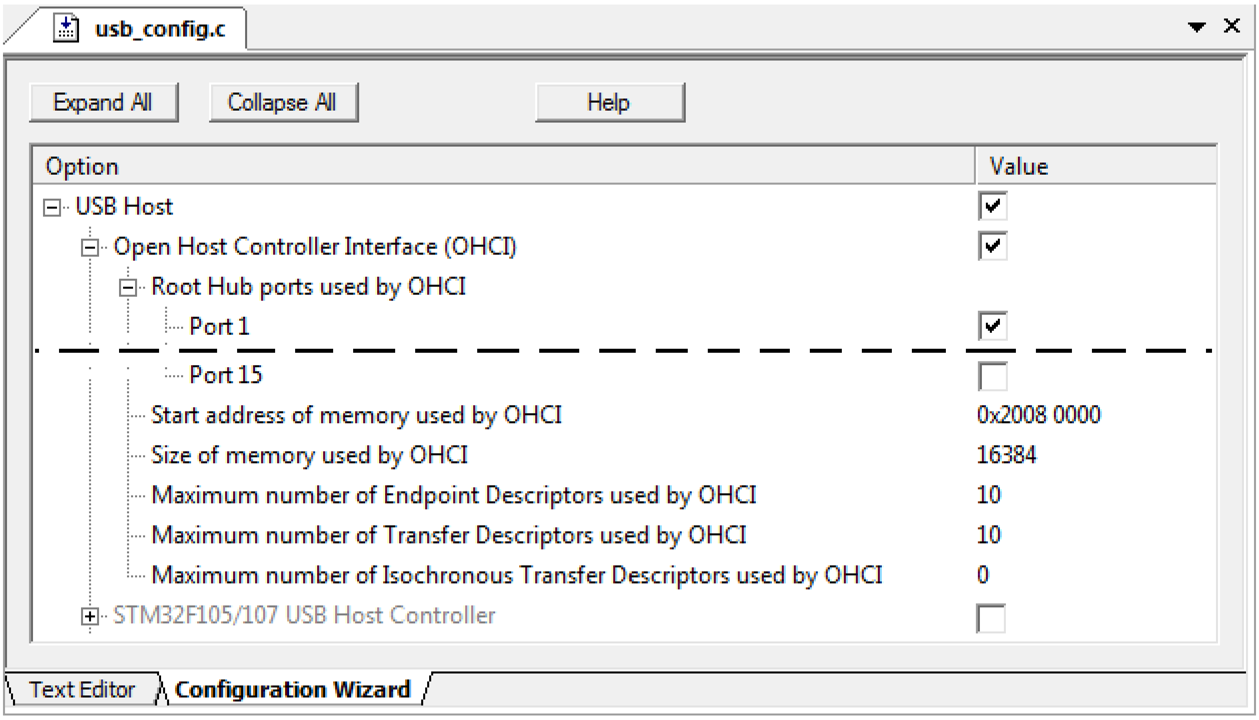
Note

- The option **Open Host Controller Interface (OHCI)** cannot be used in combination with **STM32F105/107 USB Host Controller**. Both options configure the USB Host Controller Driver, but address different device types.

Open Host Controller Interface (OHCI)

Open Host Controller Interface (OHCI) activates the OHCI. This option configures the **USB Host Controller Driver for OHCI** layer of the RL-USB Host Controller Software Stack. Enable this option when using any device that supports OHCI. This option corresponds to *#define USBH_OHCI_ENABLE* .

```
#define USBH_OHCI_ENABLE                   1                            //
OHCI enabled (0=disabled)
```



Option	Value
USB Host	☑
Open Host Controller Interface (OHCI)	☑
Root Hub ports used by OHCI	
Port 1	☑
Port 15	□
Start address of memory used by OHCI	0x2008 0000
Size of memory used by OHCI	16384
Maximum number of Endpoint Descriptors used by OHCI	10
Maximum number of Transfer Descriptors used by OHCI	10
Maximum number of Isochronous Transfer Descriptors used by OHCI	0
STM32F105/107 USB Host Controller	□

The following configuration settings are available:

- **Root Hub ports used by OHCI** sets the number of ports used by the OHCI. 15 Ports can be configured. This option corresponds to *#define USBH_OHCI_PORTS*.

  ```
  #define USBH_OHCI_PORTS               0x00000001                        //
  activate the first port
  ```
- **Start address of memory used by OHCI** sets the memory start address for descriptors and communication data. This option corresponds to *#define USBH_OHCI_MEM_ADDR*.

  ```
  #define USBH_OHCI_MEM_ADDR            0x20080000                        //
  memory address start
  ```
- **Size of memory used by OHCI** sets the memory size used by the descriptors and communication data. The size of maximum 1048576 bytes can be set. This option

corresponds to *#define USBH_OHCI_MEM_SIZE*.

-

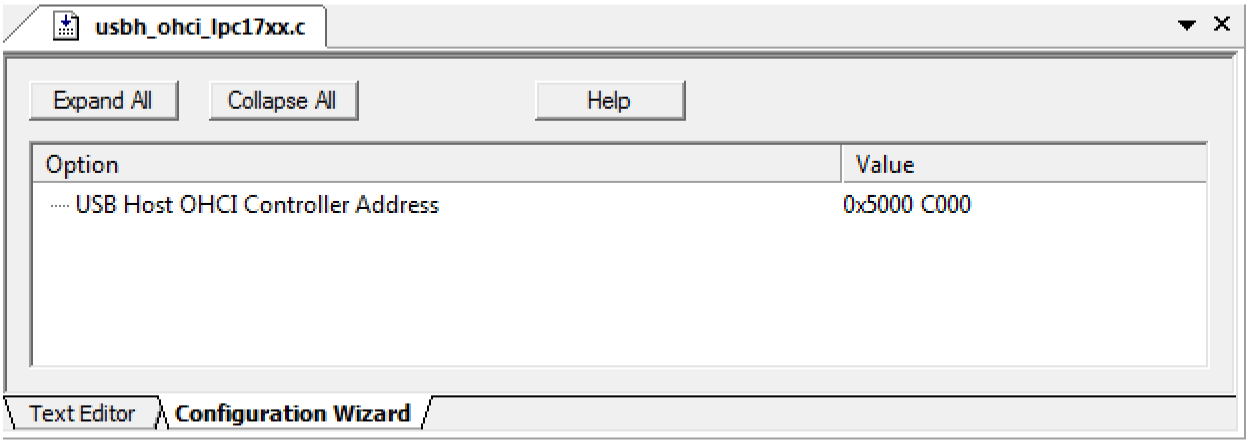
-
  ```
  #define USBH_OHCI_MEM_SIZE            16384                  //
  memory size
  ```
- **Maximum number of Endpoint Descriptors used by OHCI** sets the maximum number of serviced endpoints. At least one endpoint has to be set. Maximum 64 endpoints can be serviced. This option corresponds to *#define USBH_OHCI_ED_MAX_NUM*.
-

-
  ```
  #define USBH_OHCI_ED_MAX_NUM          10                     //
  10 Endpoints
  ```
- **Maximum number of Transfer Descriptors used by OHCI** sets the maximum number of transfer slots. At least one transfer slot has to exist. A maximum of 64 transfer slots are possible. This option corresponds to *#define USBH_OHCI_TD_MAX_NUM*.
-

-
  ```
  #define USBH_OHCI_TD_MAX_NUM          10                     //
  10 transfers per frame
  ```
- **Maximum number of Isochronous Transfer Descriptors used by OHCI** sets the maximum number of isochronous transfer slots. A maximum of 64 isochronous transfers slots are possible. This option can be set to zero. This option corresponds to *#define USBH_OHCI_ITD_MAX_NUM*.
-

-
  ```
  #define USBH_OHCI_ITD_MAX_NUM         0                      //
  no Isochronous transfers
  ```

The USB Host OHCI Controller Address is set in the file **usbh_ohci_*family device name*.c** and configures the **USB Host Controller Hardware Driver** layer of the RL-USB Host Controller Software Stack.



Option	Value
USB Host OHCI Controller Address	0x5000 C000

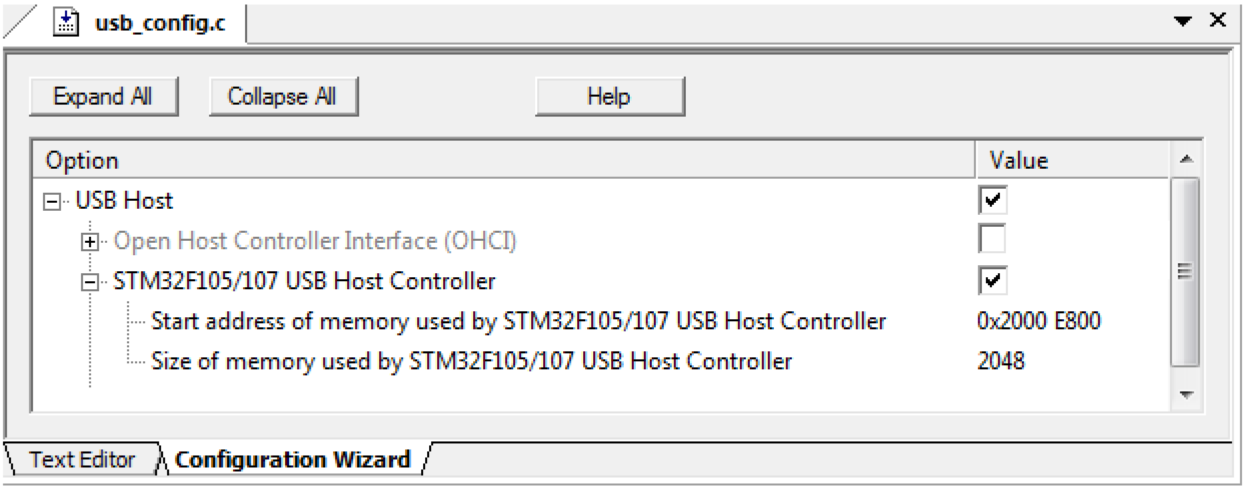
The option **USB Host OHCI Controller Address** corresponds to *#define USBH_OHCI_ADDR*.

```
define USBH_OHCI_ADDR                0x5000C000                //
Host Controller address
```

STM32F105/107 USB Host Controller

STM32F105/107 USB Host Controller activates a custom USB Host Controller driver. This option configures the **USB Host Controller Driver and Hardware Driver for Custom USB Host Controller** layer in the RL-USB Host Controller Software Stack. Enable this option when using ST STM32 devices. This option corresponds to *#define USBH_STM32_ENABLE*.

```
#define USBH_STM32_ENABLE            1                         //
custom HC enabled (0=disabled)
```



Option	Value
USB Host	☑
Open Host Controller Interface (OHCI)	□
STM32F105/107 USB Host Controller	☑
Start address of memory used by STM32F105/107 USB Host Controller	0x2000 E800
Size of memory used by STM32F105/107 USB Host Controller	2048

The following configuration settings are available:

- **Start address of memory used by STM32F105/107 USB Host Controller** sets the memory start address for communication data. This option corresponds to *#define USBH_STM32_MPOOL_MEM_POS*.

```
#define USBH_STM32_MPOOL_MEM_POS        0x2000E800                  //
memory start address
```

- **Size of memory used by STM32F105/107 USB Host Controller** sets the memory size for communication data. The size of maximum 1048576 bytes can be set. This option corresponds to *#define USBH_STM32_MPOOL_MEM_SZ*.

```
#define USBH_STM32_MPOOL_MEM_SZ         0x00000800                  //
memory size
```

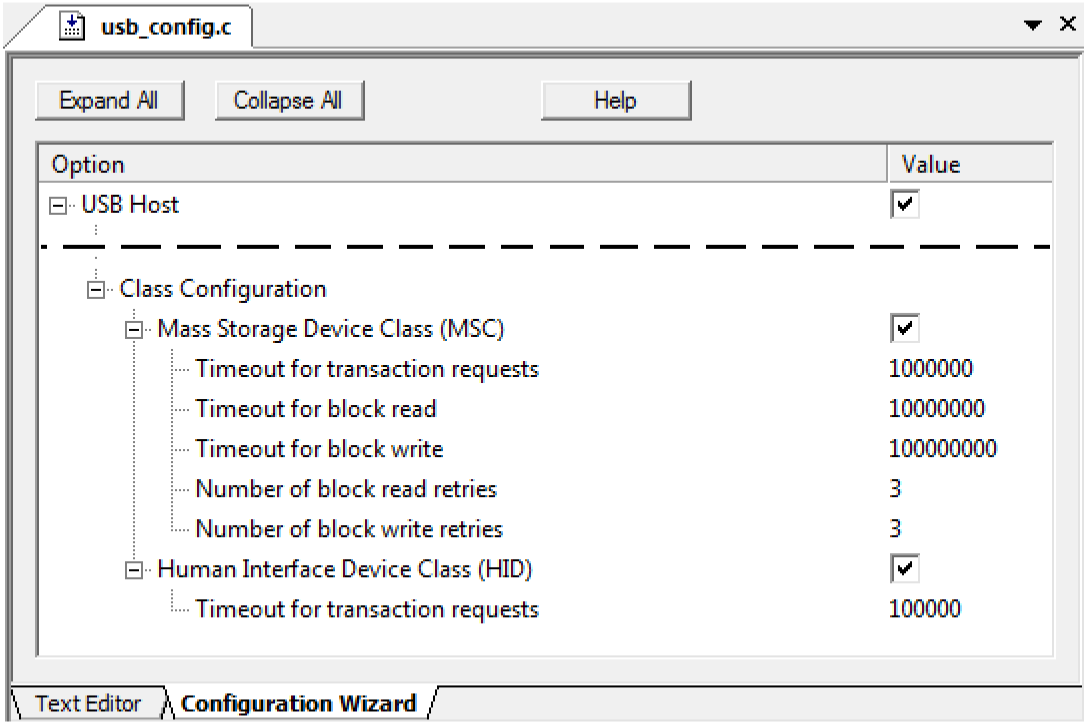
Host Class Driver Configuration

This section explains the configuration options for the USB Host Class Driver layer. The options are set in the file **usb_config.c**.

The **Class Configuration** options enable the host to interact with the following device types:

- Mass Storage Device Class (MSC)
- Human Interface Class (HID)

Mass Storage Device Class (MSC)



Option	Value
USB Host	☑
Class Configuration	
Mass Storage Device Class (MSC)	☑
Timeout for transaction requests	1000000
Timeout for block read	10000000
Timeout for block write	100000000
Number of block read retries	3
Number of block write retries	3
Human Interface Device Class (HID)	☑
Timeout for transaction requests	100000

Mass Storage Device Class (MSC) activates the host support for accessing mass storage devices. This option corresponds to *#define USBH_MSC_ENABLE*.

```
#define USBH_MSC_ENABLE                    1                            //
MSC enabled; (0=disabled)
```

- **Timeout for transaction requests** sets the maximum time allowed for transaction requests. The value has to be set at least to 1. A maximum of 2E+9 is possible. This option corresponds to *#define USBH_MSC_TR_TOUT*.
-

-
  ```
  #define USBH_MSC_TR_TOUT                1000000                         //
  Default=1000000;  ~0.08s @100MHz
  ```
- **Timeout for block read** sets the maximum time allowed for reading a block of data. The value has to be set at least to 1. A maximum of 2E+9 is possible. This option corresponds to *#define USBH_MSC_RD_BLK_TOUT*.
-

-
  ```
  #define USBH_MSC_RD_BLK_TOUT            10000000                        //
  Default=10000000;  ~0.8s @100MHz
  ```
- **Timeout for block write** sets the maximum time allowed to for writing a block of data. The value has to be set at least to 1. A maximum of 2E+9 is possible. This option corresponds to *#define USBH_MSC_WR_BLK_TOUT*.
-

-
  ```
  #define USBH_MSC_WR_BLK_TOUT            100000000                       //
  Default=100000000;  ~8s @100MHz
  ```
- **Number of block read retries** sets the maximum number of attempts allowed to read a block of data in case of failures. The value has to be set at least to 1. A maximum of 100

attempts are possible. This option corresponds to *#define USBH_MSC_RD_RETRIES*.

```
#define USBH_MSC_RD_RETRIES            3                              // try to read 3 times
```

- **Number of block write retries** sets the maximum number of attempts allowed to write a block of data in case of failures. The value has to be set at least to 1. A maximum of 100 attempts are possible. This option corresponds to *#define USBH_MSC_WR_RETRIES*.

```
#define USBH_MSC_WR_RETRIES            3                              // try to write 3 times
```

Human Interface Device Class (HID)

Human Interface Device Class (HID) activates the host support for interacting with human interface devices. This option corresponds to *#define USBH_HID_ENABLE*.

```
#define USBH_HID_ENABLE                1                              // HID enabled; (0=disabled)
```

- **Timeout for transaction requests** sets the maximum time allowed for transaction requests. The value has to be set at least to 1. A maximum of 2E+9 is possible. This option corresponds to *#define USBH_HID_TR_TOUT*.

```
#define USBH_HID_TR_TOUT               1000000                        // Default=1000000;  ~0.08s @100MHz
```

Create USB Host Applications

Create USB Host Applications shows how to program a USB Host unsing the RL-USB Library. The application source code is located in the folders **\Keil\ARM\Boards\*Vendor**Board*\RL\USB\Host** .

- **Create USB Host HID Applications**

  Configures a USB Host to interact with a USB HID Device.
- **Create USB Host MSD Applications**

  configures a USB Host to interact with a USB MSD Device.

Each example demonstrates a USB Host supporting another class function:

- **HID_Kbd Example**

  This example shows a USB Host interacting with a human interface device. It uses the Measure example to demonstrate command inputs via a USB keyboard. The application configures the evaluation board as a USB Host.
- **MSD_File Example**

  This example application shows a USB Host interacting with a mass storage device. It uses the Flash File System to store, retrieve, and manage files. The application configures the evaluation board as a USB Host.

Create USB Host HID Applications

Create USB Host HID Applications shows how to program a USB Host to support USB human interface devices using the RL-USB Library.

Include into the project and configure the following RL-USB Host Source Files:

1. The USB library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. **usbh_ohci_*device family*.c** - to configure the device hardware layer (for devices with OHCI host controllers).
3. **usb_config.c** - to configure the USB system.
4. ***main file*.c** - to initialize the USB Host from *main()*.

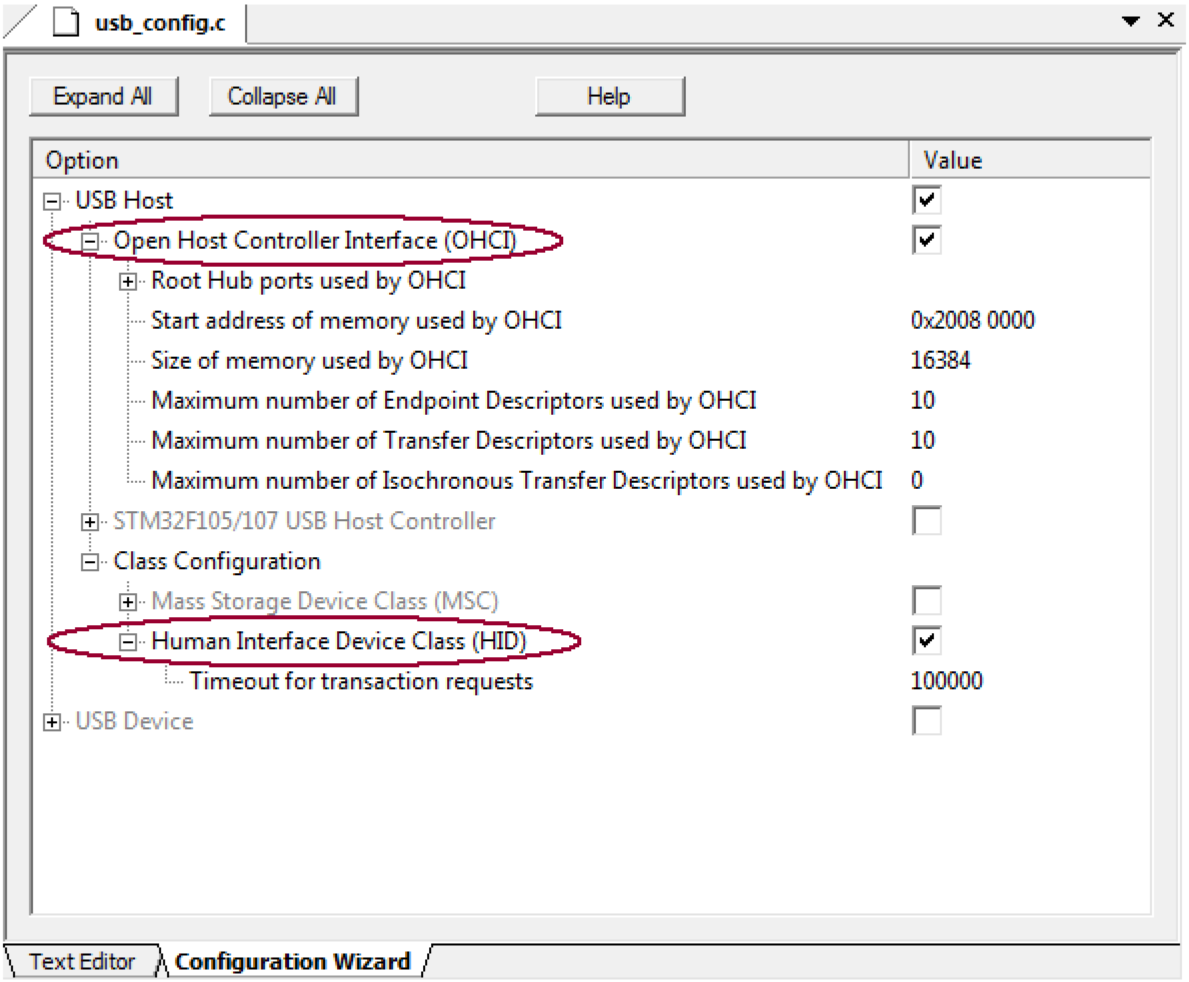
```

..
usbh_init();                                          // initialize USB Host Controller
while(1) {
  ..
  usbh_engine();                                      // check for new devices
  ..
}
```

Several µVision USB Host HID projects can be used as a template for creating new USB Host HID applications. The files can be edited directly or using the µVision Configuration Wizard.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\Board\RL\USB\Host\HID_Kbd** to a new folder and open the ***.uvproj** project with µVision.
2. Open the file **usb_config.c** and set **USB Host**.

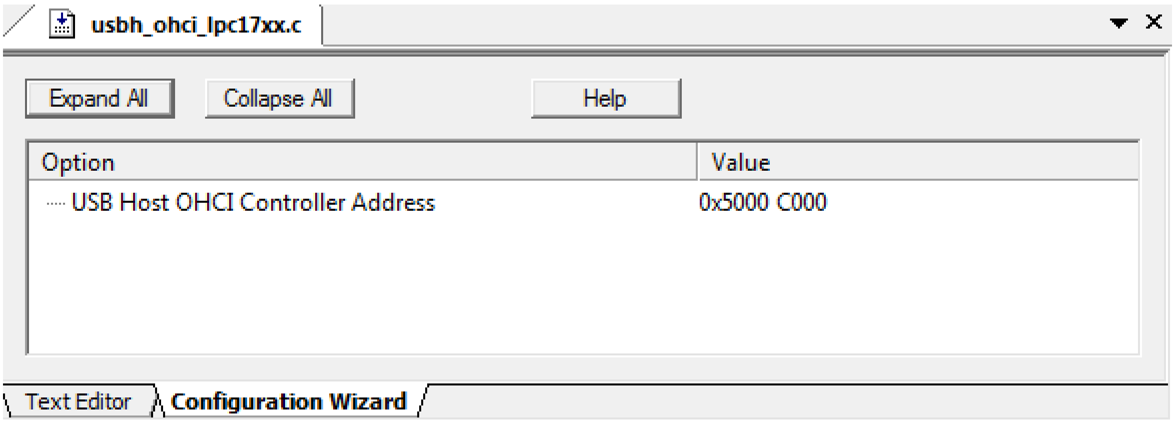
For USB Host devices that support OHCI:

3. Enable the option **Open Host Controller Interface (OHCI)**.



Option	Value
USB Host	☑
Open Host Controller Interface (OHCI)	☑
Root Hub ports used by OHCI	
Start address of memory used by OHCI	0x2008 0000
Size of memory used by OHCI	16384
Maximum number of Endpoint Descriptors used by OHCI	10
Maximum number of Transfer Descriptors used by OHCI	10
Maximum number of Isochronous Transfer Descriptors used by OHCI	0
STM32F105/107 USB Host Controller	□
Class Configuration	
Mass Storage Device Class (MSC)	□
Human Interface Device Class (HID)	☑
Timeout for transaction requests	100000
USB Device	□

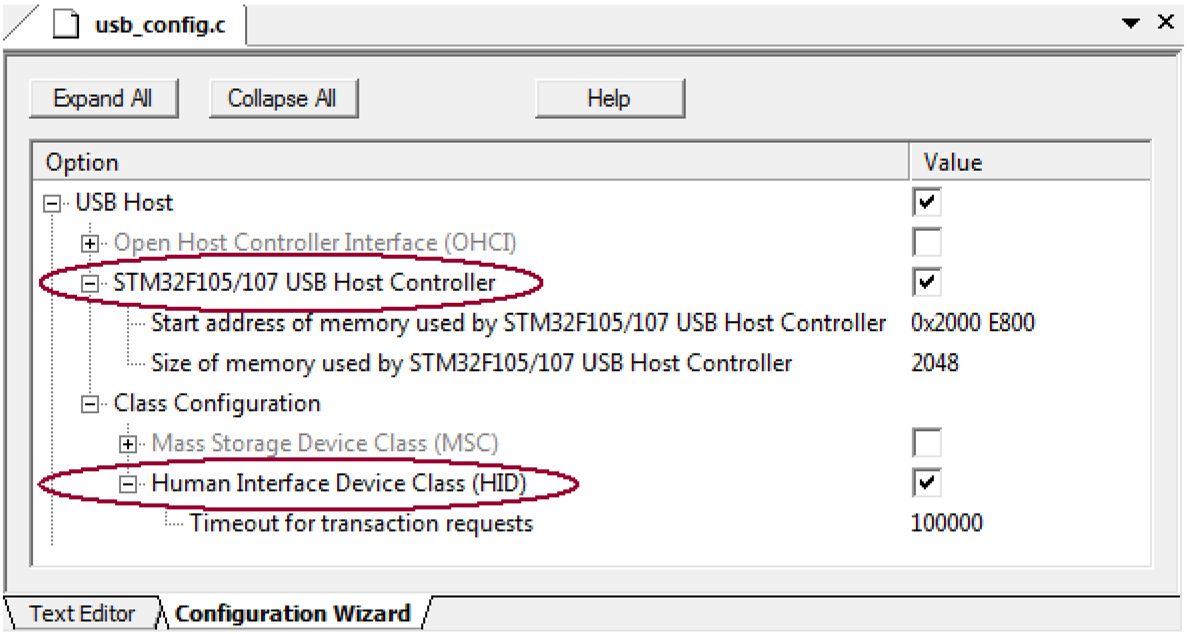
4. Set the configuration values.
5. Disable any custom USB Host Controller Driver.
6. In the **Class Configuration** section, enable **Human Interface Device Class (HID)** and set the timeout for transaction requests.
7. Open the file **usbh_ohci_*device family*.c** and set the **USB Host OHCI Controller Address**.



Option	Value
USB Host OHCI Controller Address	0x5000 C000

For USB Host devices that support a Custom USB Host Controller:

3. Disable the option **Open Host Controller Interface (OHCI)**.



Option	Value
⊟ USB Host	☑
⊞ Open Host Controller Interface (OHCI)	□
⊟ STM32F105/107 USB Host Controller	☑
Start address of memory used by STM32F105/107 USB Host Controller	0x2000 E800
Size of memory used by STM32F105/107 USB Host Controller	2048
⊟ Class Configuration	
⊞ Mass Storage Device Class (MSC)	□
⊟ Human Interface Device Class (HID)	☑
Timeout for transaction requests	100000

4. Enable the ***Custom USB Host Controller***.
5. Set the configuration options.
6. In the **Class Configuration** section, enable **Human Interface Device Class (HID)** and set the timeout for transaction requests.

Note

- The configuration options are explained in RL-USB Host Configuration.

HID_Kbd Example

The **HID_Kbd** example application shows a USB Host device interacting with a USB HID Class device. The interaction is explained on behalf of the Measure program. Commands are entered via a USB keyboard connected to an evaluation board configured as the USB Host. The Measure output and input prompt is shown on the board's LCD display.

Required hardware:

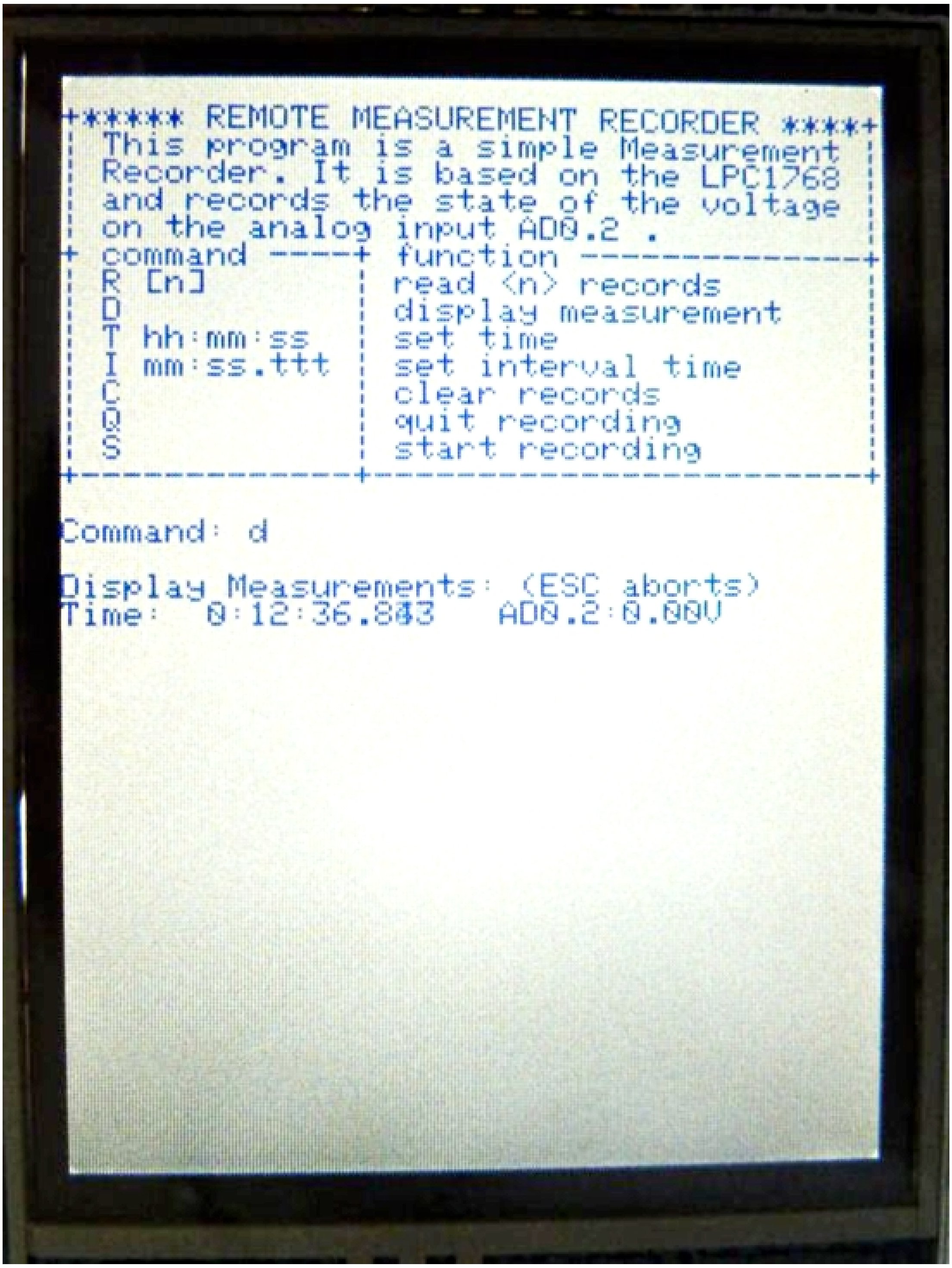
- An evaluation board that supports USB Host.
- USB keyboard.
- ULINK2 USB-JTAG debug adapter.

Hardware setup:

- Connect the board to the ULINK2 USB-JTAG adapter, and the adapter to the PC.
- Connect the USB keyboard to the board.
- Verify the jumper settings on the board.
- Power-up the board.

Open the project **Measure.uvproj** located in the folder **\ARM\Boards\*Vendor**Board*\RL\USB\Host\HID_Kbd** with µVision.

1. Build the project and download it to target.
2. Start a debugging session. Click **Run** to continue.
   The board's LCD display shows the menu of the Measure program.
3. Press "D" on the USB keyboard to display the measured values.



```
+***** REMOTE MEASUREMENT RECORDER ****+
| This program is a simple Measurement |
| Recorder. It is based on the LPC1768 |
| and records the state of the voltage |
| on the analog input AD0.2 .          |
+ command -----+ function -------------+
| R [n]        | read <n> records      |
| D            | display measurement   |
| T hh:mm:ss   | set time              |
| I mm:ss.ttt  | set interval time     |
| C            | clear records         |
| Q            | quit recording        |
| S            | start recording       |
+--------------+-----------------------+

Command: d

Display Measurements: (ESC aborts)
Time:  0:12:36.843   AD0.2:0.00V
```

Create USB Host MSC Applications

Create USB Host MSC Applications shows how to create a USB Host application that supports the mass storage class using the RL-USB Library.

Include into the project and configure the following RL-USB Host Source Files:

1. The USB library that matches the device core:
   USB_CM3.lib - for Cortex-M devices.
   USB_ARM_L.lib - for ARM7 or ARM9 devices.
2. The FlashFile library that matches the device core:
   FS_CM3.lib - for Cortex-M devices.
   FS_ARM_L.lib - for ARM7 or ARM9 devices.
3. **usbh_ohci_*device family*.c** - to configure the device hardware layer (for devices with OHCI host controllers).
4. **usb_config.c** - to configure the USB system.
5. **File_Config.c** - to configure the USB Flash Drive.
6. **fs_usbh_msc.c** - to adapt the code to the application needs.
7. ***main file*.c** - to initialize the USB Host from *main()*.

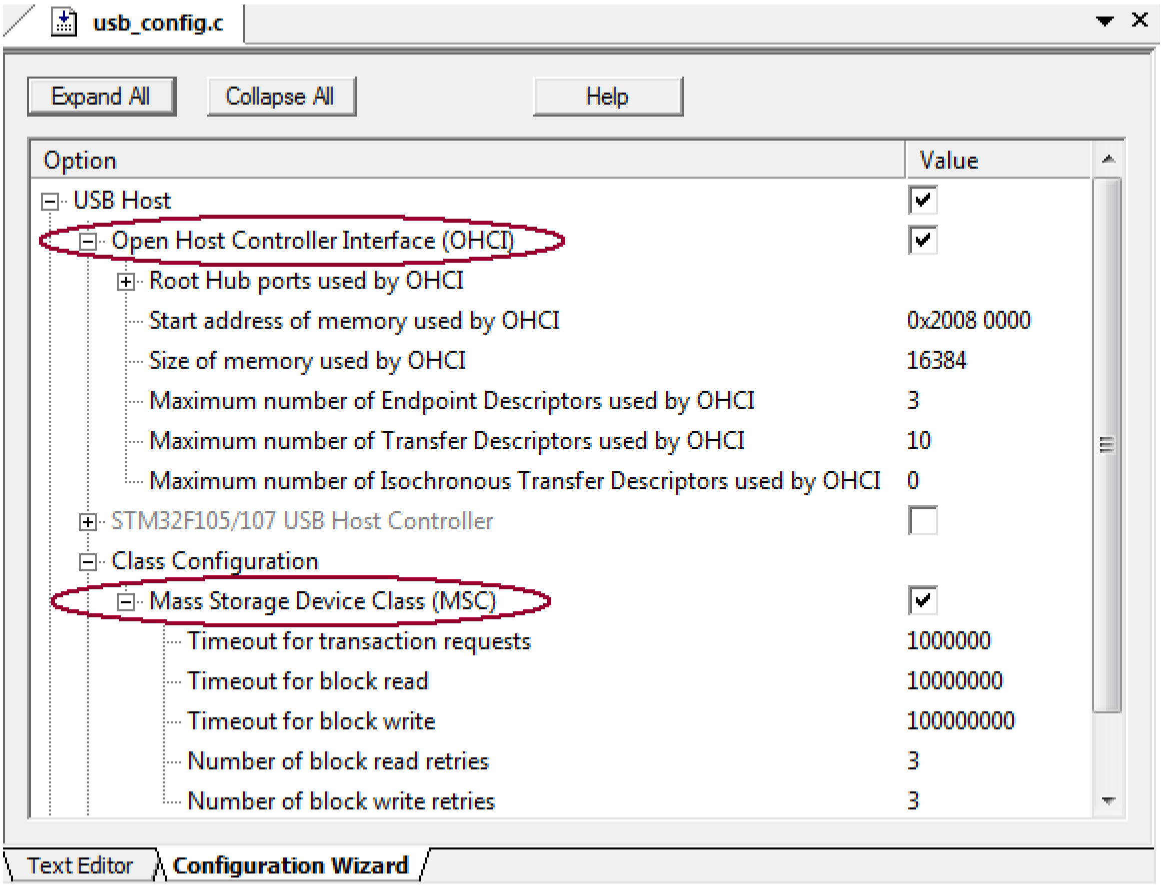
```

..
usbh_init();                                          // initialize USB Host Controller
while(1) {
  ..
  usbh_engine();                                      // check for new devices
  ..
}
```

Several µVision USB Host MSD projects can be used as a template for creating new USB Host MSD applications. The files can be edited directly or using the µVision Configuration Wizard.

1. Copy all files from any folder **\ARM\Boards\*Vendor*\Board\RL\USB\Host\MSD_File** to a new folder and open the ***.uvproj** project with µVision.
2. Open the file **usb_config.c** and set **USB Host**.

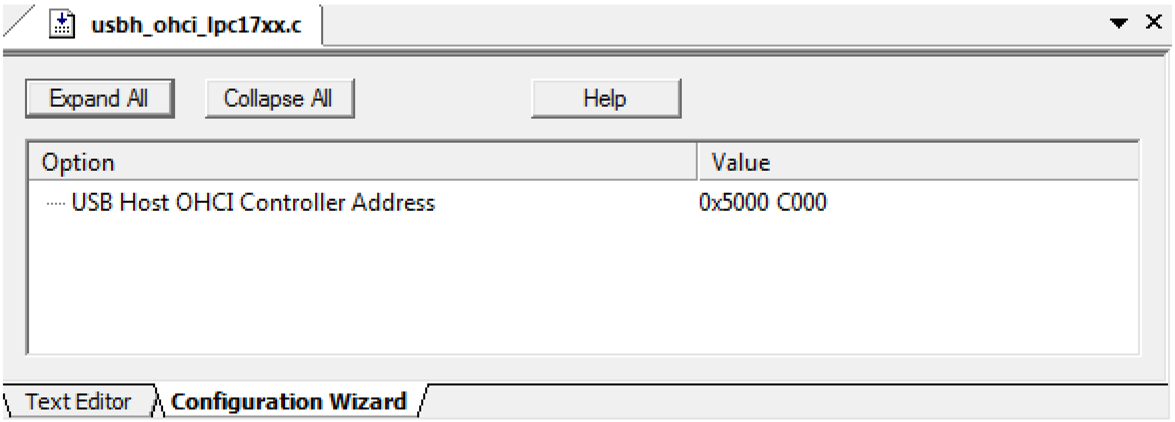
For USB Host devices that support OHCI:

3. Enable the option **Open Host Controller Interface (OHCI)**.



Option	Value
USB Host	☑
Open Host Controller Interface (OHCI)	☑
Root Hub ports used by OHCI	
Start address of memory used by OHCI	0x2008 0000
Size of memory used by OHCI	16384
Maximum number of Endpoint Descriptors used by OHCI	3
Maximum number of Transfer Descriptors used by OHCI	10
Maximum number of Isochronous Transfer Descriptors used by OHCI	0
STM32F105/107 USB Host Controller	□
Class Configuration	
Mass Storage Device Class (MSC)	☑
Timeout for transaction requests	1000000
Timeout for block read	10000000
Timeout for block write	100000000
Number of block read retries	3
Number of block write retries	3

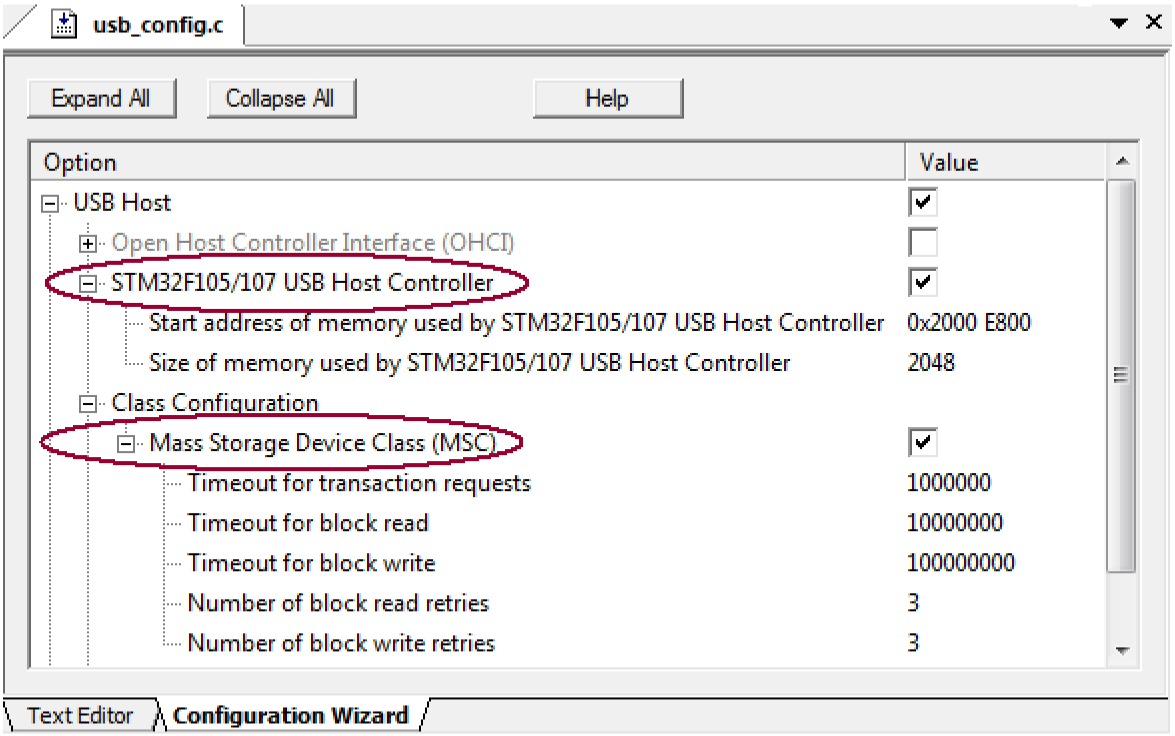
4. Set the configuration values.
5. Disable any custom USB Host Controller Driver.
6. In the **Class Configuration** section, set **Mass Storage Device Class (MSC)**, define the timeouts, and the number of read- and write retries.
7. Open the file **usbh_ohci_*device family*.c** and set the **USB Host OHCI Controller Address**.



Option	Value
USB Host OHCI Controller Address	0x5000 C000

For USB Host devices that support a Custom USB Host Controller:

3. Open the file **usb_config.c** and disable the option **Open Host Controller Interface (OHCI)** .

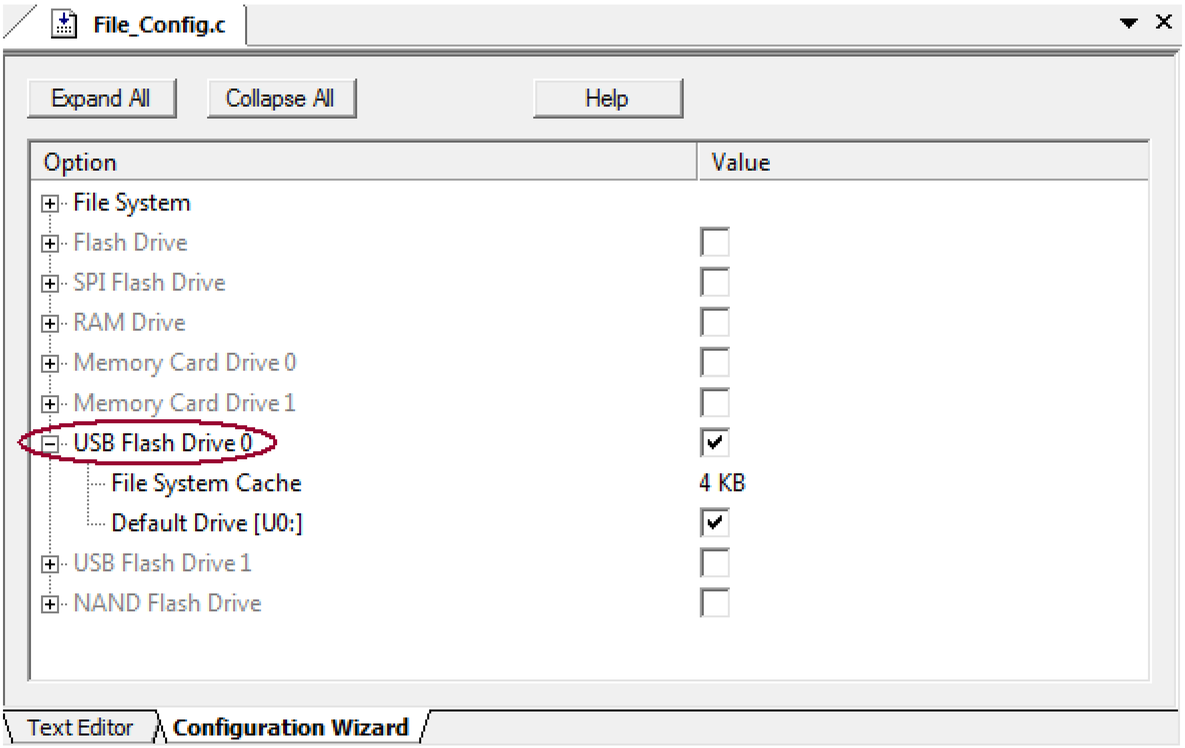


Option	Value
⊟ USB Host	☑
⊞ Open Host Controller Interface (OHCI)	☐
⊟ STM32F105/107 USB Host Controller	☑
Start address of memory used by STM32F105/107 USB Host Controller	0x2000 E800
Size of memory used by STM32F105/107 USB Host Controller	2048
⊟ Class Configuration	
⊟ Mass Storage Device Class (MSC)	☑
Timeout for transaction requests	1000000
Timeout for block read	10000000
Timeout for block write	100000000
Number of block read retries	3
Number of block write retries	3

4. Enable ***Custom USB Host Controller***.
5. Set the configuration options.
6. In the **Class Configuration** section, set **Mass Storage Device Class (MSC)**, define the timeouts, and the number of read and write retries.

Continue configuring the USB Flash Drive. Option details are explained in USB Flash Drive.

a. Open the file **File_Config.c**, enable the option **USB Flash Drive**.

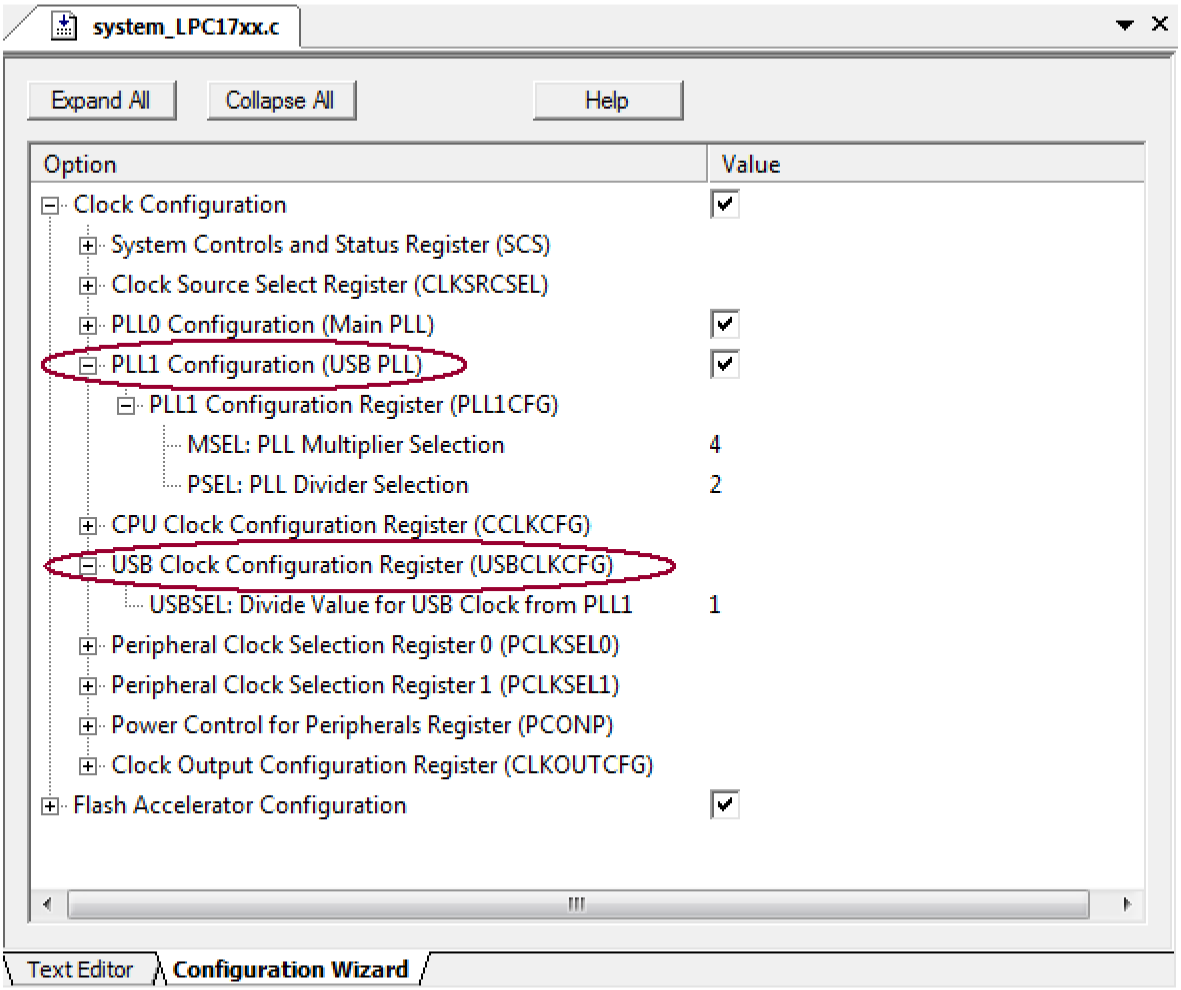


Option	Value
⊞ File System	
⊞ Flash Drive	☐
⊞ SPI Flash Drive	☐
⊞ RAM Drive	☐
⊞ Memory Card Drive 0	☐
⊞ Memory Card Drive 1	☐
⊟ USB Flash Drive 0	☑
File System Cache	4 KB
Default Drive [U0:]	☑
⊞ USB Flash Drive 1	☐
⊞ NAND Flash Drive	☐

b. Define the **File System Cache** size.
c. Enable **Default Drive [U0:]**.

Continue configuring the USB clocks:

a. Open the file **system_*device family*.c**, enable the **USB PLL** and configure the PLL-register.



Option	Value
Clock Configuration	☑
System Controls and Status Register (SCS)	
Clock Source Select Register (CLKSRCSEL)	
PLL0 Configuration (Main PLL)	☑
PLL1 Configuration (USB PLL)	☑
PLL1 Configuration Register (PLL1CFG)	
MSEL: PLL Multiplier Selection	4
PSEL: PLL Divider Selection	2
CPU Clock Configuration Register (CCLKCFG)	
USB Clock Configuration Register (USBCLKCFG)	
USBSEL: Divide Value for USB Clock from PLL1	1
Peripheral Clock Selection Register 0 (PCLKSEL0)	
Peripheral Clock Selection Register 1 (PCLKSEL1)	
Power Control for Peripherals Register (PCONP)	
Clock Output Configuration Register (CLKOUTCFG)	
Flash Accelerator Configuration	☑

b. Configure the **USB Clock Configuration Register**.

Finally, modify the file **fs_usbh_msc.c** to suite the application requirements.

Note

- The configuration options are explained in RL-USB Host Configuration.

USB Concepts

The **Universal Serial Bus (USB)** is a serial interface designed to be plug-and-play making it easy to connect peripherals to a host. Programmers can build sophisticated computing systems without having to worry about the underlying technology. To design USB peripherals, the programmer should understand the microcontroller firmware, the USB protocol, the USB device descriptors, and the USB host operating system.

This chapter contains the sections:

USB Transfer Rates

USB uses two wires to transfer power and two wires to transfer signals. Three data transfer rates are supported:

Performance	Attributes	Application
Low-Speed 1.5 Mbits/s	Lower cost Hot-pluggable Multiple peripherals	interactive devices: keyboard, mouse, game peripherals
Full-Speed 12 Mbits/s	Low cost Hot-pluggable Multiple peripherals Guaranteed latency Guaranteed bandwidth	phone, audio, compressed video, printers, scanners
High-Speed 480 Mbits/s	Guaranteed latency High bandwidth	video, mass storage

Note

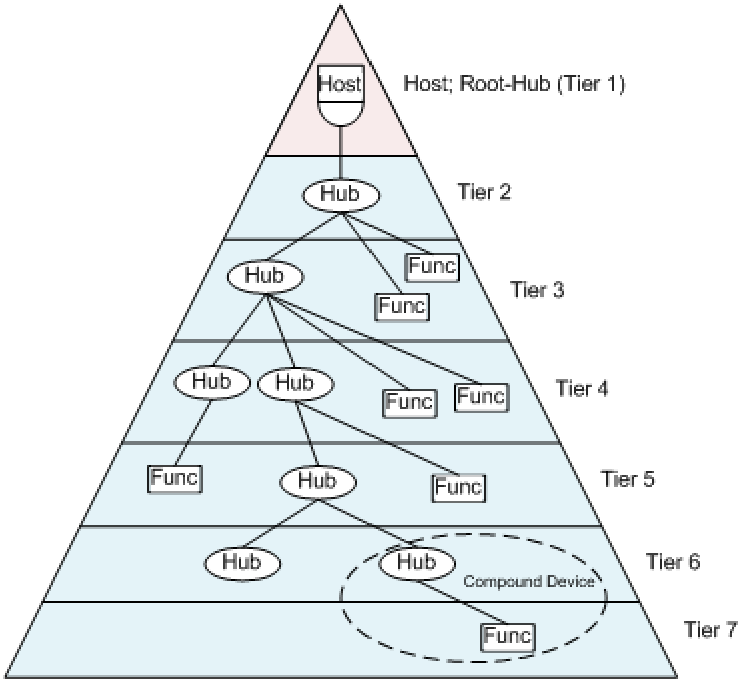
- SuperSpeed has been introduced in 2008. Devices supporting this speed are available since 2010.
- SuperSpeed - 5 Gbits/s - is not covered in this document.

USB Network

The **physical USB network** is implemented as a tiered star network with one host (master) and several devices (slaves).

The USB host provides one attachment port. If more peripherals are required, connect a hub to the root port to provide additional connection ports. The USB network can support up to 127 external nodes. Due to timing constraints for signal propagation, the maximum number of tiers allowed is seven:

- One tier for the host (bus master).
- Six tiers for hubs and devices.



USB devices are divided into device classes and can be:

- Hubs, which provide additional attachment points.
- Functions, which provide capabilities to the system.

Hubs serve to simplify USB connectivity from the user’s perspective. Each hub converts a single attachment point into multiple attachment points referred to as ports.

Functions are USB devices that transmit or receive data or control information. Each function contains configuration information describing the device capabilities and resource requirements.

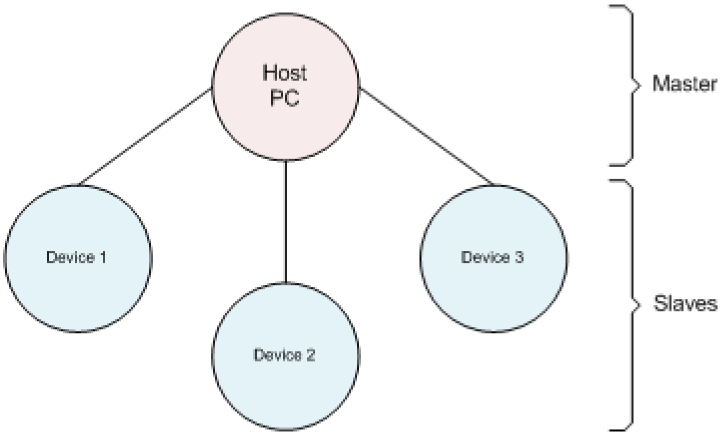
Compound Devices are physical packages that implement multiple functions and an embedded hub. A compound device appears to the host as a hub with one or more non-removable USB devices.

Composite Devices support more than one class and thus, provide more than one function to the host.

Examples of functions include the following:

- A human interface device such as a mouse, keyboard, tablet, or game controller.
- An imaging device such as a scanner, printer, or camera.
- A mass storage device such as a CD-ROM drive, floppy drive, or DVD drive.

The **logical USB network** appears as a star network to the developer with the host at the centre. Hubs do not introduce any programming complexity and are transparent as far as the programmer is concerned. A USB device will work the same way whether connected directly to a root-hub or whether connected via intermediate hubs. All USB devices are available as addressable nodes in this master/slave network. Only the host can initiate a data transfer in the network.



Note

- Only one host exists in any USB system.
- Only functions can be enabled in tier seven.
- Compound devices occupy two tiers.

Basic Communication Model

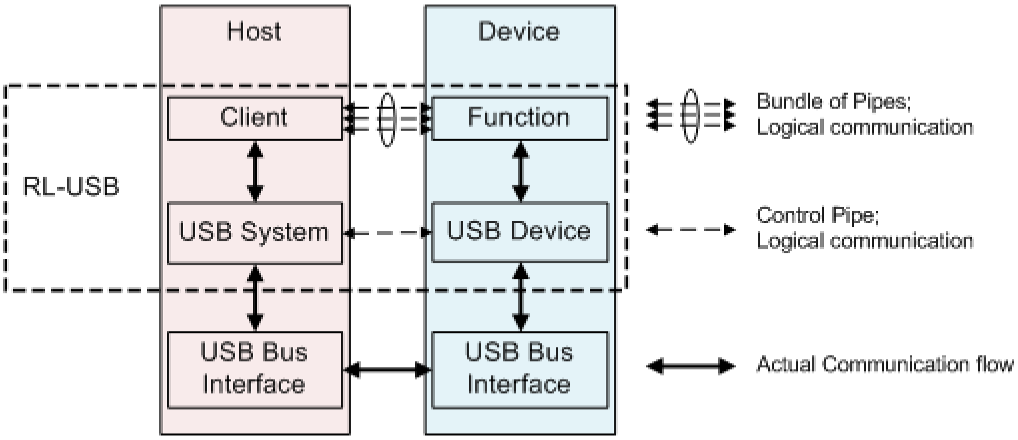
The host and the device have distinct layers, as shown in the picture below. Ultimately, the communication occurs on the physical USB wire. However, there are logical host-device interfaces between each horizontal layer.

The major host layers are the:

- **Client** describing the software entities responsible for interacting directly with USB devices.
- **USB System** managing data transfers between the host and USB devices.
- **USB Bus Interface** handling interactions of the electrical and protocol layers and actually transmits the packets.

Host layers have the role to:

- Detect the attachment and removal of USB devices.
- Manage USB control flows between the host and USB devices.
- Manage data flows between the host and USB devices.
- Collect status and activity statistics.
- Control the electrical interface.


Host
Device
RL-USB
Client
Function
USB System
USB Device
USB Bus Interface
USB Bus Interface
Bundle of Pipes; Logical communication
Control Pipe; Logical communication
Actual Communication flow

The major device layers are the:

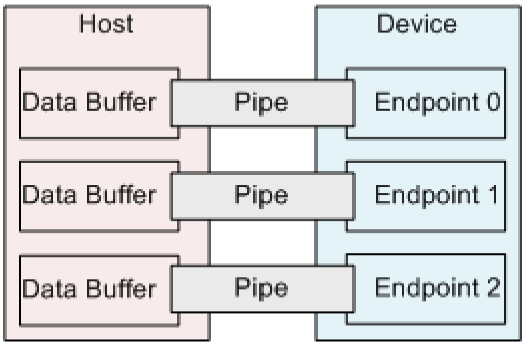
- **Function** providing a new capability, for instance, a mouse, a keyboard, an mp3-player, or an ISDN interface.
- **USB Device** managing data transfers between the host and USB devices.
- **USB Bus Interface** handling interactions of the electrical and protocol layers and actually transmits the packets.

USB Protocol

USB is a polled bus, where the host initiates all data exchanges. USB consists of several protocol layers, where data are transferred via a set of logical connections referred to as **pipes**. Two types of pipes exist:

- **Stream pipes** have no defined USB format. Stream pipes can either be controlled by the host or device. The data stream has a predefined direction, either IN or OUT. Stream pipes support **Bulk Transfers**, **Isochronous Transfers**, and **Interrupt Transfers**.
- **Message pipes** have a defined USB format. They are host controlled. Message pipes allow data to flow in both directions and support **Control Transfers** only.

Most pipes come into existence when a USB device has been connected and the signaling speed has been determined. A pipe originates from a buffer in the host and terminates inside the device at an endpoint.



Endpoints can be described as data sources or sinks. A device can have up to 16 OUT and 16 IN endpoints. An endpoint can have only one transfer direction. **Endpoint 0** is a special case and is a combination of Endpoint 0 OUT and Endpoint 0 IN. It is used to control the device.

OUT always refers to the direction pointing from the host to the device. **IN** always refers to the direction pointing towards the host.

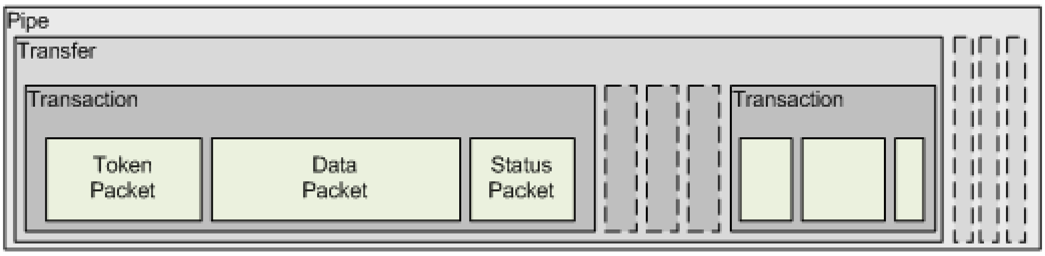
Transfers, or **data flow types**, can consist of one or more transactions. A pipe supports only one of the following transfer types:

- **Control Transfers** are typically used to setup a USB device. They are mandatory using Endpoint 0 IN/OUT.
- **Interrupt Transfers** can be used where data are sent regularly, for example status updates.
- **Isochronous Transfers** transmit real-time data such as audio and video. They have a guaranteed, fixed bandwidth.
- **Bulk Transfers** can be used to send data where timing is not important, for example to a printer.

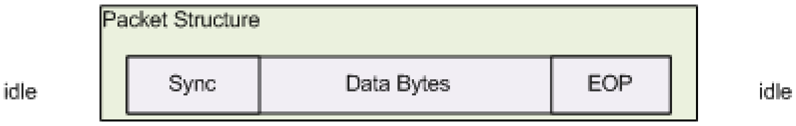
Transactions are transfers of data and mostly consist of three packets:

1. **Token packet** is the header defining the transaction type and direction, the device address, and the endpoint.
2. **Data packet** carries the information.
3. **Status packet** is a handshake packet informing whether the transfer was successful.

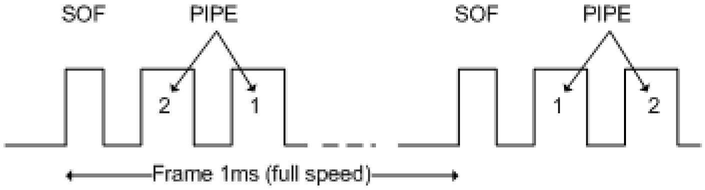
In a transaction, data are transferred either from the host to a device or from a device to the host. The transfer direction is specified in the token packet. Then, the source sends a data packet or indicates it has no data to transfer. In general, the destination responds with a status packet indicating whether the transfer was successful.



Packets could be thought of as the smallest element of data transmission. Each packet transmits an integral number of bytes at the current transmission rate. Packets starts with a synchronization pattern, followed by the data bytes of the packet, and concluded with an End of Packet (EOP) signal. All USB packet patterns are transmitted **least significant bit first**. Before and after the packet, the bus is in idle state.



A special packet is the **Start-of-Frame** packet (SOF) that splits the USB bus into time segments. Each pipe is allocated a slot in each frame. The Start-of-Frame packet is sent every 1ms on full speed links. At high speed, the 1ms frame is divided into 8 micro frames of 125 µs each. A Start-of-Frame packet is sent at the beginning of each micro frame with the same frame number. The frame number increments every 1 ms.



Control Transfer

Control Transfers are bi-directional transfers reserved for the host to send and request configuration information to and from the device using the IN and OUT Endpoint 0. Each Control Transfer consists of 2 to several transactions. The maximum packet size for the data stage is 8 bytes at low speed; 8, 16, 32, or 64 at full speed; and 64 for high speed. In general, the application software does not use this type of transfer.

Control Transfers have three stages:

- The **SETUP** stage carries 8 bytes called the Setup packet, defining the request, and specifying how many data should be transferred in the DATA stage.
- The **DATA** stage is optional. If present, it always starts with a transaction containing a DATA1 packet. Then, the transaction type alternates between DATA0 and DATA1 until all required data have been transferred.
- The **STATUS** stage is a transaction containing a zero-length DATA1 packet. If the DATA stage was IN, then the STATUS stage is OUT, and vice versa.

Interrupt Transfer

Interrupt Transfers have a limited latency to or from a device. In USB, an Interrupt Transfer, or Interrupt Pipe, has a defined polling rate between 1ms and 255ms. The developer can define how often the host can request a data transfer from the device.

For example, for a mouse, a data transfer rate at every 10 ms can be guaranteed. However, defining the polling rate does not guarantee that data will be transferred every 10 ms, but rather that the transaction will occur somewhere within the tenth frame. For this reason, a certain amount of timing jitter is inherent in a USB transaction.

Typically, Interrupt Transfer data consists of event notifications, characters, or coordinates from a pointing device.

Isochronous Transfer

Isochronous Transfers are used for transmitting real-time information such as audio and video data, and must be sent at a constant rate. USB isochronous data streams are allocated a dedicated portion of USB bandwidth to ensure that data can be delivered at the desired rate. An Isochronous pipe sends a new data packet in every frame, regardless of the success or failure of the last packet. No interrupt is generated when data arrive in the Endpoint buffer. Instead, the interrupt is raised on the Start-of-Frame token, which guarantees a regular 1ms interrupt on the Isochronous Endpoint.

Isochronous Transfers have no error detection. In other words, any error in electrical transmission is not corrected by hardware mechanisms such as retries.

Isochronous Transfers are also subject to timing jitters as described for Interrupt Transfers.

Bulk Transfer

Bulk Transfers are used for the data which are not of type Control, Interrupt, or Isochronous. Reliable exchange of data is ensured at the hardware level using error detection and invoking a limited number of retries in hardware.

Data are transferred in the same manner and with the same packet sizes as in Interrupt Transfers, but have no defined polling rate. Bulk Transfers take up all the bandwidth that is available after the other transfers have finished. If the bus is very busy, then a Bulk Transfer may be delayed. If the bus is idle, multiple Bulk Transfers can take place in a single 1ms frame (Interrupt and Isochronous Transfers are limited to a maximum of one packet per frame).

For example, Bulk Transfers send data to a printer. As long as the data is printed in a reasonable time frame, the exact transfer rate is not important.

Descriptors

USB devices report their attributes using **descriptors**. A descriptor is a data structure with a defined format. Each descriptor begins with a byte-wide field containing the total number of bytes in the descriptor followed by a byte-wide field identifying the descriptor type.

This is not a complete list of all the possible descriptors a USB host can request. However, as a minimum, the USB device must provide the device descriptor, configuration descriptor, interface descriptor, and three endpoint descriptors.

- Device Configuration explains the device configuration options and the structure of the descriptors.
- Device Descriptor describes the basic information that identifies a device.
- Configuration Descriptor describes the power requirements and the number of interfaces a device can contain.
- Interface Descriptor describes the collection of endpoints and the interfaces a device can have.
- Endpoint Descriptor specifies the attributes of an endpoint.
- Device Qualifier Descriptor describes the alternative information needed when the device operates in different speed modes.

The USB Host sends setup requests as soon as the device has joined the USB network. The device will be instructed to select a configuration and an interface to match the needs of the application running on the USB Host. Once a configuration and an interface have been selected, the device must service the active endpoints to exchange data with the USB Host.

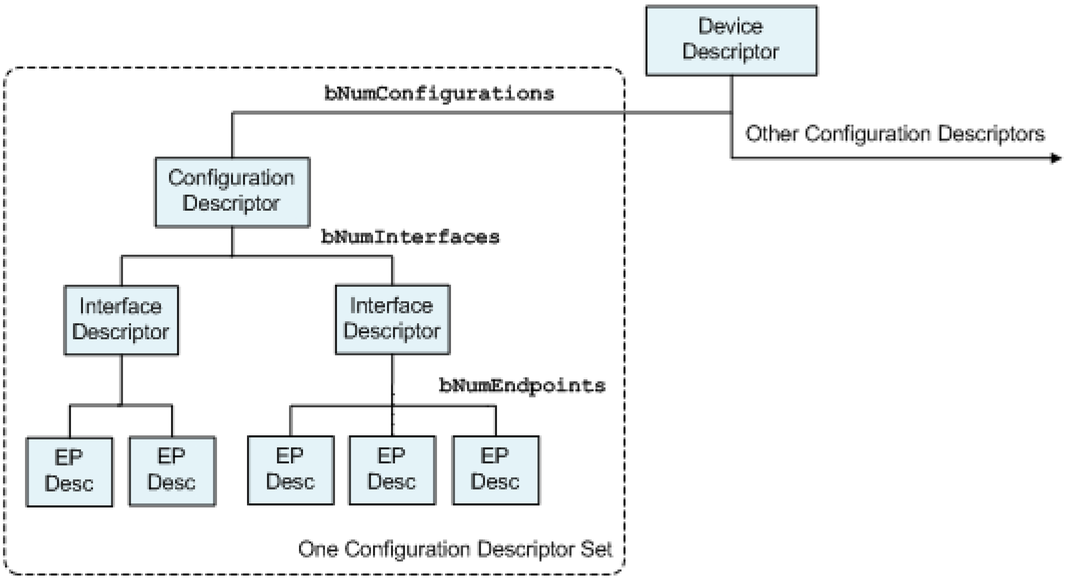
Device Configuration

When a USB device is attached to or removed from the USB, the host uses a process known as **bus enumeration** to identify and manage the device.

The data requested by the host are stored in a hierarchy of descriptors. Descriptors are arrays of data, which fully describe the device.

The minimum and required number of descriptors are:

- One Device Descriptor.
- One Configuration Descriptor.
- One Interface Descriptor.
- Three Endpoint Descriptors (one control, one IN, and one OUT).



Complex devices have multiple interfaces. Each interface can have a number of endpoints representing a functional unit. For example, a voice-over-IP phone might have:

- One audio class interface with 2 endpoints for transferring audio data in each direction.
- One HID interface with a single IN interrupt endpoint for a built-in keypad.

Device Descriptor

The **Device Descriptor** is the root of the descriptor tree and contains basic device information. The unique numbers, ***idVendor*** and ***idProduct***, identify the connected device. The Windows operating system uses these numbers to determine which device driver to load.

idVendor is the number assigned to each company producing USB-based devices. The USB Implementers' Forum is responsible for administering the assignment of Vendor IDs.

The ***idProduct*** is another 16-bit field containing a number assigned by the manufacturer to identify a specific product.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	Device Descriptor Type.
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	bDeviceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass field. If the bDeviceClass field is reset to zero, this field must also be reset to zero. If the bDeviceClass field is not set to FFH, all values are reserved for assignment by the USB-IF.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class specific protocols on a device basis. However, it may use class specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for Endpoint zero (only 8, 16, 32, or 64 are valid).
8	idVendor	2	ID	Vendor ID (assigned by the USB-IF).
10	idProduct	2	ID	Product ID (assigned by the manufacturer).
12	bcdDevice	2	BCD	Device release number in binary-coded decimal.

Offset	Field	Size	Value	Description
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer.
15	iProduct	1	Index	Index of string descriptor describing product.
16	iSerialNumber	1	Index	Index of string descriptor describing the device's serial number.
17	bNumConfigurations	1	Number	Number of possible configurations.

Configuration Descriptor

The **Configuration Descriptor** contains information about the device's power requirements and the number of interfaces it can support. A device can have multiple configurations. The host can select the configuration that best matches the requirements of the application software it is running.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	Configuration Descriptor Type.
2	wTotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration.
5	bConfigurationValue	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration.
6	iConfiguration	1	Index	Index of string descriptor describing this configuration.
7	bmAttributes	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in bMaxPower to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request. If a device configuration supports remote wakeup, D5 is set to one.
8	bMaxPower	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100 mA).

Interface Descriptor

The **Interface Descriptor** describes a collection of endpoints. This interface supports a group of pipes that are suitable for a particular task. Each configuration can have multiple interfaces. The interface can be selected dynamically by the USB host. The **Interface Descriptor** can associate its collection of pipes with a device class, which in turn has an associated class device driver within the host operating system. Typically, the device class is a functional type such as a printer class or mass storage class.

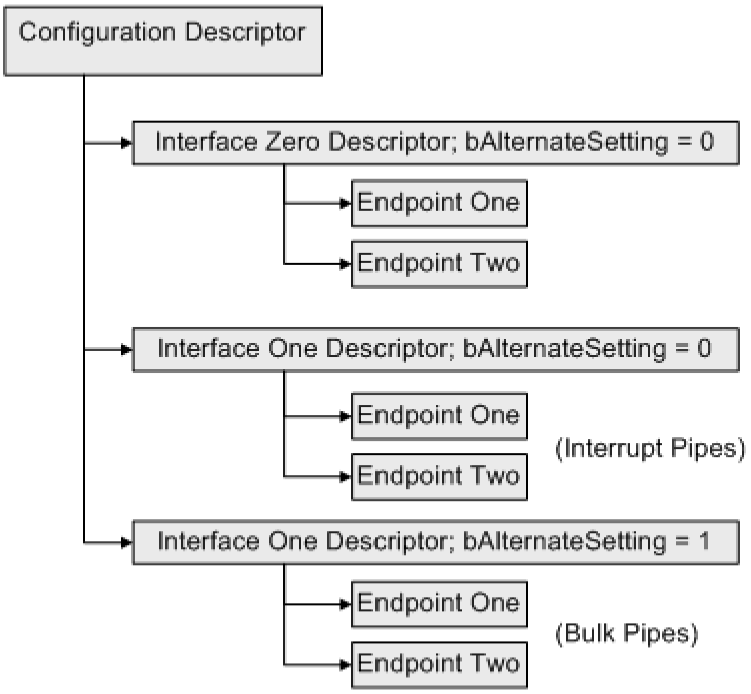
An interface descriptor never includes Endpoint 0 in the number of endpoints. If an interface uses only Endpoint 0, then the field *bNumEndpoints* must be set to zero.

If no class type has been selected for the device, then none of the standard USB drivers is loaded, and the developer has to provide its own device driver.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	Device Descriptor Type.
2	bInterfaceNumber	1	Number	The number of this interface. Zero-based value, identifying the index in the array of concurrent interfaces supported by this configuration.
3	bAlternateSetting	1	Number	Value used to select this alternate setting for the interface identified in the prior field. Allows an interface to change its settings on the fly.
4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses the Default Control Pipe.
5	bInterfaceClass	1	Class	Class code (assigned by the USB-IF). A value of zero is reserved for future standardization. If this field is set to FFH, the interface class is vendor-specific. All other values are reserved for assignment by the USB-IF.
6	bInterfaceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). If the bInterfaceClass field is reset to zero, this field must also be reset to zero. If the bInterfaceClass field is not set to FFH, all values are reserved for assignment by the USB-IF.
7	bInterfaceProtocol	1	Protocol	Protocol code (assigned by the USB). If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined in the device class. If this field is reset to zero, the device does not use a class-specific protocol on this interface. If this field is set to FFH, the device uses a vendor-specific protocol for this interface.
8	iInterface	1	Index	Index of string descriptor describing this interface.

For example, two device with different interfaces are needed. The first interface, *Interface Zero*, has the field *bInterfaceNumber* set to zero.
The next interface, *Interface One*, has the field *bInterfaceNumber* set to one and the field *bAlternativeSetting* also set to zero (default). It is possible to define an alternative setting for this device, by leaving the field *bInterfaceNumber* set to one, but with the field *bAlternativeSetting* is set to one instead of zero.

The first two interface descriptors with *bAlternativeSettings* equal to zero are used. However, the host can send a *SetInterface()* request to enable the alternative setting.


Configuration Descriptor
Interface Zero Descriptor; bAlternateSetting = 0
Endpoint One
Endpoint Two
Interface One Descriptor; bAlternateSetting = 0
Endpoint One
Endpoint Two
(Interrupt Pipes)
Interface One Descriptor; bAlternateSetting = 1
Endpoint One
Endpoint Two
(Bulk Pipes)

Endpoint Descriptor

The **Endpoint Descriptor** is used to specify the transfer type, direction, polling interval, and maximum packet size for each endpoint. Endpoint zero, the default endpoint, is always assumed to be a control endpoint and never has a descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	Endpoint Descriptor Type (05h).
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints. 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bitmap	The endpoint's attribute when configured through *bConfigurationValue*. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt For non-isochronous endpoints, bits 5..2 must be set to zero. For isochronous endpoints, they are defined as: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data 01 = Feedback 10 = Implicit feedback 11 = Reserved All other bits are reserved and must be reset to zero.
4	wMaxPacketSize	2	Number	Is the maximum packet size of this endpoint. For isochronous endpoints, this value is used to reserve the time on the bus, required for the per-(micro)frame data payloads. Bits 10..0 = max. packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 = number of additional transaction opportunities per micro-frame: 00 = None (1 transaction per micro-frame) 01 = 1 additional (2 per micro-frame) 10 = 2 additional (3 per micro-frame) 11 = Reserved Bits 15..13 are reserved and must be set to zero.
6	bInterval	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or micro-frames depending on the operating speed (1 ms, or 125 µs units).

Device Qualifier Descriptor

A high-speed capable device that has different device information for full-speed and high-speed must also have a **Device Qualifier Descriptor**. For example, if the device is currently operating at full-speed, the **Device Qualifier** returns information about how it would operate at high-speed and vice-versa.

The fields for the vendor, product, device, manufacturer, and serial number are not included. This information is constant for a device regardless of the supported speeds.

If a full-speed only device receives a *GetDescriptor()* request for a *device_qualifier*, it must respond with a request error. Then, the host must not make a request for an *other_speed_configuration descriptor*.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	Device Qualifier Type.
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant. At least V2.00 is required to use this descriptor.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	bDeviceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass field. If the bDeviceClass field is reset to zero, this field must also be reset to zero. If the bDeviceClass field is not set to FFH, all values are reserved for assignment by the USB-IF.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use classspecific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for other speed.
8	bNumConfigurations	1	Number	Number of other-speed configurations.

Offset	Field	Size	Value	Description
9	bReserved	1	Zero	Reserved for future use, must be zero.

Example Programs

The RL-ARM kit includes several example programs that are configured for ARM® devices.

The example programs are in the **\Keil\ARM\RL\...\Examples** and **\Keil\ARM\Boards\<vendor>\<board>\RL\...** folders. Each example program is stored in a separate folder along with project files that help you quickly build the projects and run the programs.

To begin using one of the example projects, use the µVision®3 **Project —> Open Project** menu and load the project file.

You can review the following projects when you start using RL-ARM:

Example	Component	Description
RTX_ex1	RL-RTX	A very simple program to control 2 tasks and to pass signals between the tasks.
RTX_ex2	RL-RTX	A more complex program to control 4 tasks and to pass signals between the tasks.
Blinky	RL-RTX	A simple LED flasher program to demonstrate Kernel I/O control.
Mailbox	RL-RTX	The Mailbox task inter-communication example program. Fixed size messages are sent using fixed memory block management functions.
Measure	RL-RTX	A data acquisition example program. The measure application is split into several tasks for easy control of the application.
Traffic	RL-RTX	A program that controls a traffic light.
RoundRobin	RL-RTX	Demonstrates Round-Robin task switching.
Semaphore	RL-RTX	Demonstrates how to use a shared resource (a UART) from different tasks using a semaphore.
SD_File	RL-FlashFS	A program that creates and manipulates with files located on the Memory Card.
MSD_File	RL-FlashFS	A program that creates and manipulates with files located on the USB Mass Storage Device.
File_ex1	RL-FlashFS	A program that creates and manipulates the SRAM files.
HTTP_demo	RL-TCPnet	An embedded Web Server demonstration program. This example shows you how to create web pages and use a CGI interface.
Telnet_demo	RL-TCPnet	An embedded Telnet Server demonstration program. This example shows you how to build a command-line interface and control an embedded device using Telnet.
TFTP_demo	RL-TCPnet RL-FlashFS	A file transfer over TCP/IP demonstration program. It uses a TFTP Server which manipulates files from the Flash File System.
SMTP_demo	RL-TCPnet	A program that sends an e-mail using SMTP Client application.
DNS_demo	RL-TCPnet	A program that resolves the IP address from a host name using the DNS Client application.
LEDSwitch	RL-TCPnet	A control embedded system from C++ application over TCP/IP to switch LED diodes on and off.
LEDSwitch\Client	RL-TCPnet	A control embedded system from another embedded system over TCP/IP to switch LED diodes on and off. This program replaces the C++ control application from the LEDSwitch example.

RTX_ex1

The RTX example program controls two tasks and passes signals between the tasks. This example is in the
\Keil\ARM\RL\RTX\Examples\RTX_ex1 folder.

The two tasks, named task1 and task2, are to repeat after a pause of 50 ms which starts when both tasks are complete. There is an additional pause of 20 ms between the completion of task1 and start of task2.

Follow these steps to build the application:

1. Place the code for the two activities into task1 and task2. Tasks can be declared in RVCT as a special type of function by using the C language extension key word **__task**.

```

__task void task1 (void) {
  .... code of task 1 placed here ....
}

__task void task2 (void) {
  .... code of task 2 placed here ....
}
```

10. Start the Real-Time Executive before running the tasks. Do this by calling the function **os_sys_init()** in the C main function.

    Pass the task ID as an argument to the os_sys_init function so that the task executes immediately, rather than the program continuing execution in the main function.

    Start task1 first. At the beginning of task1, use a call of **os_tsk_create** to start task2:

```

__task void task1 (void) {
  os_tsk_create (task2, 0);
  .... code of task 1 placed here ....
}

__task void task2 (void) {
  .... code of task 2 placed here ....
}

void main (void) {
  os_sys_init (task1);
}
```

11. Place the activities into endless loops to repeat indefinitely inside the two tasks.

    First, the system function **os_dly_wait()** pauses a task for a number of system intervals. The RTX kernel starts a system timer by programming one of the on-chip hardware timers of the ARM processor. By default, the system interval is 10 ms and timer 0 is used (you can change these settings).

    Then use the functions **os_evt_wait_or** and **os_evt_set** to wait for and set event flags.

    For this example, use the event flag at bit 2. The listing shown below contains all the statements required to run the RTX example:

```

/* Include type and function declarations for RTX */
#include "RTL.h"

/* id1, id2 will contain task identifications at run-time */
OS_TID id1, id2;
```

```
/* Forward reference. */
__task void task1 (void);
__task void task2 (void);

__task void task1 (void) {
  /* Obtain own system task identification number */
  id1 = os_tsk_self ();
  /* Assign system identification number of task2 to id2 */
  id2 = os_tsk_create (task2, 0);
  for (;;) {    /* do-this */
    /* Indicate to task2 completion of do-this */
    os_evt_set (0x0004, id2);
    /* Wait for completion of do-that (0xffff means no time-out)*/
    os_evt_wait_or (0x0004, 0xffff);
    /* Wait now for 50 ms */
    os_dly_wait (5);
  }
}

__task void task2 (void) {
  for (;;) {
    /* Wait for completion of do-this (0xffff means no time-out) */
    os_evt_wait_or (0x0004, 0xffff); /* do-that */
    /* Pause for 20 ms until signaling event to task1 */
    os_dly_wait (2);
    /* Indicate to task1 completion of do-that */
    os_evt_set (0x0004, id1);
  }
}

void main (void) {
  os_sys_init (task1);
}
```

12. Compile the example program and link it with the RTX function library. Select the RTX Operating System for the project under: **Options for Target —> Target —> Operating System —> RTX Kernel**. The final product (absolute file) can be run on your target or under the µVision Simulator in the normal way.

Traffic Example

The TRAFFIC example is a pedestrian traffic light controller that shows the usage of the multitasking RTX Real-time operating system. The traffic light operates during a user-defined time interval. Outside this time interval, the yellow light flashes. If a pedestrian pushes the request button, the traffic light goes immediately into walk state. Otherwise, the traffic light works continuously.

Traffic Light Controller Commands

The serial commands that the TRAFFIC controller supports are listed in the following table. These commands are composed of ASCII text characters. All commands must be terminated with a carriage return.

Command	Serial Text	Description
Display	D	Display clock, start, and end times.
Time	T hh:mm:ss	Set the current time in 24-hour format.
Start	S hh:mm:ss	Set the start time in 24-hour format. The traffic light controller operates normally between the start and end times. Outside these times, the yellow light flashes.
End	E hh:mm:ss	Set the end time in 24-hour format.

Software

The TRAFFIC application is composed of three files, which can be found in the \Keil\ARM\RL\RTX\Examples\Traffic folder.

TRAFFIC.C contains the traffic light controller program, which is divided into the following tasks:

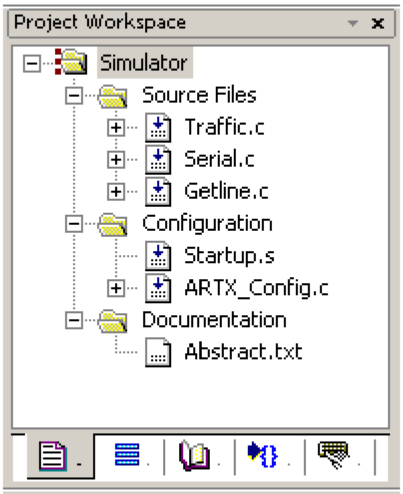
- Task **init**: initializes the serial interface and starts all other tasks. This task deletes itself since initialization is needed only once.
- Task **command**: is the command processor for the traffic light controller. This task controls and processes the received serial commands.
- Task **clock**: controls the time clock.
- Task **blinking**: flashes the yellow light when the clock time is outside the active time range.
- Task **lights**: controls the traffic light phases while the clock time is in the active time range (between the start and end times).
- Task **keyread**: reads the pedestrian push button and sends a signal to the task lights.
- Task **get_escape**: If an ESC character is encountered in the serial stream, the command task gets a signal to terminate the display command.

SERIAL.C implements an interrupt driven serial interface using events. This file contains the functions putchar and getkey. The high-level I/O functions printf and getline call these basic I/O routines. The traffic light application will also operate without using interrupt driven serial I/O but will not perform so well without it.

GETLINE.C is the command line editor for characters received from the serial port. This source file is also used by the MEASURE application.

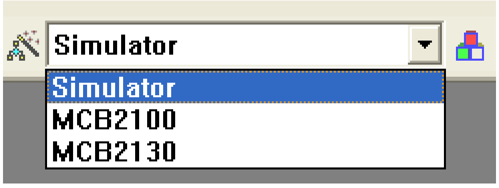
TRAFFIC Project

Open the TRAFFIC.UV2 project file that is located in the \Keil\ARM\RL\RTX\Examples\Traffic folder with µVision®. The source files for the TRAFFIC project are visible in the **Project Workspace** area. If the files are not visible in the Project Workspace area, then select the **Files** tab.



Select the RTX kernel Real-Time OS under **Options for Target**.

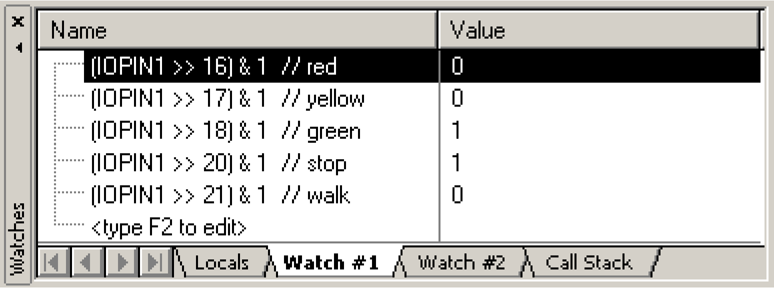
Build the TRAFFIC program with **Project —> Build** or the toolbar button. Select the target for build:



Run in Simulator

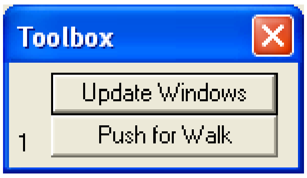
Start the µVision debugger.

The watch variables shown on the right allow you to view port status that drives the lights.

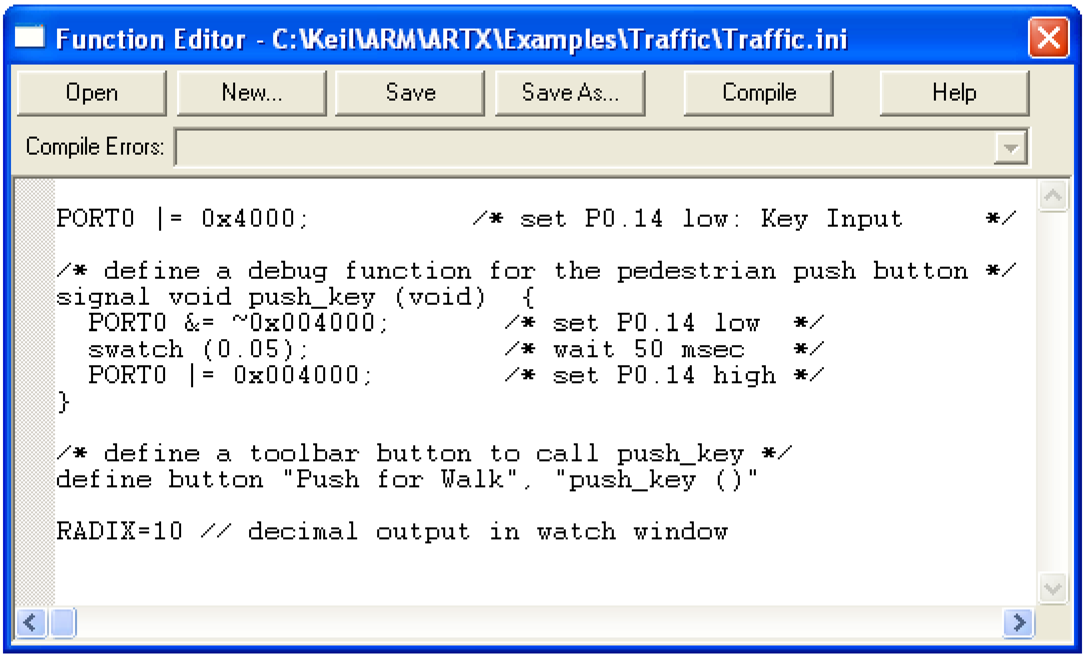


Name	Value
(IOPIN1 >> 16) & 1 // red	0
(IOPIN1 >> 17) & 1 // yellow	0
(IOPIN1 >> 18) & 1 // green	1
(IOPIN1 >> 20) & 1 // stop	1
(IOPIN1 >> 21) & 1 // walk	0
<type F2 to edit>	

The **push_key** signal function simulates the pedestrian push key that switches the light system to the walk state. This function is called with the Push for Walk toolbar button.



Use **Debug —> Function Editor** to open **TRAFFIC.INC**. This file is specified under **Options for Target —> Debug —> Initialization File** and defines the signal function **push_key**, the port initialization and the toolbar button.


Function Editor - C:\Keil\ARM\ARTX\Examples\Traffic\Traffic.ini
Open
New...
Save
Save As...
Compile
Help
Compile Errors:

```
PORT0 |= 0x4000;          /* set P0.14 low: Key Input    */

/* define a debug function for the pedestrian push button */
signal void push_key (void)  {
  PORT0 &= ~0x004000;        /* set P0.14 low  */
  swatch (0.05);             /* wait 50 msec   */
  PORT0 |= 0x004000;         /* set P0.14 high */
}

/* define a toolbar button to call push_key */
define button "Push for Walk", "push_key ()"

RADIX=10 // decimal output in watch window
```

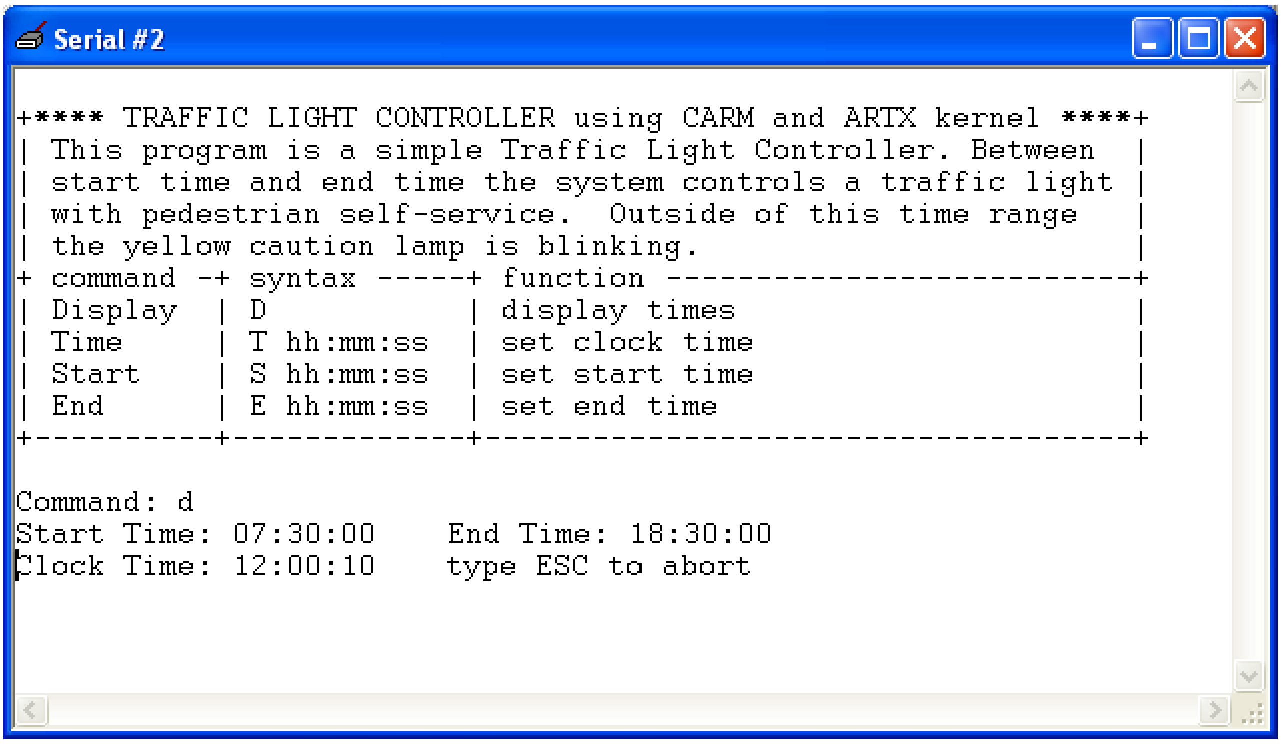
Note

- the VTREG symbol Clock is literalized with a back quote ('), since there is a C function named clock in the TRAFFIC.C module. Refer to "Literal Symbols" on page 132 for more information.

Now run the TRAFFIC application. Enable **View —> Periodic Window Update** to view the lights in the watch window during program execution.

The Serial Window #2 displays the printf output and allows you to enter the traffic light controller commands described in the table above.

Set the clock time outside of the active time interval to flash the yellow light.



```
+***** TRAFFIC LIGHT CONTROLLER using CARM and ARTX kernel *****+
| This program is a simple Traffic Light Controller. Between   |
| start time and end time the system controls a traffic light |
| with pedestrian self-service.  Outside of this time range   |
| the yellow caution lamp is blinking.                         |
+ command -+ syntax -----+ function ---------------------------+
| Display  | D           | display times                       |
| Time     | T hh:mm:ss  | set clock time                      |
| Start    | S hh:mm:ss  | set start time                      |
| End      | E hh:mm:ss  | set end time                        |
+----------+-------------+-------------------------------------+

Command: d
Start Time: 07:30:00    End Time: 18:30:00
Clock Time: 12:00:10    type ESC to abort
```

Run on MCB21xx

This example is configured for several targets. Select the **MCB2100** or the **MCB2130** evaluation board and rebuild the project. **ULINK**® USB-JTAG adapter is used for debugging. The application is loaded to the internal flash of the controller.

Download the HEX file into the on-chip Flash ROM of the LPC21xx device on the MCB21xx Evaluation Board.
Start the µVision debugger.

The **INT1** switch is used as a **push_key**. INT1 is configured as a standard digital input with jumpers **J1** and **J7**. Jumper positions are:

Jumper	Setting
J1	not inserted
J7	inserted

You can view the serial output and enter the traffic light controller **commands** with the Microsoft Windows HyperTerminal application: **Start —> Programs —> Accessories —> Communication —> HyperTerminal**.

Connect a serial cable to the COM1 serial port of the MCB21xx board and set the following serial parameters:

- **9600** bits per second
- **8** data bits
- Parity **None**
- Stop bits **1**
- Flow control **None**.

SD_File

The **SD_File** example program shows you how to use the **Flash File System** to store, retrieve, and manage files. This example is in the **\ARM\Boards\<vendor>\<board>\RL\FlashFS** folder.

This example is configured to use the **Secure Data Card** drive for storing files.

1. Load the project, select **Open Project** from the **Project** menu and open the **SD_File** project from the folder \ARM\Boards\<vendor>\<board>\RL\FlashFS\SD_File\.
2. Build the project.
3. Download example to target board.
4. Power-up the target board.
5. Enter the file commands from a serial window. A simple command-line interface is available with the following commands:

Command	Description
CAP *"fname" [/A]*	Capture serial data to ***fname*** file. ***/A*** option appends data to a ***fname*** file.
FILL *"fname" [nnnn]*	Create a ***fname*** file filled with text. ***nnnn*** specifies number of lines of text that will be filled in ***fname*** file (default is 1000).
TYPE *"fname"*	Display the content of a ***fname*** text file.
REN *"fname1" "fname2"*	Rename a ***fname1*** file to ***fname2*** file.
COPY *"fin" ["fin2"] "fout"*	Copy a ***fin*** file to ***fout*** file. ***fin2*** option merges ***fin*** and ***fin2*** file to ***fout*** file.
DEL *"fname"*	Delete a ***fname*** file.
DIR *["mask"]*	Displays the list of files and folders.
FORMAT *[label] /FAT32*	Formats the media and gives it ***label*** label. ***/FAT32*** option forces media to be formatted as FAT32.

Example of using command interface.

1. **FORMAT KEIL**
   - format the media and give it label **KEIL**
2. **FILL "M:\Test folder\Test file.txt"**
   - create a long file name **Test file.txt** in **Test folder** subfolder filled with text
3. **DIR "M:\Test folder*.*"**
   - display all the files in the **Test folder** subfolder
4. **TYPE "M:\Test folder\Test file.txt"**
   - display the content of the **Test file.txt** located in the **Test folder** subfolder
5. **REN "M:\Test folder\Test file.txt" "Test file renamed.txt"**
   - rename the file **Test file.txt** located in the **Test folder** subfolder to the **Test file renamed.txt** file in the same subfolder
6. **COPY "M:\Test folder\Test file renamed.txt" "M:\test.txt"**
   - copy the file **Test file renamed.txt** located in the **Test folder** subfolder to the **TEST.TXT** file in the root folder
7. **DEL "M:\Test folder\Test file renamed.txt"**
   - delete the **Test file renamed.txt** file located in the **Test folder** subfolder
8. **DEL "M:\Test folder\"**
   - delete the **Test folder** subfolder
9. **DIR "M:\*.*"**
   - display all files in the root folder
10. **TYPE "M:\test.txt"**
    - display the content of the **TEST.TXT** file located in the root folder

note

- **M:\** can be omitted if default drive in file **File_Config.c** is selected as **Memory Card**
- This is just an example of using Flash File System file manipulation functions. It does not emulate DOS nor should it be mistaken as DOS.

MSD_File

The **MSD_File** example application shows how to use the **Flash File System** to store, retrieve, and manage files. It is configured to use the **USB Mass Storage Device** drive for storing files. This example is located in the folder **\ARM\Boards\<vendor>\<board>\RL\USB\Host**.

1. Load the project. Select **Project - Open Project** from the µVision menu and open the project **MSD_File.uvproj** from the folder **\ARM\Boards\*Vendor**Board*\RL\USB\Host\MSD_File**.
2. Build the project.
3. Power-up and connect the target board.
4. Download application to target board.
5. Enter the file commands from a serial window. A simple command-line interface is available with the following commands:

Command	Description
CAP *"fname" [/A]*	Capture serial data to ***fname*** file. ***/A*** option appends data to a ***fname*** file.
FILL *"fname" [nnnn]*	Create a ***fname*** file filled with text. ***nnnn*** specifies number of lines of text that will be filled in ***fname*** file (default is 1000).
TYPE *"fname"*	Display the content of a ***fname*** text file.
REN *"fname1" "fname2"*	Rename a ***fname1*** file to ***fname2*** file.
COPY *"fin" ["fin2"] "fout"*	Copy a ***fin*** file to ***fout*** file. ***fin2*** option merges ***fin*** and ***fin2*** file to ***fout*** file.
DEL *"fname"*	Delete a ***fname*** file.
DIR *["mask"]*	Displays the list of files and folders.
FORMAT *[label] /FAT32*	Formats the media and gives it ***label*** label. ***/FAT32*** option forces media to be formatted as FAT32.

Example of using command interface.

1. **FORMAT KEIL**
   - format the media and give it label **KEIL**
2. **FILL "U:\Test folder\Test file.txt"**
   - create a long file name **Test file.txt** in **Test folder** subfolder filled with text
3. **DIR "U:\Test folder*.*"**
   - display all the files in the **Test folder** subfolder
4. **TYPE "U:\Test folder\Test file.txt"**
   - display the content of the **Test file.txt** located in the **Test folder** subfolder
5. **REN "U:\Test folder\Test file.txt" "Test file renamed.txt"**
   - rename the file **Test file.txt** located in the **Test folder** subfolder to the **Test file renamed.txt** file in the same subfolder
6. **COPY "U:\Test folder\Test file renamed.txt" "U:\test.txt"**
   - copy the file **Test file renamed.txt** located in the **Test folder** subfolder to the **TEST.TXT** file in the root folder
7. **DEL "U:\Test folder\Test file renamed.txt"**
   - delete the **Test file renamed.txt** file located in the **Test folder** subfolder
8. **DEL "U:\Test folder\"**
   - delete the **Test folder** subfolder
9. **DIR "U:\*.*"**
   - display all files in the root folder
10. **TYPE "U:\test.txt"**
    - display the content of the **TEST.TXT** file located in the root folder

note

- **U:\** can be omitted if default drive in file **File_Config.c** is selected as **USB Flash**
- This is just an example of using Flash File System file manipulation functions. It does not emulate DOS nor should it be mistaken as DOS.

File_ex1

The File_ex1 example program shows you how to use the **Flash File System** to store, retrieve, and manage files. This example is in the **\Keil\ARM\RL\FlashFS\Examples\File_ex1** folder.

This example is configured to use the RAM drive for storing files. A portion of the system RAM is reserved for storing files. This example also works in simulation mode.

Follow these steps to build the application:

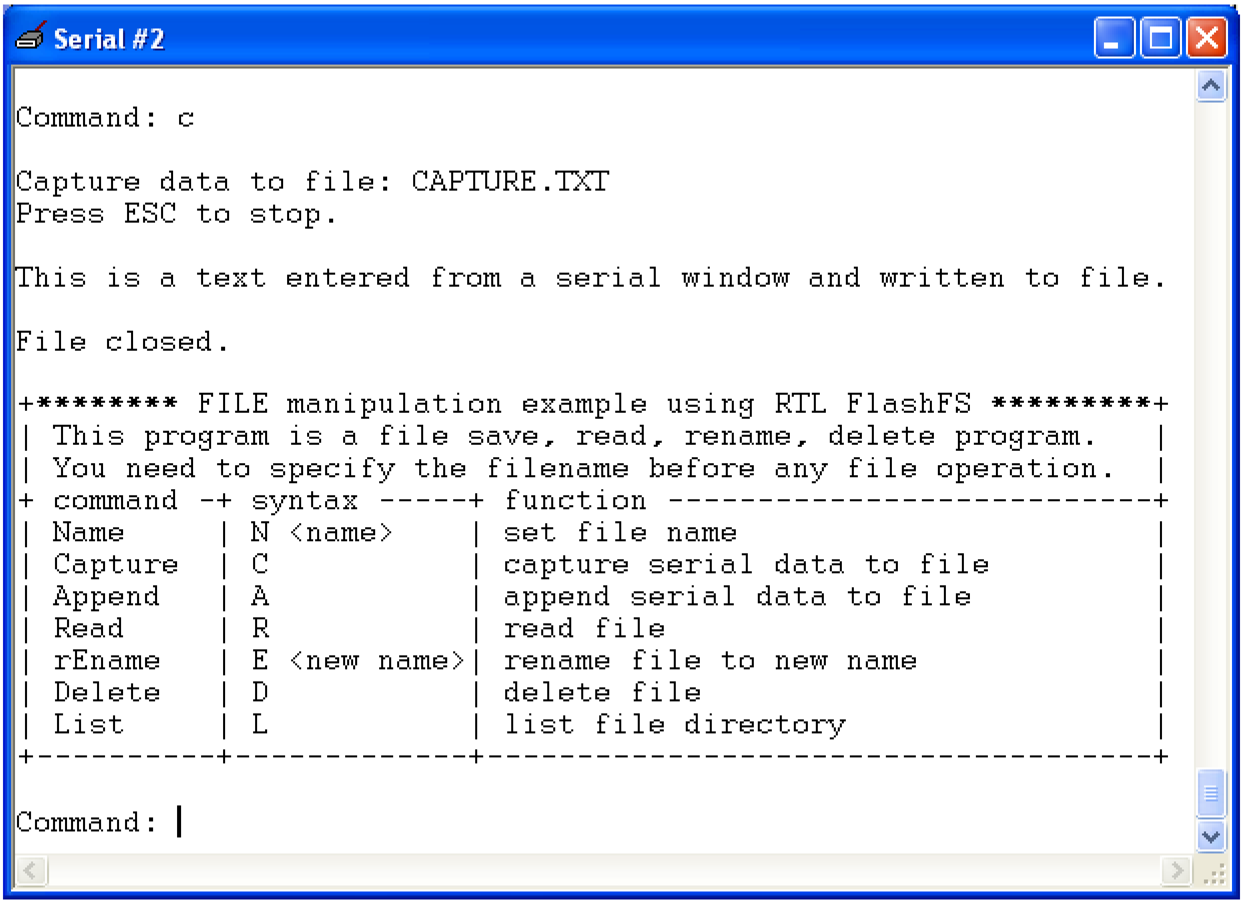
1. Load the project, select **Open Project** from the **Project** menu and open the **File_ex1** project from the folder \Keil\ARM\RL\FlashFS\Examples\File_ex1\.
2. Enter the file commands from a serial window. A simple command-line interface is available with the following commands:

Command	Description
N *name*	Set the file ***name*** for all file operations. This name is used for other commands.
C	Capture the text entered from a serial window to a file. Use the filename set by the **N** command.
A	Append the text entered from a serial window to the end of file.
R	Read a file content and output it to a serial window.
E *new*	Rename a file to a ***new*** filename. The filename set by the **N** command is renamed.
D	Delete a file.
L	List file directory.

3. Create a file by entering the capture command **C** from a serial window.
4.
5. `Command: C`

After this command is given, any text entered in the serial window writes to a file. The default filename is **CAPTURE.TXT**. However, you can change it using the **N** command before starting file capture.

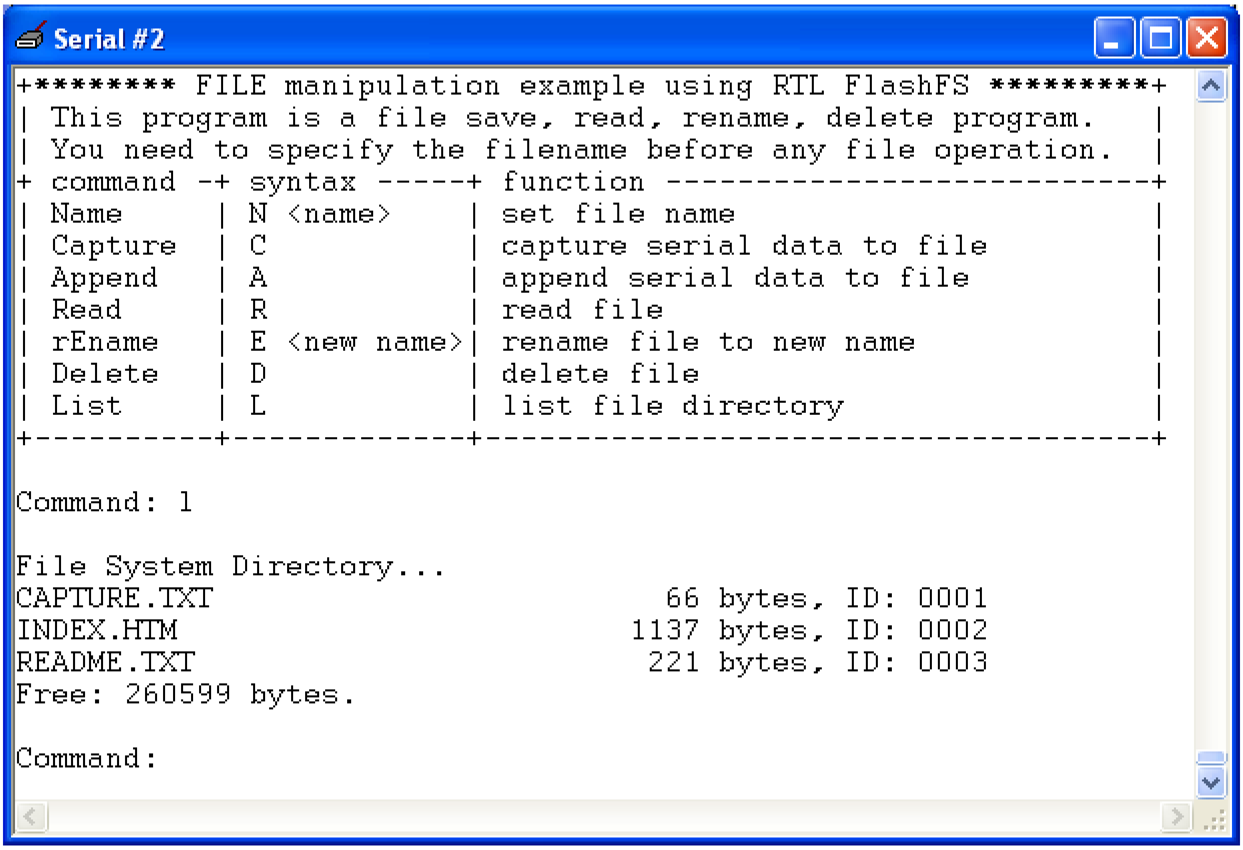
Use the **ESC** key to end the capture process.
Use the **R** command to read the file.



```
Command: c

Capture data to file: CAPTURE.TXT
Press ESC to stop.

This is a text entered from a serial window and written to file.

File closed.

+********* FILE manipulation example using RTL FlashFS *********+
| This program is a file save, read, rename, delete program.    |
| You need to specify the filename before any file operation.   |
+ command -+ syntax -----+ function ---------------------------+
| Name     | N <name>    | set file name                       |
| Capture  | C           | capture serial data to file         |
| Append   | A           | append serial data to file          |
| Read     | R           | read file                           |
| rEname   | E <new name>| rename file to new name             |
| Delete   | D           | delete file                         |
| List     | L           | list file directory                 |
+----------+-------------+-------------------------------------+

Command: |
```

6. You can create additional files by first defining a new filename and then starting the capture process. The next file is **INDEX.HTML**.
7.
8. `Command: N INDEX.HTML`
9. `Command: C`
10. Use the **L** command to display the list of all files stored to the SRAM.
11.
12. `Command: L`

The files appear as in the following example:



```
+********* FILE manipulation example using RTL FlashFS **********+
| This program is a file save, read, rename, delete program.    |
| You need to specify the filename before any file operation.   |
+ command -+ syntax -----+ function ----------------------------+
| Name     | N <name>    | set file name                        |
| Capture  | C           | capture serial data to file          |
| Append   | A           | append serial data to file           |
| Read     | R           | read file                            |
| rEname   | E <new name>| rename file to new name              |
| Delete   | D           | delete file                          |
| List     | L           | list file directory                  |
+----------+-------------+--------------------------------------+

Command: l

File System Directory...
CAPTURE.TXT                         66 bytes, ID: 0001
INDEX.HTM                         1137 bytes, ID: 0002
README.TXT                         221 bytes, ID: 0003
Free: 260599 bytes.

Command:
```

HTTP_demo

The **HTTP_demo** example program shows you how to use the **Web Server** application to control the Embedded system. It is configured for several different evaluation boards. This example is in the **\ARM\Boards\<vendor>\<name>\RL\TCPnet** folder.

Network Configuration

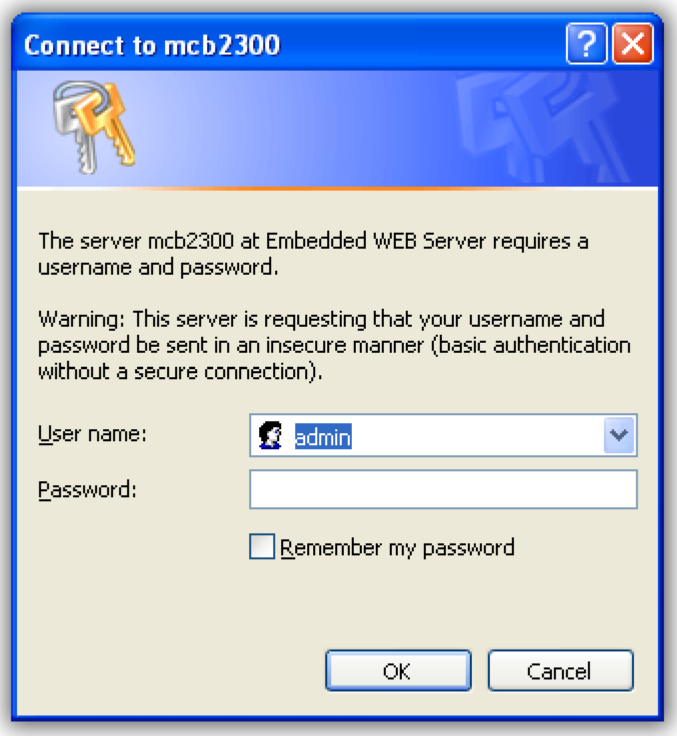
This example is configured to run on the Local Area Network (LAN). In order to run it, the network parameters like IP address, network mask, default gateway, and DNS Server IP address must be specified. You can specify these parameters in the following way:

- Automatically when **DHCP Client** is **enabled** in the **Net_Config.c** configuration file. This is the default setting for this example. You must have a running DHCP Server in your local area network. Your DHCP Server provides an IP address, network mask, default gateway, and DNS Server IP address for the embedded system automatically when this example starts.
- Static **IP address**, **network mask**, **Default Gateway**, and **DNS server** **configured** in the **Net_Config.c** configuration file. You must specify a free IP address and copy the netmask from the LAN configuration.
  Be careful defining these parameters because this example does not work if the network configuration parameters are misconfigured. A DNS Server IP address is not really needed for this example and can be left at 0.0.0.0.

Now you must connect a network cable and a JTAG cable for ULINK®. Then compile and link this example, download it to the target, and run it. This example downloads through the ULINK adapter from your computer to the target hardware SRAM or Flash memory, and it executes from there.

Testing the HTTP_demo

When you start the HTTP_demo example, you can connect to the embedded Web Server from your local computer using a web browser. Since **user authentication** is enabled by default, you must provide a username and a password when connecting. The default username is **admin**, and it has no password. Just press enter to connect.



Connect to mcb2300

The server mcb2300 at Embedded WEB Server requires a username and password.

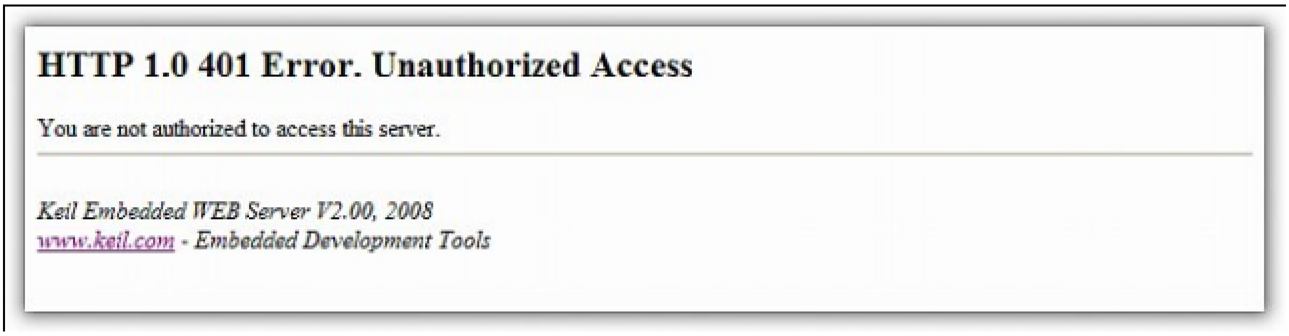
Warning: This server is requesting that your username and password be sent in an insecure manner (basic authentication without a secure connection).

User name: admin

Password:

□ Remember my password

If the username and password are not correct, the Web Server displays a warning page.

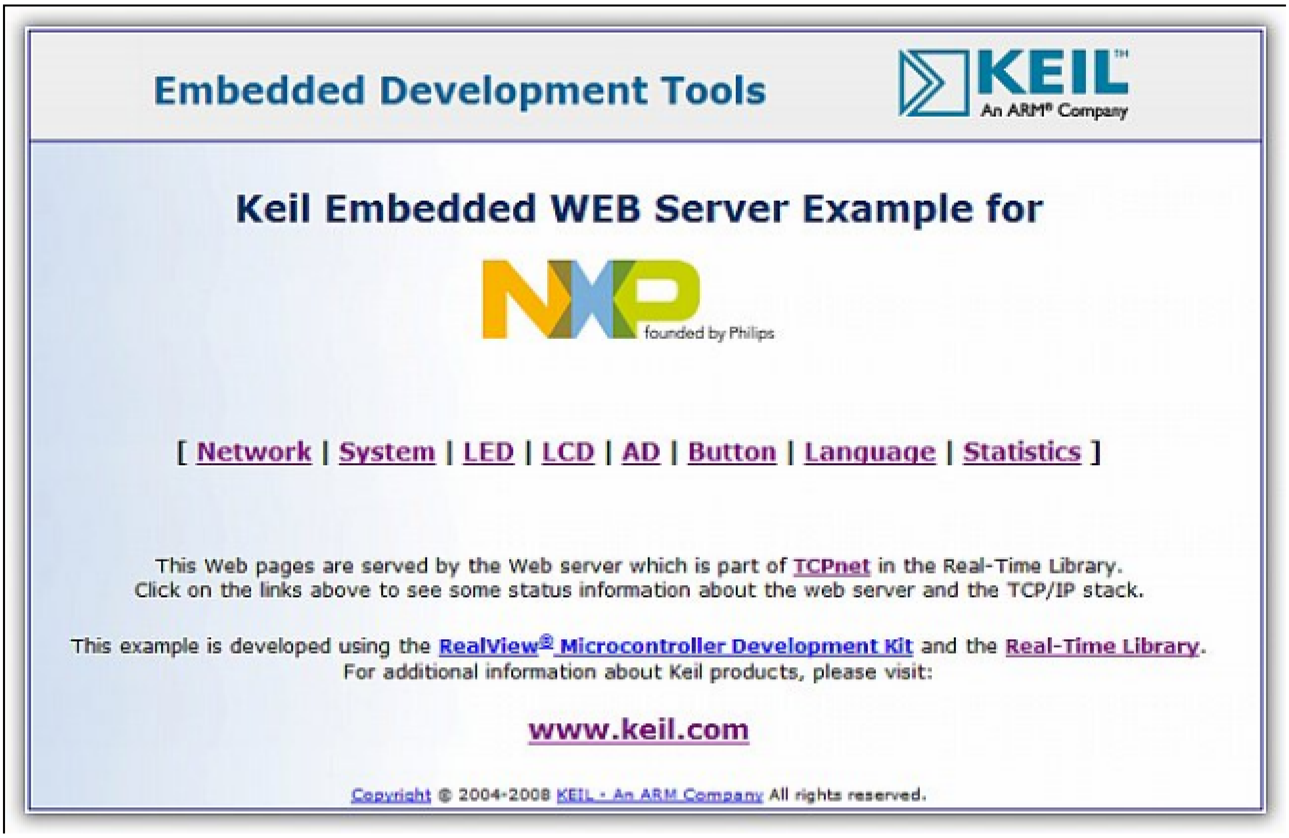


HTTP 1.0 401 Error. Unauthorized Access

You are not authorized to access this server.

Keil Embedded WEB Server V2.00, 2008
www.keil.com - Embedded Development Tools

After a successful login, the default page (**index.htm**) is shown. From here several pages can be selected.



Embedded Development Tools

Keil Embedded WEB Server Example for

[Network | System | LED | LCD | AD | Button | Language | Statistics]

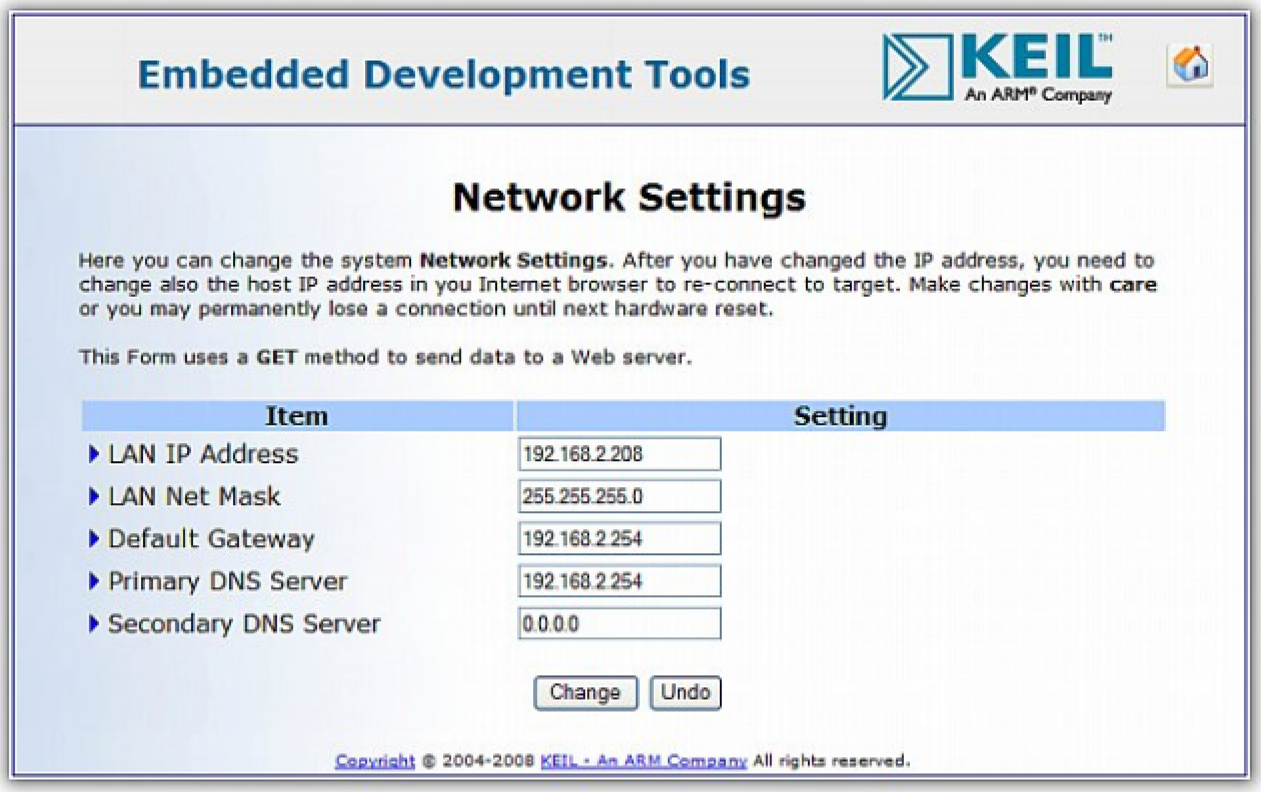
This Web pages are served by the Web server which is part of TCPnet in the Real-Time Library.
Click on the links above to see some status information about the web server and the TCP/IP stack.

This example is developed using the RealView® Microcontroller Development Kit and the Real-Time Library.
For additional information about Keil products, please visit:

www.keil.com

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **Network Settings** page allows you to change the network parameters. After the parameters are changed, the new settings are active immediately. So be careful when making changes because your example might stop responding.



Embedded Development Tools

KEIL An ARM® Company

Network Settings

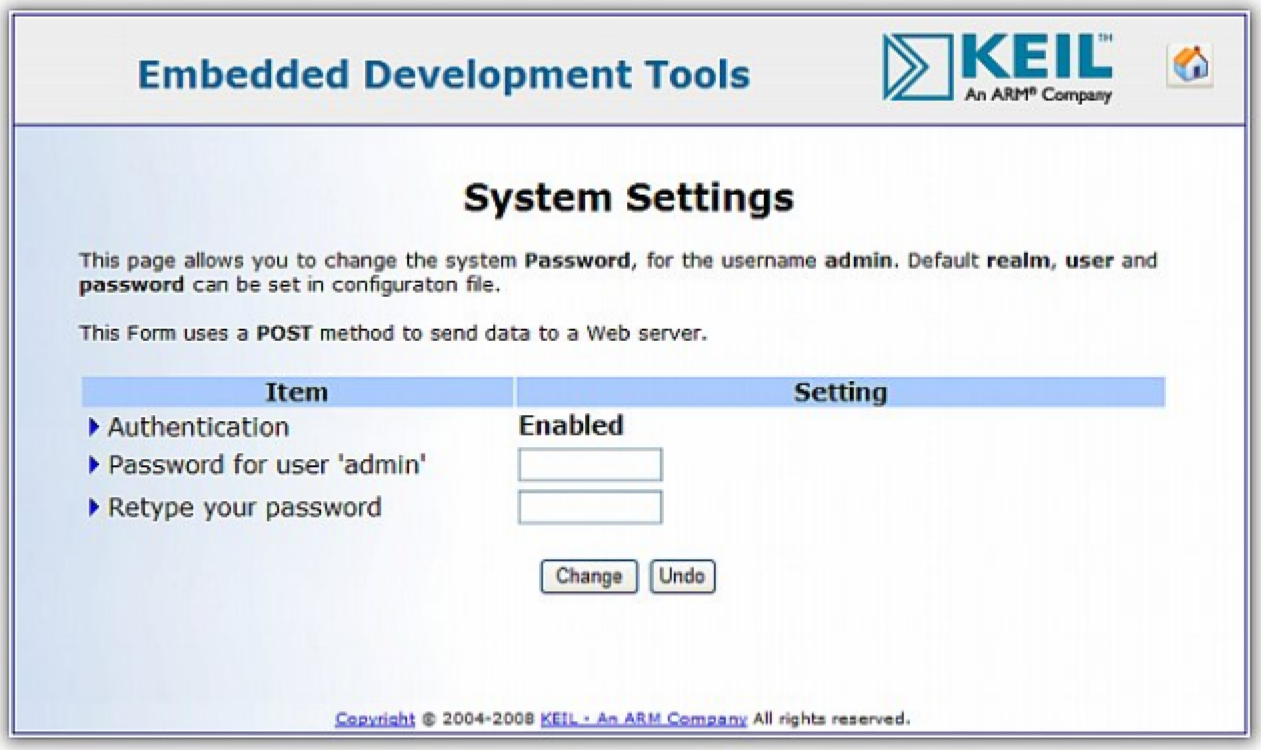
Here you can change the system **Network Settings**. After you have changed the IP address, you need to change also the host IP address in you Internet browser to re-connect to target. Make changes with **care** or you may permanently lose a connection until next hardware reset.

This Form uses a **GET** method to send data to a Web server.

Item	Setting
LAN IP Address	192.168.2.208
LAN Net Mask	255.255.255.0
Default Gateway	192.168.2.254
Primary DNS Server	192.168.2.254
Secondary DNS Server	0.0.0.0

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **System Settings** page allows you to change the system authentication **password**. After a new value is entered, you are immediately prompted for a new username and password to access the web pages on the embedded server. It is not possible to change the username dynamically. It can be changed in the Net_Config.c configuration file.



Embedded Development Tools

KEIL An ARM® Company

System Settings

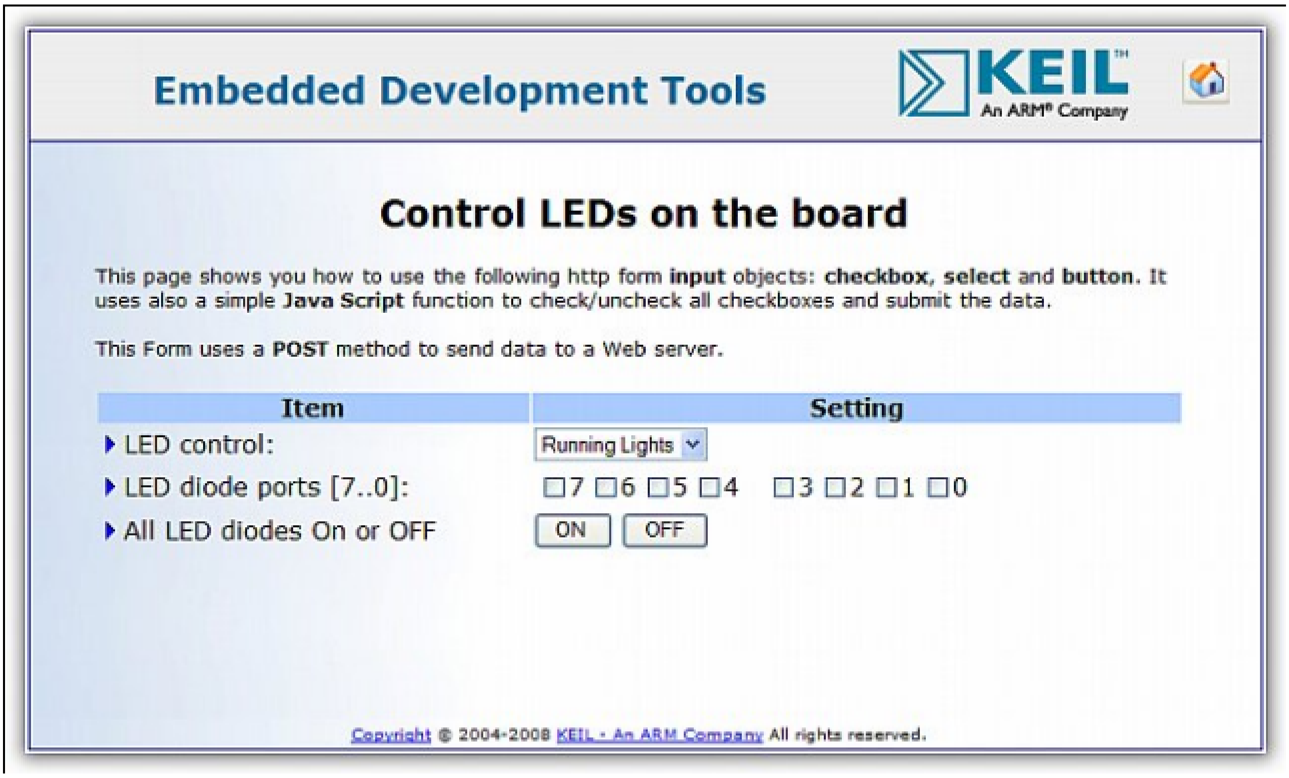
This page allows you to change the system **Password**, for the username **admin**. Default **realm**, **user** and **password** can be set in configuraton file.

This Form uses a **POST** method to send data to a Web server.

Item	Setting
Authentication	**Enabled**
Password for user 'admin'	
Retype your password	

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **LED** page allows you to control the status of the LED diodes on the evaluation board. You can switch them on and off, or enable or disable the running lights.



Embedded Development Tools

Control LEDs on the board

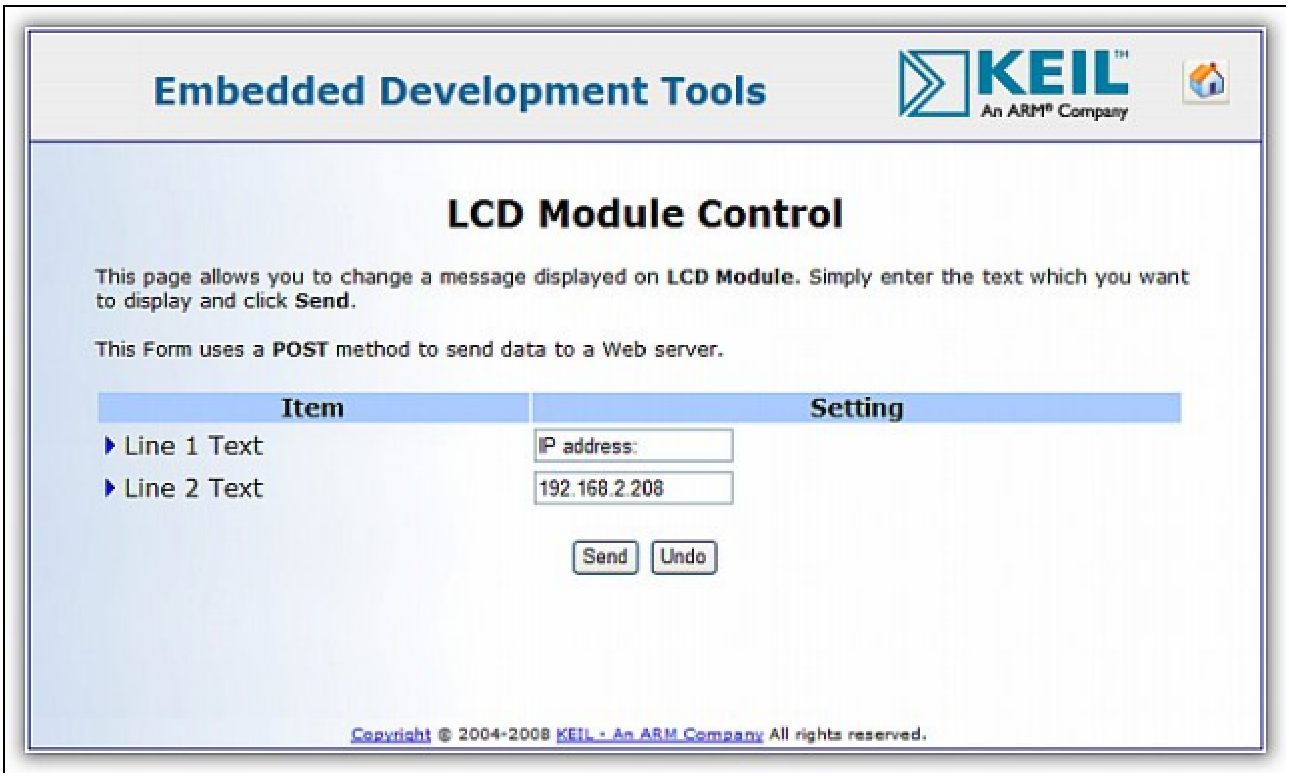
This page shows you how to use the following http form **input** objects: **checkbox**, **select** and **button**. It uses also a simple **Java Script** function to check/uncheck all checkboxes and submit the data.

This Form uses a **POST** method to send data to a Web server.

Item	Setting
LED control:	Running Lights
LED diode ports [7..0]:	□7 □6 □5 □4 □3 □2 □1 □0
All LED diodes On or OFF	ON OFF

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **LCD** page allows you to change the text displayed on the LCD module of the evaluation board.



Embedded Development Tools

LCD Module Control

This page allows you to change a message displayed on **LCD Module**. Simply enter the text which you want to display and click **Send**.

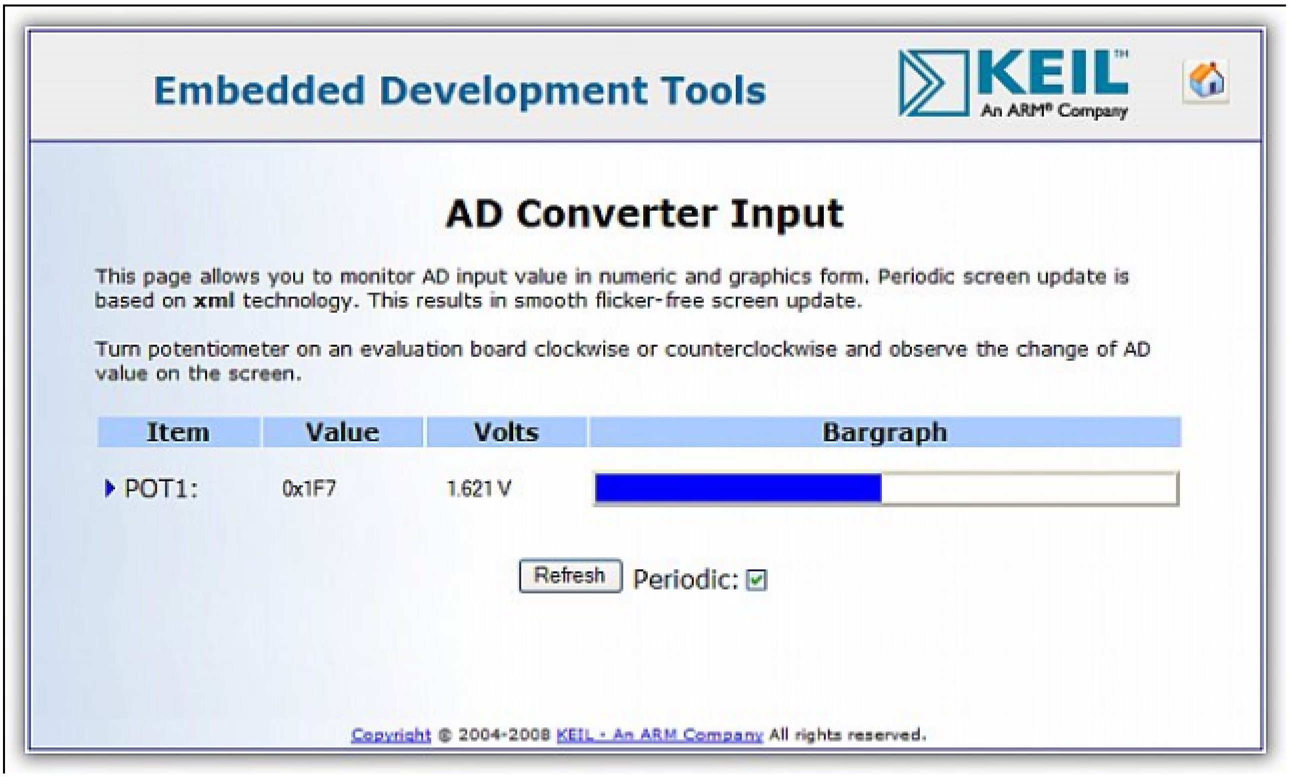
This Form uses a **POST** method to send data to a Web server.

Item	Setting
Line 1 Text	IP address:
Line 2 Text	192.168.2.208

Send Undo

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **AD** page displays the voltage of analog input. You must enable the **Periodic** update on the page to watch the change of analog voltage in real time.



AD Converter Input

This page allows you to monitor AD input value in numeric and graphics form. Periodic screen update is based on **xml** technology. This results in smooth flicker-free screen update.

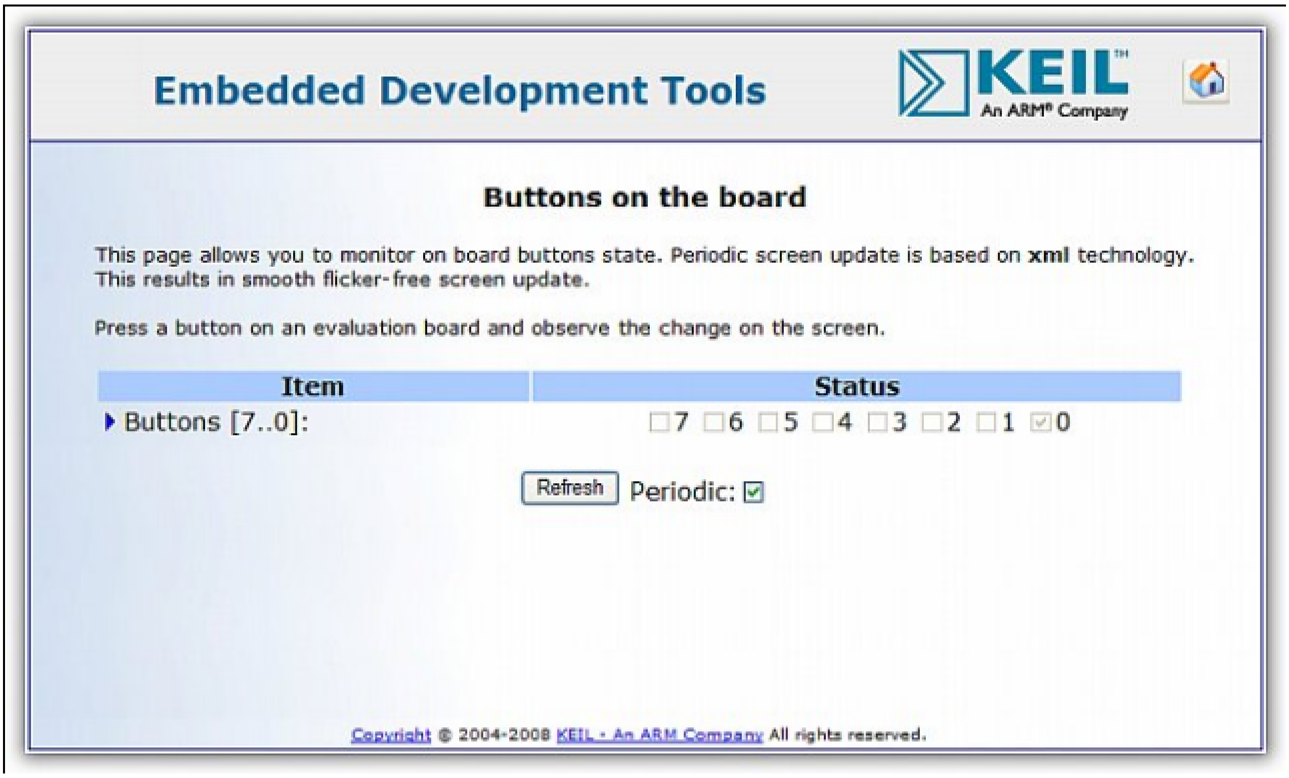
Turn potentiometer on an evaluation board clockwise or counterclockwise and observe the change of AD value on the screen.

Item	Value	Volts	Bargraph
▸ POT1:	0x1F7	1.621 V	

Refresh Periodic: ☑

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **Button** page displays the status of push buttons on the evaluation board. You must enable the **Periodic** update on the page to watch the status change in real time.



Buttons on the board

This page allows you to monitor on board buttons state. Periodic screen update is based on **xml** technology. This results in smooth flicker-free screen update.

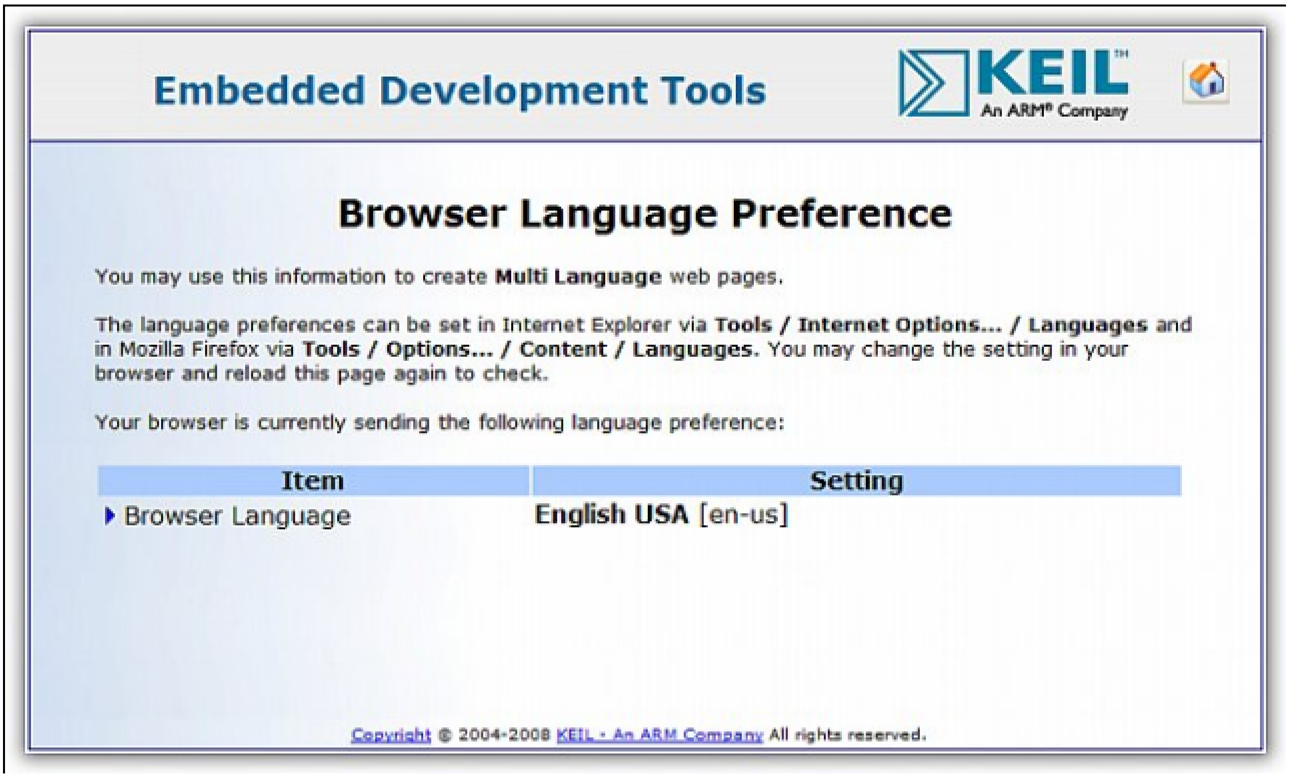
Press a button on an evaluation board and observe the change on the screen.

Item	Status
▸ Buttons [7..0]:	□7 □6 □5 □4 □3 □2 □1 ☑0

Refresh Periodic: ☑

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **Language** page displays the language preferences of your browser. You can use this information in your own application to support **multi-language** web pages.



Browser Language Preference

You may use this information to create **Multi Language** web pages.

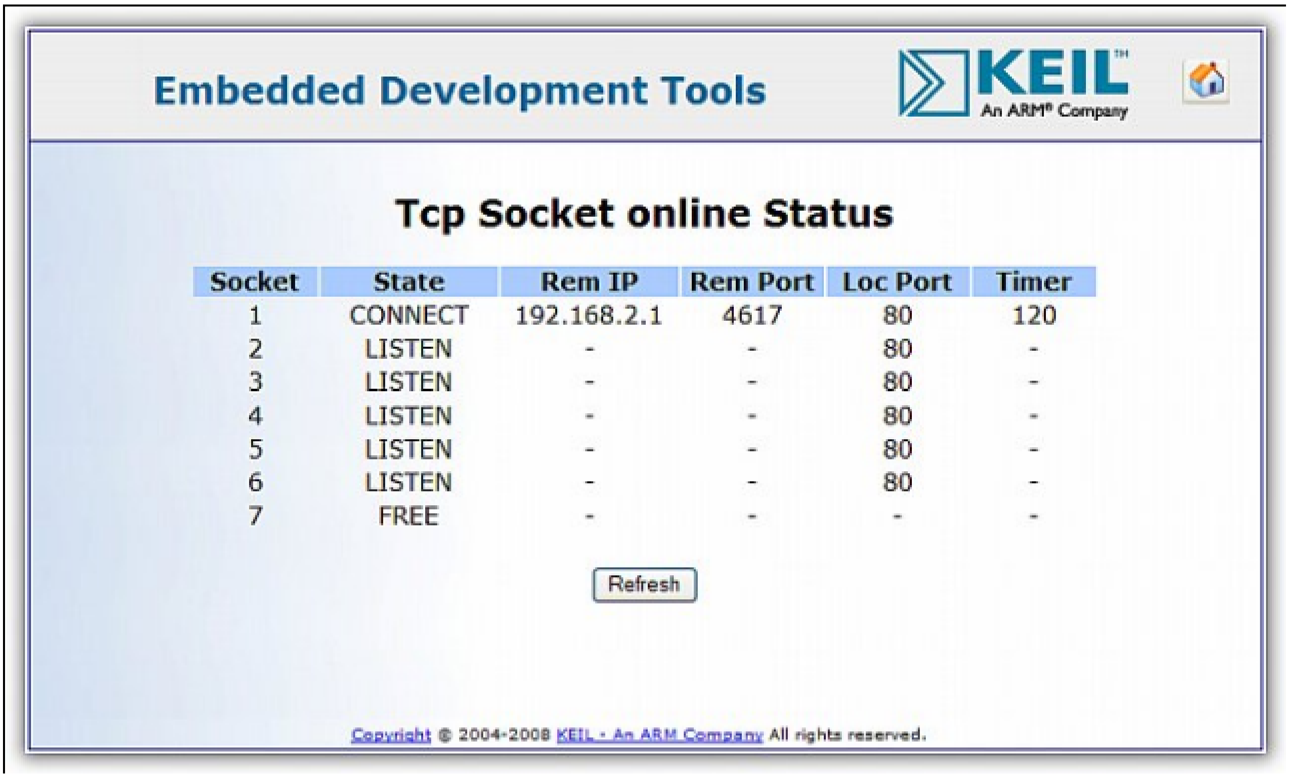
The language preferences can be set in Internet Explorer via **Tools / Internet Options... / Languages** and in Mozilla Firefox via **Tools / Options... / Content / Languages.** You may change the setting in your browser and reload this page again to check.

Your browser is currently sending the following language preference:

Item	Setting
▸ Browser Language	**English USA** [en-us]

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

The **Statistics** page displays the online status of TCP sockets. This web page refreshes every 5 seconds and displays the current status of all TCP sockets.



Tcp Socket online Status

Socket	State	Rem IP	Rem Port	Loc Port	Timer
1	CONNECT	192.168.2.1	4617	80	120
2	LISTEN	-	-	80	-
3	LISTEN	-	-	80	-
4	LISTEN	-	-	80	-
5	LISTEN	-	-	80	-
6	LISTEN	-	-	80	-
7	FREE	-	-	-	-

Copyright © 2004-2008 KEIL - An ARM Company All rights reserved.

Note

- You must type a different **address** on the web browser to access the web page on different evaluation boards. Read the **abstract.txt** file for details.

Telnet_demo

The **Telnet_demo** example program shows you how to use the **Telnet Server** application to control the Embedded system. This example program is in the **\Keil\ARM\Boards\Phytec\LPC229x\RL\TCPnet** folder.

The Telnet_demo **command-line** interface has the following commands:

Command	Description
led *xx*	Write the hex value ***xx*** to the LED port to switch the LED diodes on and off.
led	Re-enable the running lights that were stopped by the previous **led *xx*** command.
adin *x*	Read the Analog-to-Digital converter input ***x***. The range for ***x*** is from 0 to 7.
meas *n*	Display ***n*** measurements. A measurement is a sampled analog input.
rinfo	Display the remote machine's **IP** and **MAC** address.
tcpstat	Display the TCP status. This page is continuously updated.
passw *newp*	Change the system login password to the new password ***newp***.
passwd	Display the current password.
help **?**	Display command help.
bye **<ESC>**	Disconnect the telnet connection.

Network Configuration

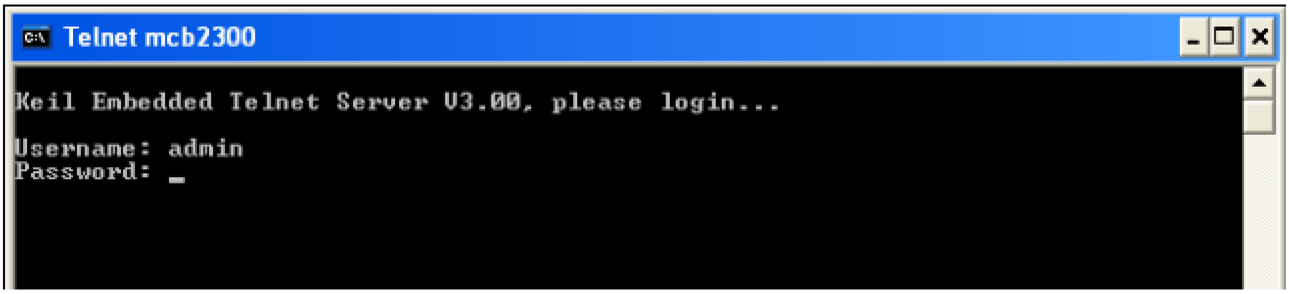
This example is configured to run on the Local Area Network (LAN). In order to run it, the network parameters like IP address, network mask, default gateway, and DNS Server IP address must be specified. You can specify these parameters in the following way:

- Automatically when **DHCP Client** is **enabled** in the **Net_Config.c** configuration file. This is the default setting for this example. You must have a running DHCP Server in your local area network. Your DHCP Server provides an IP address, network mask, default gateway, and DNS Server IP address for the embedded system automatically when this example starts.
- Static **IP address**, **network mask**, **Default Gateway**, and **DNS server** **configured** in the **Net_Config.c** configuration file. You must specify a free IP address and copy the netmask from the LAN configuration.
  Be careful defining these parameters because this example does not work if the network configuration parameters are misconfigured. A DNS Server IP address is not really needed for this example and can be left at 0.0.0.0.

Now you need to connect a network cable and a serial cable to the target. Then compile and link this example, download it to target, and run it. This example downloads through a serial cable from your computer to the target hardware SRAM, and it executes from the SRAM.

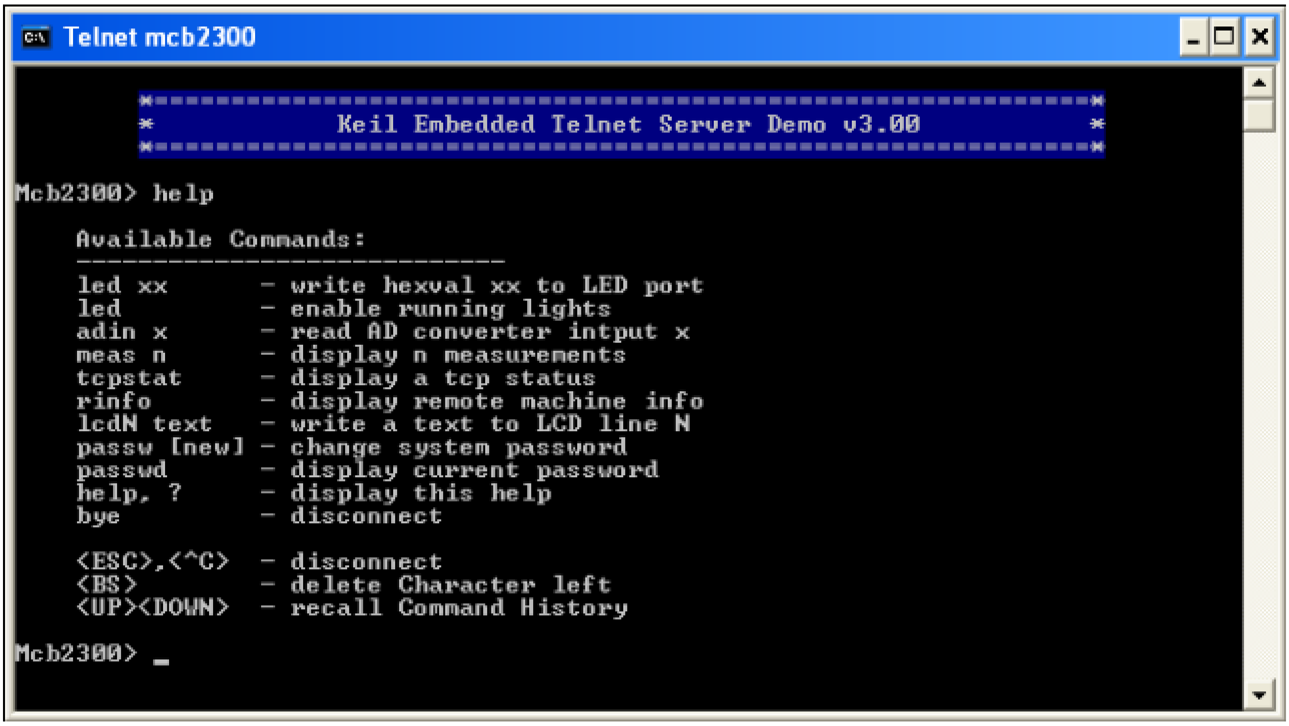
Testing the Telnet_demo

When you start the Telnet_demo, you can connect to a Telnet Server from your local computer. Run the **telnet** client program on your computer. Because **user authentication** is enabled by default, you must provide a username and a password when connecting. The default username is **admin**, and it has no password. Just press enter to connect.



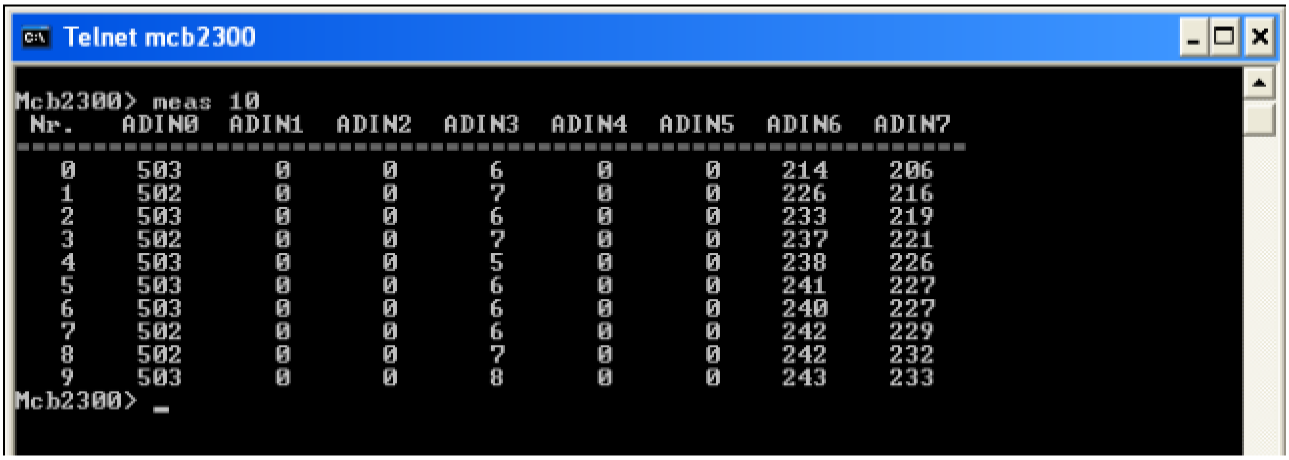
```
Keil Embedded Telnet Server V3.00, please login...

Username: admin
Password: _
```

You can check the integrated commands with the **help** command.



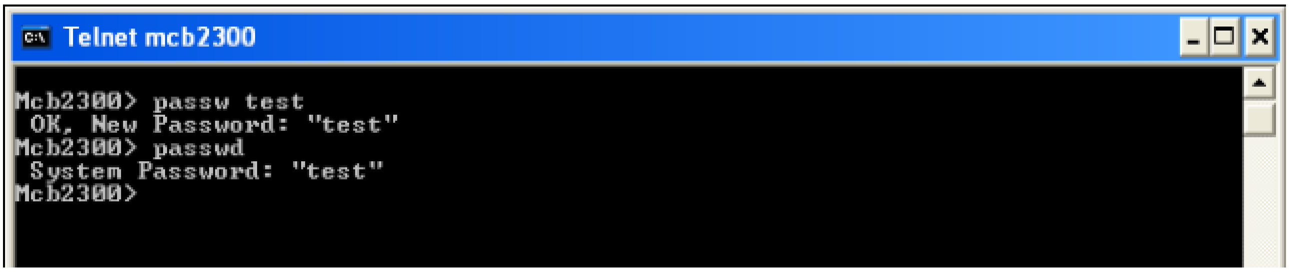
```
        *--------------------------------------------------------------*
        *              Keil Embedded Telnet Server Demo v3.00          *
        *--------------------------------------------------------------*
Mcb2300> help

    Available Commands:
    ----------------------------
    led xx        - write hexval xx to LED port
    led           - enable running lights
    adin x        - read AD converter intput x
    meas n        - display n measurements
    tcpstat       - display a tcp status
    rinfo         - display remote machine info
    lcdN text     - write a text to LCD line N
    passw [new]   - change system password
    passwd        - display current password
    help, ?       - display this help
    bye           - disconnect

    <ESC>,<^C>    - disconnect
    <BS>          - delete Character left
    <UP><DOWN>    - recall Command History

Mcb2300> _
```

Use the **meas** command to sample analog inputs and display the results. You can provide a parameter ***n*** to specify how many lines to display. The command example below, **meas 10**, shows you how **long lists** display in the Telnet window.



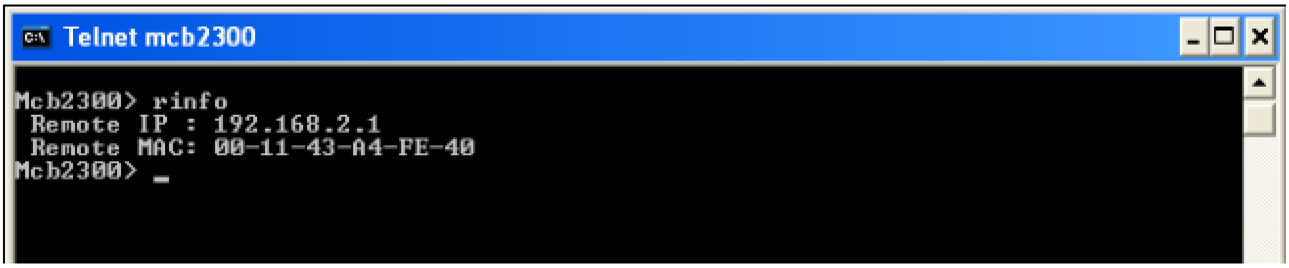
```
Mcb2300> meas 10
 Nr.    ADIN0   ADIN1   ADIN2   ADIN3   ADIN4   ADIN5   ADIN6   ADIN7
==============================================================================
   0     503       0       0       6       0       0     214     206
   1     502       0       0       7       0       0     226     216
   2     503       0       0       6       0       0     233     219
   3     502       0       0       7       0       0     237     221
   4     503       0       0       5       0       0     238     226
   5     503       0       0       6       0       0     241     227
   6     503       0       0       6       0       0     240     227
   7     502       0       0       6       0       0     242     229
   8     502       0       0       7       0       0     242     232
   9     503       0       0       8       0       0     243     233
Mcb2300> _
```

You can change a login password with the command **passw**. You can display the current password with the command **passwd**



```
Mcb2300> passw test
 OK, New Password: "test"
Mcb2300> passwd
 System Password: "test"
Mcb2300>
```

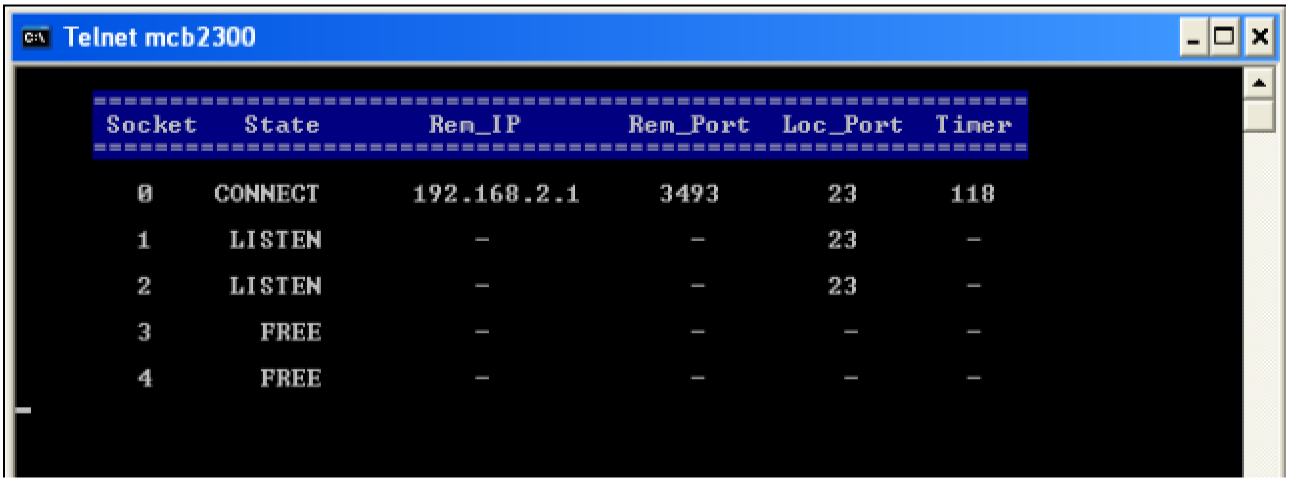
You can check the remote machine's IP and MAC address with the command **rinfo**. The remote machine is a Telnet Client computer that is connected to the Embedded Telnet Server.



Telnet mcb2300

```
Mcb2300> rinfo
 Remote IP : 192.168.2.1
 Remote MAC: 00-11-43-A4-FE-40
Mcb2300> _
```

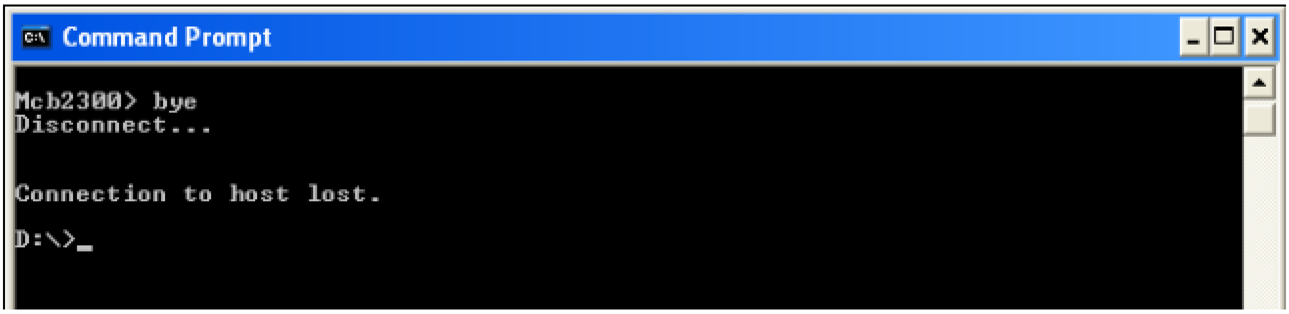
The Embedded Telnet Server also displays **continuously updating pages**. The command **tcpstat** monitors the status of the TCP sockets. This page is similar to the **Web Server example** page **Statistics**, where the monitoring is done by updating a web page.



Telnet mcb2300

```
==============================================================
 Socket   State        Rem_IP       Rem_Port  Loc_Port  Timer
==============================================================

   0     CONNECT     192.168.2.1     3493       23       118

   1      LISTEN          -            -        23        -

   2      LISTEN          -            -        23        -

   3        FREE          -            -         -        -

   4        FREE          -            -         -        -
_
```

After this example is tested, you can disconnect the Telnet connection with the **bye** command.



Command Prompt

```
Mcb2300> bye
Disconnect...

Connection to host lost.

D:\>_
```

TFTP_demo

The **TFTP_demo** example program shows you how to use the **TFTP Server** application to upload and download files to an embedded system. This example program is in the **\Keil\ARM\Boards\Phytec\LPC229x\RL\TCPnet** folder.

This example program is preconfigured for the following target hardware:

- **Phytec LPC229x** evaluation board
-

Network Configuration

This example is configured to run on the Local Area Network (LAN). In order to run it, the network parameters like IP address, network mask, default gateway, and DNS Server IP address must be specified. You can specify these parameters in the following way:

- Automatically when **DHCP Client** is **enabled** in the **Net_Config.c** configuration file. This is the default setting for this example. You must have a running DHCP Server in your local area network. Your DHCP Server provides an IP address, network mask, default gateway, and DNS Server IP address for the embedded system automatically when this example starts.
- Static **IP address**, **network mask**, **Default Gateway**, and **DNS server** **configured** in the **Net_Config.c** configuration file. You must specify a free IP address and copy the netmask from the LAN configuration.
  Be careful defining these parameters because this example does not work if the network configuration parameters are misconfigured. A DNS Server IP address is not really needed for this example and can be left at 0.0.0.0.

Now you must connect a network cable and a JTAG cable for ULINK®. Then compile and link this example, download it to the target, and run it. This example downloads through a JTAG interface over ULINK from your computer to the target hardware, and it executes from the external flash.

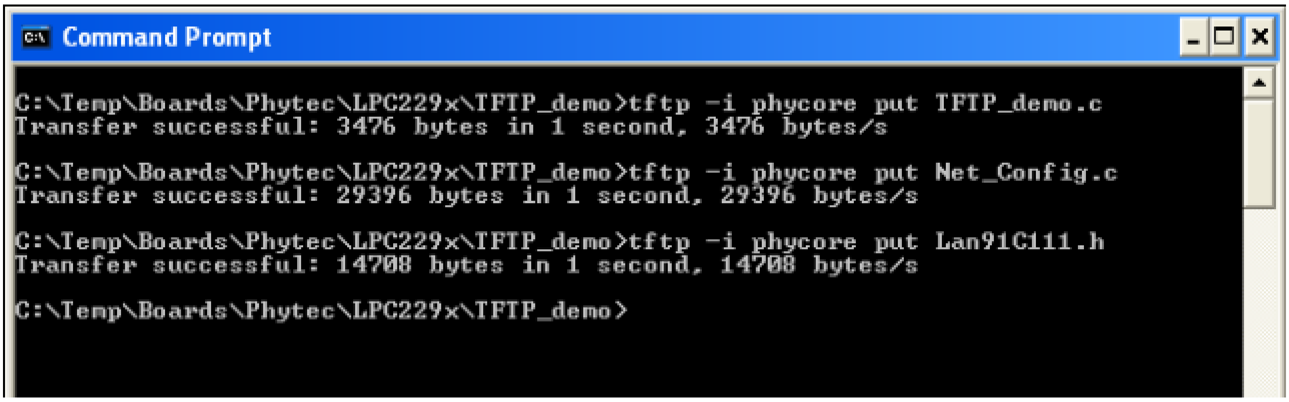
File System Configuration

This example uses a Flash File System to store files. This is configured in the **File_Config.c** configuration file. It is configured to use the **RAM File System** only. The files are stored to RAM. The File System capacity is configured for 1 Mbyte. This is also the maximum total size of files that can be stored to this RAM drive.

Testing the TFTP_demo

When you start the TFTP_demo example, you can transfer files to it. Run the **tftp** client program on your computer and send a file to the embedded system. You can use the name **phycore** for the target system. This name must be defined in the **system configuration**.

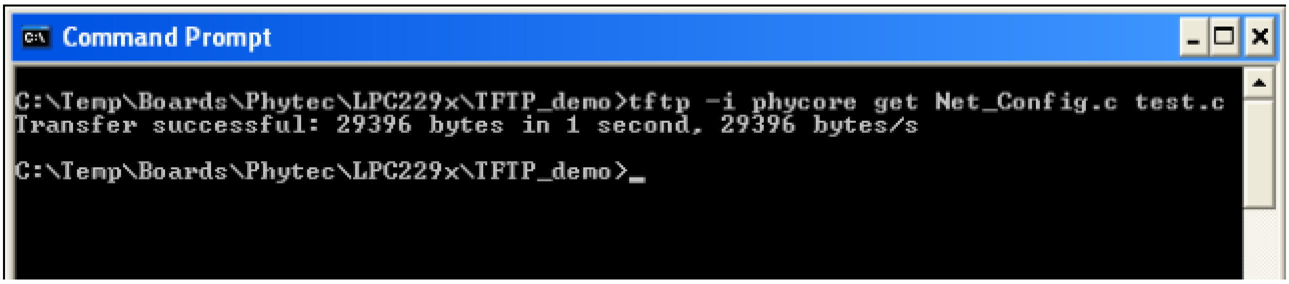
Use the **PUT** command to send files to the embedded TFTP Server as shown:



```
C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>tftp -i phycore put TFTP_demo.c
Transfer successful: 3476 bytes in 1 second, 3476 bytes/s

C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>tftp -i phycore put Net_Config.c
Transfer successful: 29396 bytes in 1 second, 29396 bytes/s

C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>tftp -i phycore put Lan91C111.h
Transfer successful: 14708 bytes in 1 second, 14708 bytes/s

C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>
```

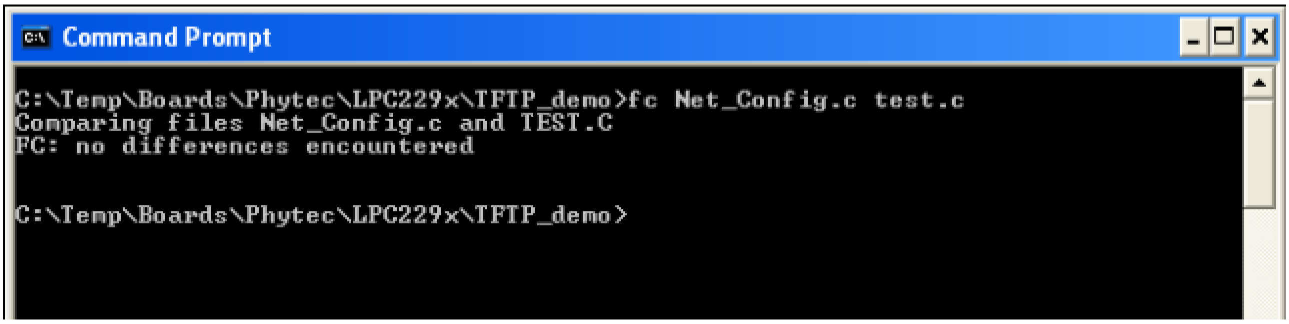
Now you can check the files with an integrated debug dialog. This is opened from the **Peripherals** menu when you select **RTX Kernel**. You need to **stop** the execution of the TFTP_demo application to allow µVision access to the target system and to allow it to retrieve information about the files for this dialog.

Read the file **Net_Config.c** stored on the embedded system and make a copy of it on the local computer under a new name **test.c**.



```
C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>tftp -i phycore get Net_Config.c test.c
Transfer successful: 29396 bytes in 1 second, 29396 bytes/s

C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>_
```

For a final check, compare the original file and the new file just received from the embedded system to see if they are equal.



```
C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>fc Net_Config.c test.c
Comparing files Net_Config.c and TEST.C
FC: no differences encountered


C:\Temp\Boards\Phytec\LPC229x\TFTP_demo>
```

Library Reference

The Real-Time Library provides more than 50 predefined functions and macros you may use in your ARM® real-time programs. The library makes embedded software development easier by providing routines that perform common real-time tasks.

Data Types

The Real-Time Library contains definitions for a number of types which may be used by the library routines. They are declared in include files which you may access from your C programs.

BIT

The **BIT** type is defined in rtl.h. It specifies the bit type used by the real-time kernel routines. The **BIT** type is defined as:

```
typedef unsigned char BIT;
```

and is used as shown in the following example:

```
#include <rtl.h>

BIT bit_val;
```

BOOL

The **BOOL** type is defined in rtl.h. It specifies the boolean type used by the real-time kernel routines. The **BOOL** type is defined as:

```
typedef unsigned int BOOL;
```

and is used as shown in the following example:

```

#include <rtl.h>


BOOL bval;
```

CAN_ERROR

The **CAN_ERROR** type is defined in can_error.h. It defines the error return values for the CAN driver routines. The **CAN_ERROR** type is defined as:

```
typedef enum {
  CAN_OK = 0,                    /* No error                             */
  CAN_NOT_IMPLEMENTED_ERROR,     /* Function has not been implemented    */
  CAN_MEM_POOL_INIT_ERROR,       /* Memory pool initialization error     */
  CAN_BAUDRATE_ERROR,            /* Baudrate was not set                 */
  CAN_TX_BUSY_ERROR,             /* Transmitting hardware busy           */
  CAN_OBJECTS_FULL_ERROR,        /* No more rx or tx objects available   */
  CAN_ALLOC_MEM_ERROR,           /* Unable to allocate memory from pool */
  CAN_DEALLOC_MEM_ERROR,         /* Unable to deallocate memory          */
  CAN_TIMEOUT_ERROR,             /* Timeout expired                      */
  CAN_UNEXIST_CTRL_ERROR,        /* Controller does not exist            */
  CAN_UNEXIST_CH_ERROR,          /* Channel does not exist               */
} CAN_ERROR;
```

and is used as shown in the following example:

```
#include <can_error.h>
   ..
  CAN_ERROR retval;
   ..
  retval = CAN_send (2, &msg_buf, 0x0F00);
   ..
```

CAN_msg

The **CAN_msg** type is defined in rtx_can.h. It specifies the CAN message structure used by the CAN driver routines. The **CAN_msg** type is defined as:

```
typedef struct {
  U32 id;         /* 11/29 bit message ID                  */
  U8  data[8];    /* Data field                            */
  U8  len;        /* Length of data field in bytes         */
  U8  ch;         /* Object channel                        */
  U8  format;     /* 0-STANDARD, 1-EXTENDED IDENTIFIER */
  U8  type;       /* 0-DATA FRAME, 1-REMOTE FRAME          */
} CAN_msg;
```

and is used as shown in the following example:

```
#include <rtx_can.h>

   ..
  CAN_msg msg_buf = {
    33,                            // ID
    { 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00 },    // Data
    1,                             // Length
    1,                             // Channel
    STANDARD_FORMAT,               // Format
    DATA_FRAME                     // Type
  };
   ..
  CAN_send (2, &msg_buf, 0x0F00);
   ..
```

FILE

The **FILE** type is defined in stdio.h. It specifies the structure used in all stream I/O operations. The fields of this structure store information about the current state of the stream. The **FILE** type is used as shown in the following example:

```
#include <stdio.h>

void capture_file (char mode) {
  FILE *f;
   ..
  f = fopen (filename,fmode);        /* open a file for writing          */
  if (f == NULL) {
    printf ("\nCan not open file!\n");/* error when trying to open file */
    return;
  }
                                     /* read line-edited serial input    */
   ..
  fclose (f);                        /* close the output file            */
  printf ("\nFile closed.\n");
}
```

S8

The **S8** type is defined in rtl.h. It specifies the signed 8-bit type used by the real-time kernel routines. The **S8** type is defined as:

```
typedef signed char S8;
```

and is used as shown in the following example:

```
#include <rtl.h>

S8 schar;
```

S16

The **S16** type is defined in rtl.h. It specifies the signed 16-bit type used by the real-time kernel routines. The **S16** type is defined as:

```
typedef short S16;
```

and is used as shown in the following example:

```
#include <rtl.h>

S16 sshort;
```

S32

The **S32** type is defined in rtl.h. It specifies the signed 32-bit type used by the real-time kernel routines. The **S32** type is defined as:

```
typedef int S32;
```

and is used as shown in the following example:

```
#include <rtl.h>

S32 sint;
```

S64

The **S64** type is defined in rtl.h. It specifies the signed 64-bit type used by the real-time kernel routines. The **S64** type is defined as:

```
typedef long long S64;
```

and is used as shown in the following example:

```
#include <rtl.h>

S64 sllong;
```

U8

The **U8** type is defined in rtl.h. It specifies the unsigned 8-bit type used by the real-time kernel routines. The **U8** type is defined as:

```
typedef unsigned char U8;
```

and is used as shown in the following example:

```
#include <rtl.h>

U8 uchar;
```

U16

The **U16** type is defined in rtl.h. It specifies the unsigned 16-bit type used by the real-time kernel routines. The **U16** type is defined as:

```
typedef unsigned short U16;
```

and is used as shown in the following example:

```
#include <rtl.h>

U16 ushort;
```

U32

The **U32** type is defined in rtl.h. It specifies the unsigned 32-bit type used by the real-time kernel routines. The **U32** type is defined as:

```
typedef unsigned int U32;
```

and is used as shown in the following example:

```
#include <rtl.h>

U32 uint;
```

U64

The **U64** type is defined in rtl.h. It specifies the unsigned 64-bit type used by the real-time kernel routines. The **U64** type is defined as:

```
typedef unsigned long long U64;
```

and is used as shown in the following example:

```
#include <rtl.h>

U64 ullong;
```

Include Files

The **\KEIL\ARM\RV31\INC** folder contains some of the include files provided with the Real-Time Library. Some include files are located in different folders. These files contain definitions for constants, macros, types, and function prototypes. The table below summarizes the include files, their use, and their location.

Filename	Description
can_cfg.h	This configures the hardware-level CAN driver using simple parameters, which the user can easily modify. You can edit this file using the Configuration Wizard in µVision. If you use a different CAN controller, you might have to modify this file. The file for the LPC2xxx device is located in: \Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1
can_error.h	This defines the error codes that can be returned by the RL-CAN routines. Location: \Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1
can_hw.h	This defines the prototypes for the hardware-level CAN driver routines. If you use a different CAN controller, you might have to modify these routines. The file for the LPC2xxx device is located in: \Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1
can_reg.h	This include file defines the register interface for the hardware-level CAN driver. If you use a different CAN controller, you might have to modify this file. The file for the LPC2xxx device is located in: \Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1
file_config.h	This defines constants and structures used by the Flash File System. It also defines the prototypes of the flash programming routines. Location: \Keil\ARM\RV31\INC
fs_flashdev.h	This contains the flash sector layout description. If you use a different controller, you might have to modify this file. The file for the LPC2xxx device using 256 KB flash is located in: \keil\ARM\RL\FlashFS\Flash\LPC_IAP_256
lpc23_emac.h	This defines hardware driver level constants and structures for the Philips LPC2378 EMAC ethernet controller. The source also contains header files for a few other ethernet controllers. If you use a different ethernet controller, you must provide your own include file. Location: \Keil\ARM\RL\TCPnet\User
net_config.h	This defines constants, structures and prototypes for the TCPnet routines that provide modem, ethernet, TFTP, and HTTP functionality. Location: \Keil\ARM\RV31\INC
rtl.h	This defines constants, structures and prototype for all the RTX kernel routines, TCPnet routines (TCP, UDP, PPP, and SLIP), and Flash File System routines. Location: \Keil\ARM\RV31\INC
rtx_can.h	This defines constants and prototypes for all CAN driver routines that can be used with the RTX kernel. You must include this file in all source files that use CAN Driver routines. Location: \Keil\ARM\Boards\Keil\MCB2100\RL\CAN\CAN_Ex1
rtx_config.h	This defines various types and structures used by the RTX kernel to manage tasks, messages, and events. Location: \Keil\ARM\RV31\INC
stdio.h	This contains prototype definitions of the file manipulation functions that are used by the Flash File System. Location: \Keil\ARM\RV31\INC

can_cfg.h

The **can_cfg.h** include file contains manifest constants that define the following:

- **USE_CAN_CTRLx**
- **CAN_No_SendObjects**
- **CAN_No_ReceiveObjects**

file_config.h

The file_config.h include file contains prototypes for some of the RL-FlashFS routines:

- **fs_Init**
- **fs_EraseSector**
- **spi_init**
- **spi_send**
- **spi_hi_speed**
- **fs_ProgramPage**
- **fs_get_time**
- **fs_get_date**

net_config.h

The net_config.h include file contains prototypes for some of the RL-TCPnet routines:

- **cgi_process_var**
- **cgi_process_data**
- **cgi_func**
- **http_get_var**
- **http_get_lang**
- **http_get_info**
- **http_get_session**
- **tnet_cbfunc**
- **tnet_process_cmd**
- **tnet_ccmp**
- **tnet_set_delay**
- **tnet_get_info**
- **tftp_fopen**
- **tftp_fclose**
- **tftp_fread**
- **tftp_fwrite**
- **get_host_by_name**
- **smtp_cbfunc**
- **init_ethernet**
- **poll_ethernet**
- **send_frame**
- **int_enable_eth**
- **int_disable_eth**
- **init_serial**
- **com_getchar**
- **com_putchar**
- **com_tx_active**
- **init_modem**
- **modem_dial**
- **modem_hangup**
- **modem_listen**
- **modem_online**
- **modem_process**
- **modem_run**

The net_config.h include file contains definitions for:

- **DNS error codes**
- **DNS event codes**
- **DHCP messages**
- **SMTP callback events**
- **SMTP server reply codes**
- **SMTP states**
- **TFTP error codes**
- **TFTP opcodes**
- **TFTP states**
- **Telnet flags**
- **Telnet states**
- **Telnet ASCII key codes**
- **Telnet commands**
- **HTTP states**
- **TCP callback events**
- **TCP socket types**
- **TCP states**
- **TCP flags**
- **UDP states**
- **ARP info states**
- **SLIP states**
- **PPP states**

The net_config.h include file defines structures for:

- **ICMP_HEADER**

- **TCP_HEADER**
- **UDP_HEADER**
- **IP_HEADER**
- **ARP_HEADER**
- **ETH_HEADER**
- **PPP_HEADER**
- **OS_FRAME**
- **REMOTEM**

rtl.h

The rtl.h include file contains prototypes for all the RTX kernel routines, and some of the TCPnet and FlashFS routines.

Prototypes for RTX kernel routines:

- **os_sys_init**
- **os_sys_init_prio**
- **os_tsk_create_ex**
- **os_tsk_delete_self**
- **os_tsk_prio_self**
- **os_tsk_create_user**
- **os_tsk_create_user_ex**
- **os_tsk_self**
- **os_sys_init_user**
- **os_tsk_pass**
- **os_tsk_prio**
- **os_tsk_delete**
- **os_evt_wait_or**
- **os_evt_wait_and**
- **os_evt_set**
- **os_evt_clr**
- **isr_evt_set**
- **os_evt_get**
- **os_sem_init**
- **os_sem_send**
- **os_sem_wait**
- **isr_sem_send**
- **os_mbx_init**
- **os_mbx_send**
- **os_mbx_wait**
- **os_mbx_check**
- **isr_mbx_check**
- **isr_mbx_send**
- **isr_mbx_receive**
- **os_mbx_declare**
- **os_mut_init**
- **os_mut_release**
- **os_mut_wait**
- **os_dly_wait**
- **os_itv_set**
- **os_itv_wait**
- **os_tmr_create**
- **os_tmr_kill**
- **tsk_lock**
- **tsk_unlock**
- **_init_box**
- **_init_box8**
- **_alloc_box**
- **_calloc_box**
- **_free_box**
- **_declare_box**
- **_declare_box8**

Prototypes of RL-FlashFS routines:

- **finit**
- **fdelete**
- **frename**
- **ffind**
- **ffree**
- **fformat**
- **fanalyse**
- **fcheck**
- **fdefrag**

- **unlink**

Prototypes of RL-TCPnet routines:

- **init_TcpNet**
- **main_TcpNet**
- **timer_tick**
- **udp_get_socket**
- **udp_release_socket**
- **udp_open**
- **udp_close**
- **udp_get_buf**
- **udp_send**
- **tcp_get_socket**
- **tcp_release_socket**
- **tcp_listen**
- **tcp_connect**
- **tcp_get_buf**
- **tcp_max_dsize**
- **tcp_check_send**
- **tcp_get_state**
- **tcp_send**
- **tcp_close**
- **tcp_abort**
- **arp_cache_ip**
- **ppp_listen**
- **ppp_connect**
- **ppp_close**
- **ppp_is_up**
- **slip_listen**
- **slip_connect**
- **slip_close**
- **slip_is_up**
- **get_host_by_name**
- **smtp_connect**
- **dhcp_disable**

rtx_can.h

The **rtx_can.h** include file contains prototypes for all the RL-CAN driver routines.

- **CAN_init**
- **CAN_start**
- **CAN_send**
- **CAN_request**
- **CAN_set**
- **CAN_receive**
- **CAN_rx_object**
- **CAN_tx_object**

The **rtx_can.h** include file contains definitions of types and structures for:

- **CAN message type**
- **CAN message format**
- **Mailboxes for CAN transmit and receive messages**
- **CAN memory pool**
- **CAN message structure**

The **rtx_can.h** include file contains for the CAN driver errors that might be encountered.

Following is a list of the CAN driver errors.

- **CAN_OK**
  No errors were encountered.
- **CAN_NOT_IMPLEMENTED_ERROR**
  The function invoked is not implemented and does not do anything.
- **CAN_MEM_POOL_INIT_ERROR**
  The memory pool used for software message buffers did not initialize successfully.
- **CAN_BAUDRATE_ERROR**
  The baudrate was not properly setup.
- **CAN_TX_BUSY_ERROR**
  The transmit hardware is busy.
- **CAN_OBJECTS_FULL_ERROR**
  All transmit and receive message objects are in use.
- **CAN_ALLOC_MEM_ERROR**
  Memory could not be allocated from the memory pool.
- **CAN_DEALLOC_MEM_ERROR**
  Memory previously used by the CAN message was not properly deallocated.
- **CAN_TIMEOUT_ERROR**
  The function did not complete in the specified time.
- **CAN_UNEXIST_CTRL_ERROR**
  A function tried to use a CAN controller that does not exist.
- **CAN_UNEXIST_CH_ERROR**
  A function tried to use a CAN channel that does not exist.

The **rtx_can.h** include file includes the can_cfg.h include file.

rtx_config.h

The rtx_config.h include file contains prototypes for some of the RTX kernel routines:

- **tsk_lock**
- **tsk_unlock**
- **os_tmr_call**
- **_init_box**
- **_alloc_box**
- **_calloc_box**
- **_free_box**
- **_declare_box**
- **_declare_box8**
- **_init_box8**

The rtx_config.h include file contains other definitions of types and structures used by the RTX kernel.

Note

- This include file is used by the RTX kernel for ARM7™ and ARM9™ devices. RTX library for Cortex™-M devices has a different concept and does not need this include file.

stdio.h

The stdio.h include file contains prototypes for some of the RL-FlashFS routines:

- fclose
- feof
- ferror
- fflush
- fgetc
- fgets
- fopen
- fprintf
- fputc
- fputs
- fread
- fscanf
- fseek
- ftell
- fwrite
- rename
- rewind
- ungetc

Reference

The following pages describe the routines in the Real-Time Library. Routines are listed in alphabetical order and each is divided into several sections:

Summary	Briefly describes the routine's effect, lists include file(s) containing its declaration and prototype, illustrates the syntax, and describes any arguments.
Description	Provides a detailed description of the routine and how it is used.
Return Value	Describes the value returned by the routine.
See Also	Names related routines.
Example	Gives a function or program fragment demonstrating proper use of the function.

_alloc_box

Summary

```
#include <rtl.h>

void *_alloc_box (
    void* box_mem );     /* Start address of the memory pool */
```

Description

The **_alloc_box** function allocates a block of memory from the memory pool that begins at the address *box_mem*.

The **_alloc_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the memory pool using the **_init_box** function before performing any other operation on the memory pool.
- The **_alloc_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_alloc_box** function returns a pointer to the allocated block if a block was available. If there was no available block, it returns a NULL pointer.

See Also

_calloc_box, _free_box, _init_box

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_calloc_box

Summary

```
#include <rtl.h>

void *_calloc_box (
    void* box_mem );     /* Start address of the memory pool */
```

Description

The **_calloc_box** function allocates a block of memory from the memory pool that begins at the address *box_mem* and initializes the entire memory block to 0.

The **_calloc_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the memory pool using the **_init_box** function before performing any other operation on the memory pool.
- The **_calloc_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_calloc_box** function returns a pointer to the allocated block if a block was available. If there was no available block, it returns a NULL pointer.

See Also

_alloc_box, **_free_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_declare_box

Summary

```
#include <rtl.h>

#define _declare_box( \
    pool,      \    /* Name of the memory pool variable. */
    size,      \    /* Number of bytes in each block. */
    cnt )      \    /* Number of blocks in the memory pool. */
    U32 pool[((size+3)/4)*(cnt) + 3]
```

Description

The **_declare_box** macro declares an array of bytes that can be used as a memory pool for fixed block allocation.

The argument *pool* specifies the name of the memory pool variable, which can be used by the memory block allocation routines. The argument *size* specifies the size of the blocks, in bytes. The argument *cnt* specifies the number of blocks required in the memory pool.

The **_declare_box** macro is part of RL-RTX. The definition is in rtl.h.

- The macro rounds up the value of *size* to the next multiple of 4 to give the blocks a 4-byte alignment.
- The macro also declares an additional 12 bytes at the start of the memory pool to store internal pointers and size information about the memory pool.

Return Value

The **_declare_box** macro does not return any value.

See Also

_alloc_box, _calloc_box, _free_box, _init_box

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
_declare_box(mpool,20,32);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_declare_box8

Summary

```
#include <rtl.h>

#define _declare_box8( \
    pool,       \    /* Name of the memory pool variable. */
    size,       \    /* Number of bytes in each block. */
    cnt )       \    /* Number of blocks in the memory pool. */
    U64 pool[((size+7)/8)*(cnt) + 2]
```

Description

The **_declare_box8** macro declares an array of bytes that can be used as a memory pool for allocation of fixed blocks with 8-byte alignment.

The argument *pool* specifies the name of the memory pool variable that is used by the memory block allocation routines. The argument *size* specifies the size of the blocks, in bytes. The argument *cnt* specifies the number of blocks required in the memory pool.

The **_declare_box8** macro is part of RL-RTX. The definition is in rtl.h.

- The macro rounds up the value of *size* to the next multiple of 8 to give the blocks an 8-byte alignment.
- The macro also declares an additional 16 bytes at the start of the memory pool to store internal pointers and size information about the memory pool.

Return Value

The **_declare_box8** macro does not return any value.

See Also

_alloc_box, _calloc_box, _free_box, _init_box8

Example

```
#include <rtl.h>

/* Reserve a memory for 25 blocks of 30-bytes. */
_declare_box8(mpool,30,25);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box8 (mpool, sizeof (mpool), 30);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
  /* 'box' and 'cbox' are always 8-byte aligned. */
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_free_box

Summary

```
#include <rtl.h>

int _free_box (
    void* box_mem,     /* Start address of the memory pool */
    void* box );        /* Pointer to the block to free */
```

Description

The **_free_box** function returns a memory block, which was allocated using **_alloc_box** or **_calloc_box**, back to the memory pool where it was obtained from.

The *box* argument specifies the address of the memory block to be freed.

The *box_mem* argument specifies the start address of the memory pool where the block was obtained from.

The **_free_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- If you return the memory block to a memory pool that did not provide the memory block, serious memory errors might occur.
- The **_free_box** function is reentrant and thread-safe. You can call it from the main function and from an IRQ interrupt function with no restriction.

Return Value

The **_free_box** function returns 0 if the memory block was successfully returned to the memory pool. If there was an error while freeing the block, it returns 1.

See Also

_alloc_box, **_calloc_box**, **_init_box**

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
U32 mpool[32*5 + 3];

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_init_box

Summary

```
#include <rtl.h>

int _init_box  (
    void* box_mem,        /* Start address of the memory pool */
    U32   box_size,       /* Number of bytes in the memory pool */
    U32   blk_size );     /* Number of bytes in each block of the
pool */
```

Description

The **_init_box** function initializes a fixed block size memory pool. When the memory pool is initialized, the RTX kernel handles memory requests by allocating a block of memory from the memory pool.

The *box_mem* specifies the start address of the memory pool, and this address must be 4-byte aligned.

The *box_size* argument specifies the size of the memory pool, in bytes.

The *blk_size* argument specifies the size, in bytes, of the blocks in the memory pool. You can set the block size to any value from 1 to *box_size*-12. However, the *blk_size* is rounded up to the next multiple of 4 to maintain 4-byte address alignment of the blocks. For example if you initialize a memory pool for 10-byte blocks, the **_init_box** function actually initializes the memory pool for 12-byte blocks.

The **_init_box** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The first 12 bytes from the memory pool are reserved for storing pointers and size information that can be used by the functions that handle the memory pool. The *box_size* must therefore be more than 12 bytes long.
- If the start address is not 4-byte aligned, the memory pool handling functions might fail.

Return Value

The **_init_box** function returns 0 if the memory pool was initialized without any problem. If there was an initialization error, it returns 1.

See Also

_alloc_box, _calloc_box, _declare_box, _free_box

Example

```
#include <rtl.h>

/* Reserve a memory for 32 blocks of 20-bytes. */
_declare_box(mpool,20,32);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box (mpool, sizeof (mpool), 20);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

_init_box8

Summary

```
#include <rtl.h>

int _init_box8 (
    void* box_mem,        /* Start address of the memory pool */
    U32   box_size,       /* Number of bytes in the memory pool */
    U32   blk_size );     /* Number of bytes in each block of the 
pool */
```

Description

The **_init_box8** function initializes a fixed block size memory pool with 8-byte alignment. When the memory pool is initialized, the RTX kernel handles memory requests by allocating a block of memory from the memory pool.

The *box_mem* specifies the start address of the memory pool, and this address must be 8-byte aligned.

The *box_size* argument specifies the size of the memory pool, in bytes.

The *blk_size* argument specifies the size, in bytes, of the blocks in the memory pool. You can set the block size to any value from 1 to *box_size*-16. However, the *blk_size* is rounded up to the next multiple of 8, to maintain 8-byte alignment of the blocks. For example if you initialize a memory pool for 10-byte blocks, the **_init_box8** function actually initializes the memory pool for 16-byte blocks.

The **_init_box8** function is implemented as a macro and is part of RL-RTX. The definition is in rtl.h.

Note

- The first 16 bytes from the memory pool are reserved for storing pointers and size information that can be used by the functions that handle the memory pool. The *box_size* must therefore be more than 16 bytes long.
- If the start address is not 8-byte aligned, the memory pool handling functions might fail.

Return Value

The **_init_box8** function returns 0 if the memory pool was initialized without any problem. If there was an initialization error, it returns 1.

See Also

_alloc_box, _calloc_box, _declare_box8, _free_box

Example

```
#include <rtl.h>

/* Reserve a memory for 25 blocks of 30-bytes. */
_declare_box8(mpool,30,25);

void membox_test (void) {
  U8 *box;
  U8 *cbox;

  _init_box8 (mpool, sizeof (mpool), 30);
  box  = _alloc_box (mpool);
  /* This block is initialized to 0. */
  cbox = _calloc_box (mpool);
  /* 'box' and 'cbox' are always 8-byte aligned. */
   ..
  _free_box (mpool, box);
  _free_box (mpool, cbox);
}
```

ADC_IF_GetRequest

Summary

```
#include <adcuser.h>

bool ADC_IF_GetRequest  (
  void);
```

Description

The **ADC_IF_GetRequest** function sends the value of the requested audio setup parameter to the host. The supported parameters are:

- maximum volume.
- minimum volume.
- current volume.
- volume resolution.
- mute.

The **ADC_IF_GetRequest** function is part of the Function Driver layer of the RL-USB Software Stack. There is no requirement to modify this function.

Return Value

The **ADC_IF_GetRequest** returns **__TRUE** if the host request is valid and supported. Otherwise, it returns **__FALSE**.

See Also

ADC_IF_SetRequest, FIQ_Handler

Example

There is no requirement to invoke or modify the **ADC_IF_GetRequest** function.

ADC_IF_SetRequest

Summary

```
#include <adcuser.h>

bool ADC_IF_SetRequest  (
  void);
```

Description

The **ADC_IF_SetRequest** function accepts the new value for one of the audio setup parameters:

- current volume.
- mute.

The **ADC_IF_SetRequest** function is part of the USB Function Driver layer of the RL-USB Software Stack. There is no requirement to modify this function.

Return Value

The **ADC_IF_SetRequest** returns **__TRUE** if the host request is valid and supported. Otherwise, it returns **__FALSE**.

See Also

ADC_IF_GetRequest, FIQ_Handler

Example

There is no requirement to invoke or modify the **ADC_IF_SetRequest** function.

arp_cache_ip

Summary

```
#include <rtl.h>

BOOL arp_cache_ip (
    U8* ipadr,      /* Pointer to buffer containing the 4 octets of
the IP address. */
    U8  type );     /* Specifies whether the IP address is fixed or
temporary. */
```

Description

The **arp_cache_ip** function determines whether the ARP table has a MAC (ethernet) address entry for the requested IP Address. If an entry does not exist, the function forces the TcpNet system to resolve and cache the MAC address into the internal ARP table buffer.

The argument *ipadr* points to a buffer containing the four octets of the dotted decimal IP address to be resolved.

The argument *type* specifies whether the IP address is fixed or temporary. This consequently determines whether or not the TcpNet system automatically refreshes the IP address entry in the ARP cache.

Type	Description
ARP_TEMP_IP	The IP address is temporary, and thus TCPnet removes the IP address entry from the ARP cache after a timeout.
ARP_FIXED_IP	The IP address is fixed, and thus TCPnet's ARP module automatically refreshes the IP address entry after the timeout.

The **arp_cache_ip** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- Only the ethernet network interface needs to use the **arp_cache_ip** function. There is no ARP protocol for the PPP and SLIP network interfaces.
- The **arp_cache_ip** function is primarily useful before sending the first UDP packet. The function is not necessary before sending TCP packets because the TCP module can retransmit the packet if the remote machine did not receive the packet.

Return Value

The **arp_cache_ip** function returns __TRUE when both of the following conditions are satisfied:

- The requested IP address is resolved.
- The ARP table contains an entry for the IP address and its MAC address.

Otherwise, the function returns __FALSE.

Example

```
#include <rtl.h>

BOOL ip_cached;

void send_data (void) {
  static const U8 rem_IP[4] = {192,168,0,100};

  if (ip_cached == __FALSE) {
    if (arp_cache_ip (rem_IP, ARP_FIXED_IP) == __FALSE) {
      /* Wait, 'rem_IP' address not resolved yet. */
      return;
    }
    ip_cached == __TRUE;
     ..
    /* OK send UDP data packet here. */
     ..
  }
}

void main (void) {
  /* Main Thread of the TcpNet */

  init_TcpNet ();
  ip_cached = __FALSE;

  while (1) {
    timer_poll ();
    main_TcpNet ();
    send_data ();
  }
}
```

CAN_init

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_init (
    U32 ctrl,          /* CAN Controller */
    U32 baudrate);     /* Baudrate */
```

Description

The **CAN_init** function sets the *baudrate* for the CAN controller specified by *ctrl*. The *baudrate* may be a value from 10,000-1,000,000 (10 Kbps to 1 Mbps).

This function may be invoked one or more times.

- The first invocation of **CAN_init** configures hardware used by the CAN driver, initializes common resources including mailboxes used as message FIFOs, and sets the CAN bus *baudrate*.
- Subsequent invocations initialize mailboxes used as message FIFOs and set the CAN bus *baudrate*.

The **CAN_init** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Return Value

The **CAN_init** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_MEM_POOL_INIT_ERROR**
  Indicates the memory pool was incorrectly initialized.
- **CAN_BAUDRATE_ERROR**
  Indicates that the communication speed was incorrectly initialized.

See Also

CAN_start

Example

```
#include <rtx_can.h>

void main (void) {
   ..
  CAN_init (1, 250000);     /* Init CAN Controller, 250Kbps */
   ..
}
```

CAN_receive

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_receive (
    U32 ctrl,          /* CAN Controller */
    CAN_msg *msg,      /* CAN Message */
    U16 timeout);     /* Time to Wait */
```

Description

The **CAN_receive** function receives a message on the CAN controller specified by *ctrl* and copies it into *msg*.

The **CAN_receive** function does not clear hardware message FIFOs. So, if a message was received prior to invoking **CAN_receive**, that message is returned immediately. If the message FIFO is empty, **CAN_receive** waits (up to the specified *timeout*) for a message to be received.

timeout	Description
0	Return immediately.
0x0001-0xFFFE	Wait the specified number of RTX Kernel ticks.
0xFFFF	Wait infinitely.

If a message is not received by the specified time, an error is returned.

The **CAN_receive** function executes quickly since all data transfers use software buffers. Only in situations where the FIFO is empty is the **CAN_receive** function delayed.

The **CAN_receive** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Return Value

The **CAN_receive** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_DEALLOC_MEM_ERROR**
  Indicates that the memory used by the received message was not correctly deallocated.
- **CAN_TIMEOUT**
  Indicates that the timeout expired before a message was received.

See Also

CAN_send

Example

```
#include <rtx_can.h>

__task void task_rece_CAN (void) {
  CAN_msg msg_buf;

  for (;;) {
    // Wait to receive a message.
    // When the message arrives
    // activate LEDs using data[0]

    if (CAN_receive (1, &msg_buf, 0) == 0) {
      LED_Byte (msg_buf.data[0]);
    }
  }
}
```

CAN_request

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_request (
    U32 ctrl,          /* CAN Controller */
    CAN_msg *msg,      /* CAN Message to Request */
    U16 timeout);     /* Time to Wait */
```

Description

The **CAN_request** function sends a REMOTE FRAME request (a special CAN message that requests transmission of a specific *msg*) via the CAN controller hardware specified by *ctrl*.

If the CAN controller hardware is ready (no other transmissions are in progress), the **CAN_request** function sends the REMOTE FRAME request immediately. If the CAN controller is busy, the request *msg* is put into a FIFO (that is managed using an RTX mailbox). Messages stored in the the FIFO are sent in order.

The *timeout* specifies how long to wait for the FIFO (mailbox slot) to become available.

timeout	Description
0	Return immediately.
0x0001-0xFFFE	Wait the specified number of RTX Kernel ticks.
0xFFFF	Wait infinitely.

If a request is not stored in the FIFO by the specified time, an error is returned.

The **CAN_request** function executes quickly since all data transfers use software buffers. Only in situations where the FIFO is full is the **CAN_request** function delayed.

The **CAN_request** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Return Value

The **CAN_request** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_ALLOC_MEM_ERROR**
  Indicates there is no available memory in the CAN memory pool.
- **CAN_DEALLOC_MEM_ERROR**
  Indicates that the memory used by the received message was not correctly deallocated.
- **CAN_TIMEOUT**
  Indicates that the timeout expired before a message was received.

See Also

CAN_set

Example

```
#include <rtx_can.h>

__task void task_send_CAN (void) {
  CAN_msg msg_buf = {
    33,                              // ID
    { 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00 },    // Data
    1,                               // Length
    1,                               // Channel
    STANDARD_FORMAT,                 // Format
    REMOTE_FRAME                  // Type
  };

  while (1) {
    // Request DATA FRAME message ID 33
    // on CAN Controller 1
    CAN_request (1, &msg_buf, 0x0F00);

    // Wait 100ms
    os_dly_wait (10);
  }
}
```

CAN_rx_object

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_rx_object (
    U32 ctrl,                /* CAN Controller */
    U32 channel,            /* CAN Channel Number */
    U32 id,                  /* Message ID */
    U32 object_para);       /* Object Parameters */
```

Description

The **CAN_rx_object** function enables message reception for the CAN controller specified by `ctrl` on the specified `channel`. Once enabled, the CAN controller will receive messages matching the specified `id`. The `object_para` may be one of the following:

- **DATA_TYPE**
  DATA FRAME message type.

- **REMOTE_TYPE**
  REMOTE FRAME message type.

- **STANDARD_TYPE**
  Message with standard 11-bit identifier type.

- **EXTENDED_TYPE**
  Message with extended 29-bit identifier type.

Note

Types DATA_TYPE or REMOTE_TYPE can be used together with STANDARD_TYPE or EXTENDED_TYPE (type specifiers can be bit-ored together).

The **CAN_rx_object** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Note

- Some CAN controllers, like that used in the NXP LPC2000 devices, do not use the `channel` information. For these devices, you should specify a value of 0 for the `channel`.

Return Value

The **CAN_rx_object** function returns one of the following manifest constants.

- **CAN_OK**
  Success.

- **CAN_OBJECTS_FULL_ERROR**
  Indicates that no more transmit or receive objects may be defined.

See Also

CAN_tx_object

Example

```
#include <rtx_can.h>

void main (void) {
  CAN_rx_object (1, 0, 33, DATA_TYPE | STANDARD_TYPE);

  // Enable reception of a DATA message on controller 1,
  // channel 0, with STANDARD 11-bit ID 33
}
```

CAN_send

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_send (
    U32 ctrl,          /* CAN Controller */
    CAN_msg *msg,      /* CAN Message */
    U16 timeout);     /* Time to Wait */
```

Description

The **CAN_send** function transmits *msg* using the CAN controller specified by *ctrl*.

If the CAN controller hardware is ready (no other transmissions are in progress), the **CAN_send** function sends the *msg* to the CAN controller for transmission. If the CAN controller is busy, the *msg* is put into a FIFO (that is managed using an RTX mailbox). Messages stored in the the FIFO are sent in order.

The *timeout* specifies how long to wait for the FIFO (mailbox slot) to become available.

timeout	Description
0	Return immediately.
0x0001-0xFFFE	Wait the specified number of RTX Kernel ticks.
0xFFFF	Wait infinitely.

If a message is not stored in the FIFO by the specified time, an error is returned.

The **CAN_send** function executes quickly since all data transfers use software buffers. Only in situations where the FIFO is full is the **CAN_send** function delayed.

The **CAN_send** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Return Value

The **CAN_send** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_ALLOC_MEM_ERROR**
  Indicates there is no available memory in the CAN memory pool.
- **CAN_DEALLOC_MEM_ERROR**
  Indicates that the memory used by the transmitted message was not correctly deallocated.
- **CAN_TIMEOUT**
  Indicates that the timeout expired before a message was transmitted.

See Also

CAN_receive

Example

```
#include <rtx_can.h>

__task void task_send_CAN (void) {
  unsined int i = 0;

  CAN_msg msg_buf = {
    33,                              // ID
    { 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00 },    // Data
    1,                               // Length
    1,                               // Channel
    STANDARD_FORMAT,                 // Format
    DATA_FRAME                       // Type
  };

  while (1) {
    // Put data (i) into the transmit buffer
    // Send CAN message on controller 2
    msg_buf.data[0] = ++i;
    CAN_send (2, &msg_buf, 0x0F00);

    // Wait 100ms
    os_dly_wait (10);
  }
}
```

CAN_set

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_set (
    U32 ctrl,          /* CAN Controller */
    CAN_msg *msg,      /* CAN Message */
    U16 timeout);     /* Time to Wait */
```

Description

The **CAN_set** function sets the DATA FRAME message to be sent automatically by the CAN controller hardware (specified by *ctrl*) in response to a REMOTE FRAME request.

If the CAN controller hardware supports REMOTE FRAME requests, this function sets the DATA FRAME message from *msg*. If the hardware is busy it will retry every timer tick.

The *timeout* specifies how long to wait for the hardware to become available.

timeout	Description
0	Return immediately.
0x0001-0xFFFE	Wait the specified number of RTX Kernel ticks.
0xFFFF	Wait infinitely.

If the message is not set by the specified time, an error is returned.

The **CAN_set** function executes quickly because the CAN controller hardware is rarely busy for long periods of time.

The **CAN_set** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Return Value

The **CAN_set** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_TIMEOUT**
  Indicates that the timeout expired before a message was set.
- **CAN_NOT_IMPLEMENTED_ERROR**
  Indicates that hardware does not offer this functionality.

See Also

CAN_request

Example

```
#include <rtx_can.h>

__task void task_send_CAN (void) {
  unsined int i = 0;

  CAN_msg msg_buf = {
    33,                              // ID
    { 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00 },    // Data
    1,                               // Length
    1,                               // Channel
    STANDARD_FORMAT,                 // Format
    DATA_FRAME                       // Type
  };

  while (1) {
    // Put data (i) into the transmit buffer
    // Set DATA FRAME message on controller 1
    msg_buf.data[0] = ++i;
    CAN_set (1, &msg_buf, 0x0F00);

    // Wait 100ms
    os_dly_wait (10);
  }
}
```

CAN_start

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_start (
    U32 ctrl);    /* CAN Controller to Enable */
```

Description

The **CAN_start** function starts the CAN controller specified by *ctrl* and enables the CAN controller to participate on the CAN network.

The **CAN_start** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Note

- The CAN controller cannot send or receive messages on the network until the **CAN_start** function is invoked.

Return Value

The **CAN_start** function returns one of the following manifest constants.

- **CAN_OK**
  Success.

See Also

CAN_init

Example

```
#include <rtx_can.h>

void main (void) {
   ..
  CAN_start (1);    /* Start CAN Controller 1 */
   ..
}
```

CAN_tx_object

Summary

```
#include <rtx_can.h>

CAN_ERROR CAN_tx_object (
    U32 ctrl,              /* CAN Controller */
    U32 channel,          /* CAN Channel Number */
    U32 id,                /* Message ID */
    U32 object_para);     /* Object Parameters */
```

Description

The **CAN_tx_object** function enables message transmission for the CAN controller specified by *ctrl* on the specified *channel*. Once enabled, the CAN controller can transmit messages matching the specified *id*. The *object_para* may be one of the following:

- **DATA_TYPE**
  DATA FRAME message type.
- **REMOTE_TYPE**
  REMOTE FRAME message type.
- **STANDARD_TYPE**
  Message with standard 11-bit identifier type.
- **EXTENDED_TYPE**
  Message with extended 29-bit identifier type.

Note

Types DATA_TYPE or REMOTE_TYPE can be used together with STANDARD_TYPE or EXTENDED_TYPE (type specifiers can be bit-ored together).

The **CAN_tx_object** function is part of RL-CAN. The prototype is defined in RTX_CAN.h.

Note

- Enabling a transmission object is not necessary for many microcontrollers (like the NXP LPC2000 and ST Microelectronics STR7 devices) because they can send CAN messages without configuring a transmission object. As such, this function is not implemented for some devices.
- For devices like the ST Microelectronics STR7, you must leave at least one message object available for message transmission. You cannot initialize the object as a receive object and use it for transmission.
- Some CAN controllers, like that used in the NXP LPC2000 devices, do not use the *channel* information. For these devices, you should specify a value of 0 for the *channel*.

Return Value

The **CAN_tx_object** function returns one of the following manifest constants.

- **CAN_OK**
  Success.
- **CAN_OBJECTS_FULL_ERROR**
  Indicates that no more transmit or receive objects may be defined.

See Also

CAN_rx_object

Example

```
#include <rtx_can.h>

void main (void) {
  CAN_tx_object (1, 2, 33, DATA_TYPE | STANDARD_TYPE);
  // Enable DATA message transmission on controller 1,
  // channel 2, with STANDARD 11-bit ID 33
}
```

cgi_func

Summary

```
#include <net_config.h>

U16 cgi_func (
    U8*   env,        /* Pointer to input string from a TCPnet
script. */
    U8*   buf,        /* Location where to write the HTTP response
string. */
    U16   buflen,     /* Number of bytes in the output buffer. */
    U32* pcgi );      /* Pointer to a storage variable. */
```

Description

The **cgi_func** function is what the TCPnet script interpreter calls, when interpreting the TCPnet script, to output the dynamic part of the HTTP response. The script interpreter calls **cgi_func** for each line in the script that begins with the command *c* . You must customize the **cgi_func** function so that it can understand and use the input string from the TCPnet script.

The argument *env* is a pointer to the input string that **cgi_func** uses to create the dynamic response. It is the same string of bytes that is specified in the TCPnet script code using the *c* command.

The argument *buf* is a pointer to the output buffer where the **cgi_func** function must write the HTTP response.

The argument *buflen* specifies the length of the output buffer in bytes.

The argument *pcgi* is a pointer to a variable that never gets altered by the HTTP Server. Hence, you can use it to store parameters for successive calls of the **cgi_func** function. You might use this to store:

- loop counters
- number of sent bytes
- pointer to a local status buffer.

The **cgi_func** function is in the HTTP_CGI.c module. The prototype is defined in net_config.h.

note

- The contents written by the **cgi_func** function, into the output buffer, must be HTML code.
- *c* is a command that is available in the TCPnet scripting language.
- The length of the output buffer, *buflen*, might vary because buffer length is determined by the TCP socket Maximum Segment Size (MSS) negotiation. The buffer length is normally around 1400 bytes for local LAN. But this can be reduced to 500 bytes or even less.
- The length of the output buffer, *buflen* also varies because the HTTP Server tries to optimize number of generated TCP packets. It calls this function again to use the complete buffer available. It stops when there is only 240 or less bytes freee in the buffer. Then the packet is generated and transmitted. If you want to force the HTTP Server to transmit the packet, return value from this function shall be or-ed with 0x4000.
- If the **cgi_func** function writes more bytes than *buflen* into the output buffer, then a system crash resulting from corruption of memory link pointers is highly likely.
- The input string *env* might contain single-character subcommands to tell

the **cgi_func** function how to process the script line correctly. There is no rule for these subcommands, so you can create and use your own commands.

- The argument *pcgi* is private to each HTTP Session. The HTTP Server clears the data in the *pcgi* pointer, to 0, before the **cgi_func** function is called for the first time in each session.
- The **cgi_func** function must update the contents in *pcgi* for each call. You can use the 4 bytes in *pcgi* to store data in any format.

Return Value

The **cgi_func** function returns the number of bytes written to the output buffer and writes the repeat flag value in the most significant bit of the return value.

If the return value's most significant bit is set to 1 (return value or-ed with 0x8000), the TCPnet script interpreter calls the **cgi_func** function again with the same values for the arguments *env*, *buflen*, and *pcgi*, which holds the same content as previously set. The argument *buf* is adjusted according to the number of bytes that were written to the output buffer.

If the return value's second most significant bit is set to 1 (return value or-ed with 0x4000), the packet optimization is canceled and the current packet is transmitted immediatelly.

See Also

cgi_process_data, cgi_process_var, http_get_lang

Example

```
U16 cgi_func (U8 *env, U8 *buf, U16 buflen, U32 *pcgi) {
  U16 len = 0;

  switch (env[0]) {
    /* Analyze the environment string. It is the script 'c' line
starting */
    /* at position 2. What you write to the script file is
returned here. */
    case 'a' :
      /* Network parameters - file 'network.cgi' */
       ..
      break;

    case 'b':
      /* LED control - file 'led.cgi' */
       ..
      break;

    case 'c':
      /* TCP status - file 'tcp.cgi' */
       ..
      break;

    case 'd':
      /* System password - file 'system.cgi' */
      switch (env[2]) {
        case '1':
          len = sprintf(buf,&env[4],http_EnAuth ? "Enabled" :
"Disabled");
          break;
        case '2':
          len = sprintf(buf,&env[4],http_auth_passw);
          break;
      }
      break;
  }
  return (len);
}
```

cgi_process_data

Summary

```
#include <net_config.h>

void cgi_process_data (
    U8   code,      /* Type of data in received data buffer. */
    U8*  dat,       /* Pointer to the data string from the POST
method. */
    U16  len );     /* Number of bytes in the data string. */
```

Description

The **cgi_process_data** function processes the data returned from the CGI form POST method. The HTTP server calls this function when a user presses the SUBMIT button on the input form, using a web browser.

Code	Data Type	Meaning of *dat* pointer	Meaning of *len*
0	Form data	Pointer to data string returned from the POST method.	Length of data string.
1	Filename	Pointer to a Filename for the http file upload. Filename is a 0-terminated string.	Length of a filename.
2	File data	Pointer to data packet received from the host.	Length of data packet.
3	End of file	*NULL*	Don't care.
4	XML data	Pointer to data string returned from the XML-POST method. A single packet or last packet in xml data stream.	Length of data string.
5	XML data	Pointer to data string returned from the XML-POST method. The same as under 4, but with more xml data to follow.	Length of data string.

The **cgi_process_data** function is in the HTTP_CGI.c module. The prototype is defined in net_config.h.

note

- The HTTP server calls the **cgi_process_data** function only if the input form, in HTML source, is created with the attribute **METHOD=POST**. For example:

```

<FORM ACTION=index.htm METHOD=POST NAME=CGI>
  ..
</FORM>
```

- Web browsers provide a filename for HTTP file upload with **path** included. It is a user responsibility to remove path information from the filename.
- For large files, file data is received in several small packets. The size of data packet depends on the TCP Maximum Segment Size. MSS value is typically 1460 bytes.
- The **XML-POST** is generated from the web service application, for example the **Silverlight**.

Return Value

The **cgi_process_data** function does not return any value.

See Also

cgi_func, cgi_process_var, http_get_lang

Example

```
void cgi_process_data (U8 code, U8 *dat, U16 len) {
  U8 passw[12], retyped[12];
  U8 var[40], stpassw;
  switch (code) {
    case 0:
      /* Url encoded form data received. */
      break;

    case 1:
      /* Filename for file upload received as encoded by the
browser. */
      /* It might contain an absolute path to a file from the
sending */
      /* host. Open a file for writing. */
      return;

    case 2:
      /* File content data received. Write data to a file. */
      /* This function will be called several times with   */
      /* code 2 when a big file is being uploaded.         */
      return;

    case 3:
      /* File upload finished. Close a file. */
      return;

    case 4:
      /* XML encoded content type, last packet. */
      pType = http_get_content_type ();
      /* check the content type for CGX file request. */
      /* pType is a pointer to a 0-terminated string  */
      /* For example: text/xml; charset=utf-8         */
      return;

    case 5:
      /* XML encoded as under 4, but with more to follow. */
      return;

    default:
      /* Ignore all other codes. */
      return;
  }

  if (len == 0) {
    /* No data, or all items (radio, checkbox) are off. */
    return;
  }
  stpassw = 0;
  do {
    /* Parse all returned parameters. */
    dat = http_get_var (dat, var, 40);
    if (var[0] != 0) {
      /* Parameter found, returned string is non 0-length. */
      if (str_scomp (var, "pw=") == __TRUE) {
        /* Change password. */
        str_copy (passw, var+3);
        stpassw |= 1;
      }
      else if (str_scomp (var, "pw2=") == __TRUE) {
        /* Retyped password. */
        str_copy (retyped, var+4);
        stpassw |= 2;
      }
    }
  } while (dat);
  if (stpassw == 0x03) {
    len = strlen (passw);
    if (mem_comp (passw, retyped, len) == __TRUE) {
      /* OK, both entered passwords the same, change it. */
      str_copy (http_auth_passw, passw);
    }
  }
}
```

cgi_process_var

Summary

```
#include <net_config.h>

void cgi_process_var (
    U8* qs );     /* Pointer to QUERY_STRING returned from the GET
method. */
```

Description

The **cgi_process_var** function processes the environmental variable QUERY_STRING that is returned from the CGI form GET method. The HTTP server calls this function when a user presses the SUBMIT button on the input form, using a web browser.

The argument *qs* points to the QUERY_STRING that is returned from the GET method.

The **cgi_process_var** function is in the HTTP_CGI.c module. The prototype is defined in net_config.h.

note

- The querry string *qs* is terminated by the space character.
- The HTTP server calls the **cgi_process_var** function only if the input form, in HTML source, is created with attribute **METHOD=GET**. For example:

```

<FORM ACTION=index.htm METHOD=GET NAME=CGI>
  ..
</FORM>
```

Return Value

The **cgi_process_var** function does not return any value.

See Also

cgi_func, cgi_process_data

Example

```
void cgi_process_var (U8 *qs) {
  U8 var[40];

  do {
    /* Loop through all the parameters. */
    qs = http_get_var (qs, var, 40);
    /* Check the returned string, 'qs' now points to the next. */
    if (var[0] != 0) {
      /* Returned string is non 0-length. */
      if (str_scomp (var, "ip=") == __TRUE) {
        /* My IP address parameter. */
        sscanf (&var[3],
"%bd.%bd.%bd.%bd",&LocM.IpAdr[0],&LocM.IpAdr[1],

&LocM.IpAdr[2],&LocM.IpAdr[3]);
      }
      else if (str_Scomp (var, "msk=") == __TRUE) {
        /* Net mask parameter. */
        sscanf (&var[4],
"%bd.%bd.%bd.%bd",&LocM.NetMask[0],&LocM.NetMask[1],

&LocM.NetMask[2],&LocM.NetMask[3]);
      }
      else if (str_scomp (var, "gw=") == __TRUE) {
       ..
      }
    }
  }while (qs);
}
```

cgx_content_type

Summary

```
#include <net_config.h>

U8 *cgx_content_type (void);
```

Description

The **cgx_content_type** function allows you to change the *Content-Type* html header in the response to the **Silverlight** web service application requests. The function returns a pointer to the new *Content-Type* html header. You can use this function to override the default content type header from the TCPnet library. This content type header is used in **cgx** script responses.

The default content type header in **cgx** script response is:

```
Content-Type: text/xml\r\n
```

This header is used if the **cgx_content_type** function does not exist in the project or if it returns a **NULL** pointer.

The **cgx_content_type** function is in the HTTP_CGI.c module. The prototype is defined in net_config.h.

note

- This function is **optional**. If it does not exist in the project, the default library function is used instead.

Return Value

The **cgx_content_type** function returns a pointer to 0-terminated *Content-Type* string.

See Also

http_get_content_type

Example

```
U8 *cgx_content_type (void) {
  /* A 0-terminated string must contain also a termination cr-lf.
*/
  return ("Text/xml; charset=utf-8\r\n");
}
```

com_getchar

Summary

```
#include <net_config.h>

int com_getchar (void);
```

Description

The **com_getchar** function reads a character from the serial input buffer.

The **com_getchar** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- The serial driver functions must not use waiting loops because loops block the TCPnet system.
- You must provide the **com_getchar** function if the serial controller you use is different from the ones provided in the TCPnet source.

Return Value

The **com_getchar** function returns the character value it read. It returns -1 if the input buffer is empty.

See Also

com_putchar, com_tx_active, init_serial

Example

```
int com_getchar (void) {
  /* Read a byte from serial interface */
  struct buf_st *p = &rbuf;

  if (p->in == p->out) {
    /* Serial receive buffer is empty. */
    return (-1);
  }
  return (p->buf[p->out++]);
}
```

com_putchar

Summary

```
#include <net_config.h>

BOOL com_putchar (
    U8 c );    /* The character to write to the output buffer. */
```

Description

The **com_putchar** function writes the character specified by the argument *c* to the serial output buffer and activates the serial transmission if it is not already active.

The **com_putchar** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- The serial driver functions must not use waiting loops because loops block the TCPnet system.
- You must provide the **com_putchar** function if the serial controller you use is different from the ones provided in the TCPnet source.

Return Value

The **com_putchar** function returns __TRUE if it successfully wrote the character into the output buffer. Otherwise, for example if the buffer is full, it returns __FALSE.

See Also

com_getchar, **com_tx_active**, **init_serial**

Example

```
BOOL com_putchar (U8 c) {
  struct buf_st *p = &tbuf;

  /* Write a byte to serial interface */
  if ((U8)(p->in + 1) == p->out) {
    /* Serial transmit buffer is full. */
    return (__FALSE);
  }
  VICIntEnClr = (1 << 7);
  if (tx_active == __FALSE) {
    /* Send directly to UART. */
    U1THR = (U8)c;
    tx_active = __TRUE;
  }
  else {
    /* Add data to transmit buffer. */
    p->buf [p->in++] = c;
  }
  VICIntEnable = (1 << 7);
  return (__TRUE);
}
```

com_tx_active

Summary

```
#include <net_config.h>

BOOL com_tx_active (void);
```

Description

The **com_tx_active** function checks to see if the serial transmission is still active. The serial transmission is active until it has finished sending the last byte from the transmit buffer.

The **com_tx_active** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- When TCPnet has decided to shutdown the serial data link, it monitors the serial transmission using the **com_tx_active** function. Only when the output buffer is empty, TCPnet hangs up the modem and disconnects the phone line.
- The serial driver functions must not use waiting loops because loops block the TCPnet system.
- You must provide the **com_tx_active** function if the serial controller you use is different from the ones provided in the TCPnet source.

Return Value

The **com_tx_active** function returns __TRUE if the serial transmission is still active. Otherwise, it returns __FALSE.

See Also

com_getchar, com_putchar, init_serial

Example

```
BOOL com_tx_active (void) {
  /* Return status Transmitter active/not active.         */
  /* When transmit buffer is empty, 'tx_active' is FALSE. */
  return (tx_active);
}
```

dhcp_disable

Summary

```
#include <rtl.h>

void dhcp_disable (void)
```

Description

The **dhcp_disable** function permanently disables the Dynamic Host Configuration at runtime. To enable the DHCP client again, you must restart the system.

The **dhcp_disable** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Note

- You can use the function only when the DHCP Client is already enabled in the net_config.c configuration file.

Return Value

The **dhcp_disable** function does not return any value.

Example

```
#include <rtl.h>

void main (void) {
  /* Main Thread of the TcpNet */

  init_TcpNet ();

  if (DHCP_mode == 0) {
    dhcp_disable ();
  }

  while (1) {
    timer_poll ();
    main_TcpNet ();
  }
}
```

efs.EraseChip

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*EraseChip) (void);           /* Optional, NULL if not
existing */
} const EFS_DRV;
```

Description

The **EraseChip** function is a user-provided routine that globally erases the flash memory device.

The **EraseChip** function is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

- This function is **optional**. If the flash device does not support global erase, or only a proportion of available flash memory space is used for storing files, the value for this function should be set to **NULL** in the control block.

Return Value

The **EraseChip** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseSector, efs.Init, efs.ProgramPage, efs.ReadData, efs.UnInit

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL EraseChip (void) {
 /* Global Erase complete Flash Memory. */

  M16(base_adr + 0xAAA) = 0xAA;
  M16(base_adr + 0x554) = 0x55;
  M16(base_adr + 0xAAA) = ERASE;
  M16(base_adr + 0xAAA) = 0xAA;
  M16(base_adr + 0x554) = 0x55;
  M16(base_adr + 0xAAA) = ERA_CHIP;

  /* Wait until Erase Completed */
  return (Q6Polling (base_adr));
}
```

efs.EraseSector

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*EraseSector) (
    U32 adr);     /* address of sector to erase */
   ..
} const EFS_DRV;
```

Description

The **EraseSector** function is a user-provided routine that erases the flash sector specified by *adr* address.

The **EraseSector** function is is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

Return Value

The **EraseSector** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseChip, efs.Init, efs.ProgramPage, efs.ReadData, efs.UnInit

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL EraseSector (U32 adr) {
  /*  Erase Sector in Flash Memory. */
  U32 fsreg;

  M16(base_adr | 0xAAA) = 0xAA;
  M16(base_adr | 0x554) = 0x55;
  M16(base_adr | 0xAAA) = ERASE;
  M16(base_adr | 0xAAA) = 0xAA;
  M16(base_adr | 0x554) = 0x55;
  M16(adr) = ERA_SECT;

  /* Wait for Sector Erase Timeout. */
  do {
    fsreg = M16(adr);
  } while ((fsreg & DQ3) < DQ3);

  /* Wait until Erase Completed */
  return (Q6Polling (adr));
}
```

efs.Init

Summary

```
#include <file_config.h>

typedef struct {
  BOOL (*Init) (
    U32 adr,     /* base address */
    U32 clk);    /* CPU clock frequency */
   ..
} const EFS_DRV;
```

Description

The **Init** function is a user-provided routine that initializes the Flash programming algorithm for a flash memory device. It is invoked by the **finit** function on system startup.

The *adr* argument specifies the Flash Device base address as specified in the configuration file. The *clk* argument specifies the CPU clock frequency (which may be used to adjust the timing of Flash programming algorithms).

The **Init** function is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

Return Value

The **Init** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseChip, efs.EraseSector, efs.ProgramPage, efs.ReadData, efs.UnInit

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL Init (U32 adr, U32 clk)  {
  /* Initialize flash programming functions. */
  base_adr = adr;
  return (__TRUE);
}
```

efs.ProgramPage

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*ProgramPage) (
    U32 adr,     /* data page address */
    U32 sz,      /* size of the data page */
    U8 *buf);    /* buffer containing the data */
   ..
} const EFS_DRV;
```

Description

The **ProgramPage** is a user-provided routine that programs the contents of *buf* into Flash memory starting at address *adr* for *sz* bytes.

The *adr* must be 4-byte aligned, but *buf* may be not. The buffer *sz* must be a multiple of 4.

The **ProgramPage** function is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

Return Value

The **ProgramPage** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseChip, **efs.EraseSector**, **efs.Init**, **efs.ReadData**, **efs.UnInit**

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL ProgramPage (U32 adr, U32 sz, U8 *buf) {
 /* Program Page in Flash Memory. */

  for (  ; sz; sz -= 2, adr += 2, buf += 2)  {
    M16(base_adr | 0xAAA) = 0xAA;
    M16(base_adr | 0x554) = 0x55;
    M16(base_adr | 0xAAA) = PROGRAM;
    /* 'buf' might be unaligned. */
    M16(adr) = *(__packed U16 *)buf;

    /* Wait until Programming completed */
    if (Q6Polling (adr) == __FALSE) {
      return (__FALSE);
    }
  }
  return (__TRUE);
}
```

efs.ReadData

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*ReadData) (     /* Optional, NULL for memory-mapped Flash
*/
    U32 adr,              /* data page address      */
    U32 sz,               /* size of the data page  */
    U8 *buf);             /* buffer to read data to */
   ..
} const EFS_DRV;
```

Description

The **ReadData** function is a user-provided routine that reads the data from flash memory device to a buffer.

The *adr* must be 4-byte aligned, but *buf* may be not. The buffer *sz* must be a multiple of 4.

The **ReadData** function is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

- This function is **optional**. For parallel memory-mapped flash device the value for this function should be set to **NULL** in the control block. NULL value instructs the RL-FlashFS to use internal memcpy function to read the data.

Return Value

The **ReadData** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseChip, efs.EraseSector, efs.Init, efs.ProgramPage, efs.UnInit

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV sf0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL ReadData (U32 adr, U32 sz, U8 *buf)  {
  /* Read a block of Data from SPI Flash Memory. */

  spi->SetSS (0);
  spi->Send (SPI_READ_DATA);
  spi->Send ((U8)(adr >> 16));
  spi->Send ((U8)(adr >>  8));
  spi->Send ((U8)(adr >>  0));
  spi->RecBuf (buf, sz);
  spi->SetSS (1);
  return (__TRUE);
}
```

efs.UnInit

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*UnInit) (void);
   ..
} const EFS_DRV;
```

Description

The **UnInit** function is a user-provided routine that uninitializes the Flash programming algorithm for a flash memory device. It is invoked by the **funinit** function.

It can be used if during the application run time the embedded flash needs to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of embedded flash device.

The **UnInit** function is in the **Flash driver**. The prototype is defined in file_config.h. You have to customize the function in your own flash driver.

Return Value

The **UnInit** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

efs.EraseChip, **efs.EraseSector**, **efs.Init**, **efs.ProgramPage**, **efs.ReadData**

Example

```
/* Embedded Flash Device Driver Control Block */
EFS_DRV fl0_drv = {
  Init,
  UnInit,
  ReadData,
  ProgramPage,
  EraseSector,
  EraseChip
};

static BOOL UnInit (void)  {
  /* Uninitialize flash programming functions. */
  base_adr = 0;
  return (__TRUE);
}
```

fanalyse

Summary

```
#include <rtl.h>

int fanalyse (
    const char *drive);    /* drive to analyze */
```

Description

The **fanalyse** function analyzes the Flash File System and checks for file fragmentation on the specified *drive*. If the *drive* string is empty ("") the default drive specified in the **File_Config.c** configuration file is used.

The following values are allowed for the *drive* argument:

drive	Device Type
""	Default System drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive

During analysis, the invalidated data size is compared to the stored file size and a fragmentation factor is calculated. A factor of 0 indicates no files are fragmented while a factor of 255 indicates most files are fragmented. If the fragmentation factor is high (above 50) you should invoke the **fdefrag** function to defragment the drive.

The **fanalyse** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- The **fanalyse** function can not analyze a Memory Card drive. If the specified *drive* is "M:", **fanalyse** returns an error.

Return Value

The **fanalyse** function returns the fragmentation factor for the *drive*. The fragmentation factor is a number from 0-255.

See Also

fcheck, fdefrag

Example

```
#include <rtl.h>

void free_space (void) {

  printf ("\nFree space before defrag: %d bytes.", ffree(""));
  if (fanalyse("") > 50) {
    fdefrag ("");
  }
  printf ("\nFree space after defrag: %d bytes.", ffree(""));
}
```

fat.CheckMedia

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32   (*CheckMedia)  (void);            /* Optional, NULL if not
existing */
} const FAT_DRV;
```

Description

The **CheckMedia** is a user-provided routine that checks the Storage Media status of a **removable media** (ie. SD/MMC Memory Card, USB Flash dongle). It checks the Media Detect and Write Protect status.

If this status is not available, or the media is non-removable (ie. NAND Flash device), this function might be omitted. In this case enter the **NULL** value for CheckMedia into the FAT Driver control block. It is also possible to provide this function, which always returns **M_INSERTED** status.

The **CheckMedia** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

note

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.
- The **CheckMedia** function of the FAT driver calls the CheckMedia function of the 2nd level MCI driver or SPI driver.

Return Value

The **CheckMedia** function returns the or-ed status of the following values:

- **M_INSERTED**
  Media is inserted in the socket.
- **M_PROTECTED**
  Media is read-only. Lock slider on SD Card is in position Locked.

See Also

fat.Init, fat.ReadInfo, fat.ReadSect, fat.UnInit, fat.WriteSect

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static U32 CheckMedia (void) {
  /* Read Device Detected status. */

  if (media_ok == __FALSE) {
    /* Allow to initialize the media first. */
    return (M_INSERTED);
  }
  /* Allow USB Host to detect and enumerate the device. */
  usbh_engine();
  if (usbh_msc_status () == __TRUE) {
    return (M_INSERTED);
  }
  return (0);
}
```

fat.Init

Summary

```
#include <file_config.h>

typedef struct {
  BOOL (*Init) (
    U32 mode);       /* Init mode IO or Media */
   ..
} const FAT_DRV;
```

Description

The **Init** function is a user-provided routine that initializes the FAT driver. It is invoked by the **finit** function on system startup.

The argument *mode* specifies the initialization mode:

Mode	Description
DM_IO	Initialize the IO peripherals.
DM_MEDIA	Initialize the storage media ie. a Memory Card.

The Init function is called twice. First with parameter **DM_IO** to initialize the peripherals, enable system clocks, configure interrupts etc.

If the first call was successful, and the Init function has returned __TRUE, the system calls the Init function again, but with *mode* argument set to **DM_MEDIA**. In this phase, the storage media should be initialized.

In the MSD Driver, this function initializes the USB host stack and USB host controller hardware and allows enumeration of **Mass Storage Device** if such a device is connected to the USB host bus.

The **Init** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

note

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.
- The **Init** function of the FAT driver calls the Init function of the 2nd level MCI driver, SPI driver or NAND driver.
- The Flash File System calls the **Init** function at system startup. The FlashFS might call the function again if Mass Storage Device Hot Swapping is used.

Return Value

The **Init** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

fat.CheckMedia, **fat.ReadInfo**, **fat.ReadSect**, **fat.UnInit**, **fat.WriteSect**

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL Init (U32 mode) {
  /* Initialize USB Host. */
  U32 cnt;

  if (mode == DM_IO) {
    /* Initialise USB hardware. */
    media_ok = __FALSE;
    return (usbh_init());
  }

  if (mode == DM_MEDIA) {
    for (cnt = 0; cnt < 1000; cnt++) {
      usbh_engine();
      if (usbh_msc_status () == __TRUE) {
        media_ok = __TRUE;
        return (__TRUE);
      }
      Delay (500);
    }
  }
  return (__FALSE);
}
```

fat.ReadInfo

Summary

```
#include <file_config.h>

typedef struct {
    ..
  BOOL  (*ReadInfo)  (
    Media_INFO *cfg);       /* Structure where to write the read
info. */
    ..
} const FAT_DRV;
```

Description

The **ReadInfo** function is a user-provided routine that reads the media configuration info to a structure. This information is used by flash file system to check if the Storage Media is compatible, or for formatting the Storage Media.

The argument *cfg* specifies the media configuration info (block_cnt, read_blen, write_blen).

The **ReadInfo** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

note

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.

Return Value

The **ReadInfo** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

fat.CheckMedia, fat.Init, fat.ReadSect, fat.UnInit, fat.WriteSect

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL ReadInfo (Media_INFO *info) {
  /* Read Mass Storage Device configuration. */
  U32 blen;

  if (!usbh_msc_read_config (&info->block_cnt, &blen)) {
    /* Fail, Mass Storage Device configuration was not read. */
    return (__FALSE);
  }
  info->write_blen = info->read_blen = (U16)blen;
  return (__TRUE);
}
```

fat.ReadSect

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*ReadSect) (
    U32 sect,            /* Absolute sector address. */
    U8 *buf,             /* Location where to write the read data.
*/
    U32 cnt);            /* Number of sectors to read. */
   ..
} const FAT_DRV;
```

Description

The **ReadSect** function is a user-provided routine that reads one or more sectors from the FAT Device to a buffer.

The argument *buf* is a pointer to the buffer that stores the data. The argument *sect* specifies the starting sector from where the data are read. The argument *cnt* specifies the number of block to be read.

The **ReadSect** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

note

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.
- Sector size is 512 bytes.

Return Value

The **ReadSect** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

fat.CheckMedia, fat.Init, fat.ReadInfo, fat.UnInit, fat.WriteSect

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL ReadSector (U32 sect, U8 *buf, U32 cnt) {
  /* Read single/multiple sectors from Mass Storage Device. */

  return (usbh_msc_read (sect, buf, cnt));
}
```

fat.UnInit

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL  (*UnInit)  (
    U32  mode);       /* Uninit mode IO or Media */
   ..
} const FAT_DRV;
```

Description

The **UnInit** function is a user-provided routine in the **FAT driver** that uninitializes the FAT driver. It is invoked by the **funinit** function.

The argument *mode* specifies the uninitialization mode:

Mode	Description
DM_IO	Uninitialize the IO peripherals.
DM_MEDIA	Uninitialize the storage media ie. a Memory Card.

The UnInit function is called twice. First with parameter **DM_MEDIA** to uninitialize the storage media, for example the Memory Card or USB Flash dongle.

If the first call was successful, and the UnInit function has returned __TRUE, the system calls the UnInit function again. The argument *mode* is now set to **DM_IO** to uninitialize the peripherals, disable system clocks, disable peripheral interrupts etc.

It can be used if during the application run time the drive volume to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of the drive.

The **UnInit** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.
- The **UnInit** function of the FAT driver calls the UnInit function of the 2nd level MCI driver, SPI driver or NAND driver.

Return Value

The **UnInit** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

fat.CheckMedia, fat.Init, fat.ReadInfo, fat.ReadSect, fat.WriteSect

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL UnInit (U32 mode) {
  /* UnInitialize USB Host. */
  if (mode == DM_IO) {
    /* Initialise USB hardware. */
    return (usbh_uninit());
  }
  if (mode == DM_MEDIA) {
    return (__TRUE);
  }
  return (__FALSE);
}
```

fat.WriteSect

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*WriteSect) (
    U32 sect,             /* Absolute sector address. */
    U8 *buf,              /* Pointer to buffer with the data to
write. */
    U32 cnt);             /* Number of sectors to read. */
   ..
} const FAT_DRV;
```

Description

The **WriteSect** function is a user-provided routine that writes one or more sectors from a buffer to the FAT Device.

The argument *sect* specifies the starting sector to where data are written. The argument *buf* is a pointer to the buffer that holds the data that should be written. The argument *cnt* specifies the number of sectors to be written.

The **WriteSect** function is in the **FAT driver**. The prototype is defined in file_config.h. You have to customize the function in your own FAT driver.

note

- The FAT driver for Memory Card and NAND Flash drive is provided and configured internally from File_Config.c file.
- Sector size is 512 bytes.

Return Value

The **WriteSect** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

fat.CheckMedia, fat.Init, fat.ReadInfo, fat.ReadSect, fat.UnInit

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL WriteSector (U32 sect, U8 *buf, U32 cnt) {
  /* Read single/multiple sectors from Mass Storage Device. */

  return (usbh_msc_write (sect, buf, cnt));
}
```

fcheck

Summary

```
#include <rtl.h>

int fcheck (
    const char *drive);     /* drive to check */
```

Description

The **fcheck** function checks the consistency of the Flash File System on the specified *drive*. If the *drive* string is empty (""), **fcheck** uses the default drive specified in the **File_Config.c** configuration file.

The following values are allowed for the *drive* argument:

drive	Device Type
""	Default System drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive

The **fcheck** function determines if the Flash File System has been initialized. If this check fails the Flash or RAM Device must be formatted.

The following errors are detected:

- Invalid file ID present,
- Data space overlapping allocation records,
- Allocation end-pointers not in ascending order.

The **fcheck** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- The **fcheck** function can not check a Memory Card drive. If the specified *drive* is "M:", **fcheck** returns an error.

Return Value

The **fcheck** function returns a value of 0 if no errors were found in the file consistency check. A non-zero return value indicates an error was encountered.

See Also

fanalyse, **fdefrag**

Example

```
#include <rtl.h>

void tst_files (void) {

  if (fcheck ("R:") != 0) {
    printf ("Flash File System inconsistent, formatting...\n");

    if (fformat ("R:") != 0) {
      printf ("Formatting failed.\n");
    }
    else {
      printf ("Format done.\n");
    }
  }
}
```

fclose

Summary

```
#include <stdio.h>

int fclose (
    FILE *stream);    /* file stream to close */
```

Description

The **fclose** function closes the file *stream* opened by the **fopen** function. All buffers associated with the stream are flushed prior to closing. Buffers allocated by the system are released when the stream is closed.

The **fclose** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fclose** function returns a value of 0 if the stream is successfully closed. A return value of **EOF** indicates an error.

See Also

fflush, fopen

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fclose (void) {
  FILE *fin;

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    // process file content
    fclose (fin);
  }
}
```

fdefrag

Summary

```
#include <rtl.h>

int fdefrag (
    const char *drive);     /* drive to defragment */
```

Description

The **fdefrag** function defragments the Flash File System on the specified *drive*. If the *drive* string is empty ("") the default drive specified in the **File_Config.c** configuration file is used.

The following values are allowed for the *drive* argument:

drive	Device Type
""	Default System drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive

During defragmentation, the **fdefrag** function reorganizes the memory used by the file system and increases the number of available Flash pages.

The **fdefrag** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- Invoke this function only when the system is in an idle state and no files are open. If files are open, **fdefrag** aborts the defragmentation.
- The **fdefrag** function can not defragment a Memory Card drive. If a specified *drive* is "M:", **fdefrag** returns an error.
- You may use the **fanalyse** function to determine if defragmentation is required.

Return Value

The **fdefrag** function returns one of the following completion codes:

- **0**
  No error, function completed successfully.
- **1**
  Files are open for reading or writing. Defragmentation was canceled.
- **2**
  No Flash pages were available to use for defragmentation. Defragmentation was canceled.

See Also

fanalyse, fcheck

Example

```
#include <rtl.h>

void free_space (void) {

  printf ("\nFree space before defrag: %d bytes.", ffree("F:"));
  if (fanalyse("F:") > 50) {
    fdefrag ("F:");
  }
  printf ("\nFree space after defrag: %d bytes.", ffree("F:"));
}
```

fdelete

Summary

```
#include <rtl.h>

int fdelete (
    const char *filename);     /* file to delete */
```

Description

The **fdelete** function deletes the file specified by *filename*. The *filename* may contain a drive prefix that specifies where the file exists. If the drive prefix is omitted the default drive specified in **File_Config.c** configuration file is used.

The following drive prefixes are allowed in *filename* argument:

Drive Prefix	Storage Medium
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

For the **M:** drive, a filename must contain a **path**, otherwise a file from the root folder will be deleted. To delete a **subfolder** parameter *filename*, which is in this case a directory name, must contain also a terminating **backslash** character.

The **fdelete** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

note

- To maintain compatibility, rtl.h defines the identical function **unlink** as a macro that is substituted by the **fdelete** function.

Return Value

The **fdelete** function returns a value of 0 if successful. A non-zero return value indicates an error or file not found condition.

See Also

fformat, **frename**

Example

```
#include <rtl.h>

void tst_fdelete (void) {

  /* Delete a file from default drive. */
  if (fdelete ("TEST.TXT") == 0) {
    printf ("Deleted: TEST.TXT\n");
  }

  /* Delete a file from RAM FS. */
  if (fdelete ("R:DATA.LOG") == 0) {
    printf ("Deleted: DATA.LOG\n");
  }

  /* Delete a file from SD Card located in subfolder. */
  if (fdelete ("M:\\Working folder\\Temporary log.txt") == 0) {
    printf ("Deleted: Temporary log.txt\n");
  }

  /* Delete a folder from SD Card (if empty). */
  if (fdelete "M:\\Working folder\\") == 0) {
    printf ("Deleted: Working folder.\n");
  }
}
```

feof

Summary

```
#include <stdio.h>

int feof (
    FILE *stream);     /* file stream to check */
```

Description

The **feof** function determines if the end of *stream* has been reached. Once the end-of-file is reached, subsequent read operations return an end-of-file indicator until the file position changes (via a call to **fseek** or **rewind**).

The **feof** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **feof** function returns a value of 0 if no attempts have been made to read past the end of the *stream*. A non-zero value is returned once an attempt is made to read past the end-of-file.

See Also

ferror, ftell, rewind

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_feof (void) {
  FILE *fin;
  char ch;

  if (fin = fopen ("Test_feof") != NULL) {
    // Read all characters from the file
    while (!eof (fin)) {
      ch = fgetc (fin);
    }
    fclose (fin);
  }
}
```

ferror

Summary

```
#include <stdio.h>

int ferror (
    FILE *stream);    /* file stream to check */
```

Description

The **ferror** function tests for a read or write error on *stream*. If an error has occurred, the error indicator for *stream* remains set until the file is closed (using **fclose**) or rewound (using **rewind**).

The **ferror** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **ferror** function returns a value of 0 if no error has occurred. Otherwise, a non-zero value is returned.

See Also

fclose, feof, rewind

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_ferror (void) {
  FILE *fin;
  char ch;

  if (fin = fopen ("Test_ferror") != NULL) {
    // Read all characters from the file
    while (!eof (fin)) {
      ch = fgetc (fin);
      if (ferror (fin)) {
        printf ("File read error!\n");
        break;
      }
    }
    fclose (fin);
  }
}
```

ffind

Summary

```
#include <rtl.h>

int ffind (
    const char *pattern,     /* pattern to match files to */
    FINFO *info);             /* file info structure */
```

Description

The **ffind** function searches the Flash File System directory for files matching the specified *pattern* filter. Matching file information is stored in the *info* structure. **ffind** is invoked once for each matching file until a non-zero value is returned.

The *pattern* filter may include an optional drive prefix. If the drive prefix is excluded, **ffind** uses the default drive specified in the **File_Config.c** configuration file.

The following drive prefixes are allowed in *pattern* argument:

Drive Prefix	Storage Medium
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The *pattern* filter must include a filename pattern to match. Wildcards may be included in the filename. For example:

Pattern	Description
"*****" "***.***"	Search for all files in the directory.
"**abc***"	Search for files that begin with *abc*.
"***.htm**"	Search for files that end with *.htm*.
"**abc*.text**"	Search for files that begin with *abc* and that end with *.text*.

The **ffind** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- Before invoking the **ffind** function, you must set the fileID member of the *info* structure to 0. For example:

```

  ..
info.fileID = 0;
while (ffind ("R:*.*",&info) == 0) {
  ..
```

  This member of the *info* structure is used to identify the first call to **ffind**.

- Flash File System filenames are null-teminated strings.
- The dot character ('.') has no special meaning for the Flash File System. It is not used as a separator for the file name and file type.

Return Value

The **ffind** function returns a value of 0 to indicate a new file matching the *pattern* was found. A non-zero return value indicates no matching files were found.

If a matching file was found, the *info* structure is set to the information for the matching file. *info* contains the:

- **Filename**: The filename is a string limited to a maximum of 32 characters.
- **File Size**: The file size is the length of the file in bytes.
- **File ID**: The file ID is a file identification number assigned to a file by the Flash File System.

See Also

ffree

Example

```
#include <rtl.h>

void file_directory (void) {
  FINFO info;

  /* 'info.fileID' must initially be set to 0. */
  info.fileID = 0;

  while (ffind ("R:*.*",&info) == 0) {
    printf ("\n%-32s %5d bytes, ID: %04d",
            info.name,
            info.size,
            info.fileID);
  }
  if (info.fileID == 0) {
    printf ("\nNo files...");
  }
}
```

fflush

Summary

```
#include <stdio.h>

int fflush (
    FILE *stream);          /* stream to flush */
```

Description

The **fflush** function flushes the specified file *stream*. If the file associated with stream is open for output, the contents of the buffer associated with the stream are written to the file.

The **fflush** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Note

- This function writes the File Allocation Record to the file system along with the file data. Unlike the **fclose** function, the **fflush** function leaves the file open for continued writes.

Return Value

The **fflush** function returns a value of 0 if successful. A return value of **EOF** indicates an error.

See Also

fclose, fopen

Example

```
#include <rtl.h>
#include <stdio.h>

void main (void) {
  FILE *fout;
  int i;

  fout = fopen ("Flush.test","r");
  if (fout == NULL) {
    printf ("File open error!\n");
  }
  else {
    for (i = 'A'; i < 'Z'; i++) {
      fputc (i, fout);
    }
    // Now flush the file buffers
    fflush (fout);
    fputs ("Write to an empty file buffer.", fout);
    fclose (fout);
  }
}
```

fformat

Summary

```
#include <rtl.h>

int fformat (
    const char *drive);     /* drive to format */
```

Description

The **fformat** function formats the Flash File System storage media on the specified *drive*. If the *drive* string is empty (""), **fformat** uses the default drive specified in the **File_Config.c** configuration file.

The following values are allowed for the *drive* argument:

drive	Storage Medium
""	Default System drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

All Flash File System drives must be formatted before any files are created on the devices.

- **Flash devices** must be formatted once when the system is started the first time. They are erased sector by sector as specified in the **FS_FLASHDEV.C** configuration file.
- **RAM devices** must be formatted every time when the system starts. They are cleared sector by sector to an erased value (0x00 by default).
- Non-volatile and zero-power **RAM devices** must be formatted once when the system is started the first time. Since these devices are battery backed up and do not lose their contents when power is removed. They do not require reformatting each time the system starts. Non-volatile and zero-power RAM devices are cleared sector by sector to an erased value (0x00 by default).
- The **Memory Card** must be formatted before its first use. If a Memory Card has been formatted once, there is no need to format it again. The function formats the Memory Card optimized for 12-bit, 16-bit AND 32-bit FAT type. Cluster size and Cluster 2 alignment are optimized for the best Card performance. The Flash File System supports SD, SDHC and MMC Flash Memory Cards with a maximum capacity of 32 GBytes.

The **fformat** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- The **fformat** function erases all files written to the *drive*.
- The **fformat** function closes all opened files on the *drive*. All existing file handles become invalid. Reading or writing to such files after formatting the drive may produce unpredictable results or file corruption.
- When formatting a **Memory Card**, you can specify a drive label in the argument *drive* using the format "M:Drive_Label". The label is written to the drive after the formatting completes. The drive label must have a maximum length of 11 characters and cannot include spaces or special characters.
- The **fformat** function for Memory Card can accept additional paramters in the argument string:

- **/FAT32** - tells the format function to format the Memory Card using 32-bit FAT file system.
- **/WIPE** - tells the format function to clear all the saved data on the Memory Card. All sectors on the Memory Card are overwritten with the default value of 0xFF. Note that this command might take long time to execute on high capacity memory cards.

The following example will format the Memory Card with 12-bit or 16-bit FAT file system and clear all the data:

```
fformat ("M:SD_CARD /WIPE");
```

Return Value

The **fformat** function returns a value of 0 when formatting is successful. A non-zero return value indicates an error was encountered.

See Also

fcheck, fdelete

Example

```
#include <rtl.h>

void tst_fformat (void) {

  /* Format a Flash Drive. */
  if (fformat ("F:") != 0) {
    printf ("Flash File System format failed.\n");
  }
  else {
    printf ("Flash File System initialized.\n");
  }

  /* Format an SD Memory Card with Volume Label. */
  if (fformat ("M:SD_CARD") != 0) {
    printf ("SD Memory Card format failed.\n");
  }
  else {
    printf ("SD Memory Card formatted.\n");
  }
}
```

ffree

Summary

```
#include <rtl.h>

U64 ffree (
    const char *drive);     /* drive to check for free space */
```

Description

The **ffree** function calculates the free space in the Flash File System on the specified *drive*. If the *drive* string is empty ("") the default drive specified in the **File_Config.c** configuration file is used.

The following values are allowed for the *drive* argument:

drive	Storage Medium
""	Default System drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

Flash devices are organized as a number of fixed-size memory blocks. When erasing a block, Flash devices require that the entire block (not just a portion) is erased. So, to change data in a Flash block, you must first erase the entire block.

Multiple files may be stored in a single Flash block. In fact, a Flash block can simultaneously hold valid file data, invalid file data (which has been programmed but not yet erased), and free space.

When some, but not all, of the files stored in a single block are erased by the Flash File System, the whole Flash block is not erased because it still contains valid data. Only the allocation information of deleted files is destroyed. Programmed data is invalidated but remains present in flash block. When all data programmed into a Flash block is invalidated, the system finally erases it.

Memory is managed similarly in a RAM-base File System. The **ffree** function returns the size of free memory. Invalidated data is considered to be used memory. It is normal, when you have stored two small files and subsequently deleted one of them, to obtain the same return value from **ffree** as when both of the files were present.

The **ffree** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- The amount of free space reported by **ffree** may be misleading due to the memory management algorithm.

Return Value

The **ffree** function returns the amount of free space on *drive* in bytes.

See Also

ffind, fformat

Example

```
#include <rtl.h>

void free_space (void) {
  printf ("Flash Drive free: %lld bytes.\n", ffree("F:"));
  printf ("Ram Drive free: %lld bytes.\n", ffree("R:"));
  printf ("SD Card Drive 1 free: %lld bytes.\n", ffree("M0:"));
  printf ("SD Card Drive 2 free: %lld bytes.\n", ffree("M1:"));
}
```

fgetc

Summary

```
#include <stdio.h>

int fgetc (
    FILE *stream);     /* stream to read from */
```

Description

The **fgetc** function reads a single character from *stream* and updates the file pointer to point to the next character.

The **fgetc** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fgetc** function returns the character read as an int type. An **EOF** is returned for error or end-of-file conditions.

Use the **feof** or **ferror** functions to distinguish an error from end-of-file.

See Also

feof, ferror, fgets, fputc, ungetc

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fgetc (void) {
  FILE *fin;
  int ch;

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    // dump the text file to a screen
    while ((ch = fgetc (fin)) != EOF) {
      putchar (ch);
    }
    fclose (fin);
  }
}
```

fgets

Summary

```
#include <stdio.h>

char *fgets (
    char *string,      /* string to write to */
    int  n,             /* length of string */
    FILE *stream);     /* file stream to read from */
```

Description

The **fgets** function reads a string from the input *stream* and stores it in *string*. Characters are read from the current stream position...

- ...up to and including the first new-line ('\n'),
- ...up to the end of the stream,
- ...until the number of characters read is equal to *n*-1,

whichever comes first.

A null character ('\0') is appended to *string*.

The **fgets** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fgets** function returns *string* if successful. **NULL** is returned to indicate an error or end-of-file condition.

Use the **feof** or **ferror** functions to distinguish an error from end-of-file.

See Also

feof, **ferror**, **fgetc**, **fputs**

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fgets (void) {
  FILE *fin;
  char line[80];

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    while (fgets (line, sizeof (line), fin) != NULL) {
      puts (line);
    }
    fclose (fin);
  }
}
```

finit

Summary

```
#include <rtl.h>

int finit (
    const char *drive);      /* drive to initialize */
```

Description

The **finit** function initializes the Flash File System. It must be called before any other file system function.

The argument *drive* specifies the drive to be initialized. The following values are allowed for the *drive* argument:

drive	Initialized Drives
NULL	All enabled drives
""	Default system drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The **finit** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Note

- If the **finit** function is not invoked before other file system functions, the Flash File System may crash.
- Calling the **finit** for the drive, which is not enabled in the File_Config.c will fail.

Return Value

The **finit** function returns a value of 0 if successful. A non-zero return value indicates an error.

See Also

funinit

Example

```
#include <rtl.h>

void main (void) {

  /* Initialize the FlashFS. */
  finit (NULL);
   ..
}
```

FIQ_Handler

Summary

```
#include <demo.h>

void FIQ_Handler  (
  void);
```

Description

The **FIQ_Handler** function outputs the audio data from the host to the speaker. the function is implemented as a fast interrupt request function that runs every 31.25 µs. It scales the audio data according to the volume and mute settings before writing the value to the speaker’s register. The function also calculates the loudness over 32 ms and outputs this using the LEDs.

Modify this function to suit the application product hardware.

The **FIQ_Handler** function is part of the Application layer of the RL-USB Software Stack.

Note

- The function starts to write (DataRun=1) to the speaker register when at least half the buffer (DataBuf) contains data. The function stops writing (DataRun=0) to the speaker register when no data are available in the buffer.

Return Value

None.

See Also

ADC_IF_GetRequest, ADC_IF_SetRequest

Example

```
#include <demo.h>

void FIQ_Handler (void)  {


  …
  val = DataBuf[DataOut];        // Get the audio data sent by the
host
  …
  val *= volume;                 // Adjust the data according to
the volume
  …
  if (Mute)  {
    val = 0x8000;                // Change the data to mute value
  }

  DACR = val & 0xFFC0;           // Write the data to the speaker
register
  …
}
```

fopen

Summary

```
#include <stdio.h>

FILE *fopen (
    const char *filename,    /* pathname of file */
    const char *mode);        /* type of access */
```

Description

The **fopen** function opens the file specified by *filename*. Any valid string is allowed for *filename*. The *filename* may contain a drive prefix which specifies the medium where the file should be opened. If the drive prefix is omitted the default drive specified in the **FILE_CONFIG.C** configuration file is used.

The following drive prefixes are allowed in *filename* argument:

Drive Prefix	Storage Medium
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The *mode* defines the type of access permitted for the file. It may have one of the following values.

Mode	Description
"**r**"	Opens a file for reading. If the file does not exist **fopen** fails.
"**w**"	Opens an empty file for writing. If the file already exists, its contents are destroyed. If the file does not exist, an empty file is opened for writing.
"**a**"	Opens a file for writing. If the file already exists, data is appended at the end of file. If the file does not exist, an empty file is opened for writing.

For the **M:** and **U:** drives, a filename must contain a **path**, othervise a file from the root folder will be referenced or created. To create a **subfolder** parameter *filename* must containg also a **path**. If the specified subfolder does not exist, the system will create one and then create a file in this subfolder.

The **fopen** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Note

- The **fopen** function must be called to create a file handle before any other file functions are invoked.
- Use the "w" mode with care as it can destroy existing files.
- Read/write modes are currently not supported.
- You cannot open more than one file stream for writing to the same *filename*. If *filename* is open for writing, subsequent calls to **fopen** for writing to *filename* fail until you close *filename* using the **fclose** function.

Return Value

The **fopen** function returns a pointer to the open file stream structure. A null pointer value indicates an error.

See Also

fclose, fflush

Example

```
#include <rtl.h>
#include <stdio.h>
```

```
void tst_fopen (void) {
  FILE *f;

  /* Read a file from default drive. */
  f = fopen ("Test.txt","r");
  if (f == NULL) {
    printf ("File not found!\n");
  }
  else {
    // process file content
    fclose (f);
  }

  /* Create a file in subfolder on SD card. */
  f = fopen ("M:\\Temp_Files\\Dump_file.log","w");
  if (f == NULL) {
    printf ("Failed to create a file!\n");
  }
  else {
    // write data to file
    fclose (f);
  }
}
```

fprintf

Summary

```
#include <stdio.h>

int fprintf (
    FILE *stream,            /* file stream to write to */
    const char *format,      /* format string */
    ...);                     /* additional arguments */
```

Description

The **fprintf** function formats a series of strings and numeric values and writes the resulting string to *stream*. The *fmtstr* argument is a pointer to a format string which has the same form and function as the **printf** function's format string. The list of *arguments* are converted and output according to the corresponding format specifications in *fmtstr*.

The **fprintf** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fprintf** function returns the number of bytes actually written to *stream*.

See Also

fscanf

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fprintf (void) {
  FILE *fout;
  int i = 56;

  fout = fopen ("Test.txt","w");
  if (fout == NULL) {
    printf ("File open error!\n");
  }
  else {
    fprintf (fout, "printf test: val = %i fval =%f\n", i, i *
3.45);
    fclose (fout);
  }
}
```

fputc

Summary

```
#include <stdio.h>

int fputc (
    int  c,             /* character to write */
    FILE *stream);     /* file stream to write to */
```

Description

The **fputc** function writes a single character, *c*, to *stream* and increments the file pointer.

The **fputc** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fputc** function returns the character written. A return value of **EOF** indicates an error.

See Also

fgetc, fputs, ungetc

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fputc (void) {
  FILE *fout;
  int ch;

  fout = fopen ("Test.txt","w");
  if (fout == NULL) {
    printf ("File open error!\n");
  }
  else {
    // copy the stdin to a file
    while ((ch = getchar ()) != EOF) {
      fputc (ch, fout);
    }
    fclose (fout);
  }
}
```

fputs

Summary

```
#include <stdio.h>

int fputs (
    const char *string,     /* string to output */
    FILE *stream);          /* file stream to write to */
```

Description

The **fputs** function writes *string* to the output file *stream* at the current file position.

The **fputs** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fputs** function returns 0 if successful. A return value of **EOF** indicates an error.

See Also

fgets, fputc

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fputs (void) {
  FILE *fout;

  fout = fopen ("Test.txt","w");
  if (fout == NULL) {
    printf ("File open error!\n");
  }
  else {
    fputs("This is an example for fputs.\n", fout);
    fclose (fout);
  }
}
```

fread

Summary

```
#include <stdio.h>

U32 fread (
    void *buffer,     /* storage buffer for data */
    U32 size,         /* size of each item */
    U32 count,        /* number of items to read */
    FILE *stream);    /* file stream to read from */
```

Description

The **fread** function reads, from the input *stream*, up to *count* items of *size* bytes and stores them in *buffer*. The file pointer associated with *stream* is increased by the number of bytes actually read.

The **fread** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fread** function returns the number complete items actually read. This number may be less than *count* if an error occurs or if the end-of-file is reached.

Use the **feof** or **ferror** functions to distinguish an error from end-of-file.

See Also

feof, ferror, fgetc, fgets, fwrite

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fread (void) {
  int count[10];
  FILE *fin;

  fin = fopen ("Counter.log","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    fread (&count[0], sizeof (int), 10, fin);
    fclose (fin);
  }
}
```

frename

Summary

```
#include <rtl.h>

int frename (
    const char *oldname,      /* old filename */
    const char *newname);     /* new filename */
```

Description

The **frename** function changes the filename of the *oldname* to *newname*. *oldname* must be the name of an existing file and *newname* must be a valid filename that is not the name of an existing file.

The *oldname* may contain a drive prefix and optional path information (for M: and U: drives), that specifies where the file exists. If the drive prefix is omitted the default drive specified in **File_Config.c** configuration file is used.

The following drive prefixes are allowed in *oldname* argument:

Drive Prefix	Storage Medium
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The **frename** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

note

- For compatibility, the **rename** function is also defined.

Return Value

The **frename** function returns a value of 0 if successful. A non-zero return value indicates an error.

See Also

fdelete, fformat

Example

```
#include <rtl.h>

void tst_frename (void) {

  if (frename ("F:Test.txt", "New name.txt") == 0) {
    printf ("Rename Successful.\n");
  }
}
```

fs_get_date

Summary

```
#include <rtl.h>

U32 fs_get_date (void);
```

Description

The **fs_get_date** function returns the current date. The Flash File System calls the function to update the file write or file access date in the File Information Record. The function packs the year, month, and day values in the 3 least significant bytes of the 4-byte return value.

The **fs_get_date** function is part of RL-FlashFS. The prototype is defined in file_config.h. You can customize the function in fs_time.c

note

- You must complete the **fs_get_date** function yourself.
- To replace the default file time functions from the FlashFS library with your own, both **fs_get_time** and **fs_get_date** functions must be provided in your project.

Return Value

The **fs_get_date** function returns the current date.

See Also

fs_get_time

Example

```
U32 fs_get_date (void) {
  /* Return Current Date for FAT File Time stamp. */
  U32 d,m,y,date;

  /* Modify here, add a system call to read RTC. */
  /* Day:   1 - 31 */
  /* Month: 1 - 12 */
  /* Year:  1980 - 2107 */
  d = 1;
  m = 11;
  y = 2006;

  date = (y << 16) | (m << 8) | d;
  return (date);
}
```

fs_get_time

Summary

```
#include <rtl.h>

U32 fs_get_time (void);
```

Description

The **fs_get_time** function returns the current time. The Flash File System calls the function to update the file write or file access time in the File Information Record. The function packs the hour, minute, and second values in the 3 least significant bytes of the 4-byte return value.

The **fs_get_time** function is part of RL-FlashFS. The prototype is defined in file_config.h. You can customize the function in fs_time.c

note

- You must complete the **fs_get_time** function yourself.
- To replace the default file time functions from the FlashFS library with your own, both **fs_get_time** and **fs_get_date** functions must be provided in your project.

Return Value

The **fs_get_time** function returns the current time.

See Also

fs_get_date

Example

```
U32 fs_get_time (void) {
  /* Return Current Time for FAT File Time stamp. */
  U32 h,m,s,time;

  /* Modify here, add a system call to read RTC. */
  /* Hours:   0 - 23 */
  /* Minutes: 0 - 59 */
  /* Seconds: 0 - 59 */
  h = 12;
  m = 0;
  s = 0;

  time = (h << 16) | (m << 8) | s;
  return (time);
}
```

fscanf

Summary

```
#include <stdio.h>

int fscanf (
    FILE *stream,               /* file stream to read from */
    const char *fmtstr,         /* format string */
    ...);                        /* additional arguments */
```

Description

The **fscanf** function reads formatted data from *stream*. Data input are stored in the locations specified by *arguments* according to the format string *fmtstr*. Each *arguments* must be a pointer to a variable that corresponds to the type defined in *fmtstr*.

The **fscanf** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fscanf** function returns the number of input fields that were successfully converted.

See Also

fprintf

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fread (void) {
  int index, count;
  FILE *fin;

  fin = fopen ("Counter.log","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    fscanf ("%d, %d",&index, &count);
    fclose (fin);
  }
}
```

fseek

Summary

```
#include <stdio.h>

int fseek (
    FILE *stream,     /* file stream */
    long offset,      /* file position offset */
    int  origin);     /* initial offset origin */
```

Description

The **fseek** function moves the file pointer associated with *stream* to the location that is *offset* bytes from *origin*. The next operation on the stream takes place at this new location.

The *origin* may be one of the following manifest constant values:

Origin Value	Description
SEEK_CUR	Current position of file pointer.
SEEK_END	End of the file.
SEEK_SET	Beginning of the file.

The **fseek** function may reposition the file pointer anywhere in the file or past the end of the file. Attempts to position the file pointer before the beginning of the file causes an error.

The **fseek** function clears the end-of-file indicator.

The **fseek** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Note

- Seeking within a file opened for "w" mode is currently unsupported.

Return Value

The **fseek** function returns a value of 0 if successful or a value of **EOF** if unsuccessful.

See Also

ftell, rewind

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fseek (void) {
  FILE *fin;
  char ch;

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    // Read the 5th character from file
    fseek (fin, 5L, SEEK_SET);
    ch = fgetc (fin);
    // Read the last character from file
    fseek (fin, -1L, SEEK_END);
    ch = fgetc (fin);
    fclose (fin);
  }
}
```

ftell

Summary

```
#include <stdio.h>

U32 ftell (
    FILE *stream);     /* file stream */
```

Description

The **ftell** macro gets the current file position associated with *stream*. The file position is an offset relative to the beginning of the stream.

The **ftell** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **ftell** macro returns the current file position.

See Also

fseek

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_ftell (void) {
  U32 fpos;
  char line[80];
  FILE *fin;

  fin = fopen ("Counter.log","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    fgets (&line, sizeof (char), fin);
    // Get position after read
    fpos = ftell (fin);
    fclose (fin);
  }
}
```

ftp_fclose

Summary

```
#include <net_config.h>

void *ftp_fclose (
    FILE* file);  /* Pointer to the file to close. */
```

Description

The **ftp_fclose** function closes the file identified by the *file* stream pointer in the function argument.

The **ftp_fclose** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_fclose** function does not return any value.

See Also

ftp_fopen, **ftp_fread**, **ftp_fwrite**

Example

```
void ftp_fclose (void *file) {
  /* Close the file opened for reading or writing. */
  fclose (file);
}
```

ftp_fdelete

Summary

```
#include <net_config.h>

BOOL ftp_fdelete (
    U8* fname );    /* Pointer to name of file to delete. */
```

Description

The **ftp_fdelete** function deletes the file specified by *fname*.

The **ftp_fdelete** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_fdelete** function returns __TRUE when the file is successfully deleted. It returns __FALSE on failure.

See Also

ftp_ffind, **ftp_frename**

Example

```
BOOL ftp_fdelete (U8 *fname) {
  /* Delete a file, return __TRUE on success. */
  if (fdelete((char *)fname) == 0) {
    return (__TRUE);
  }
  return (__FALSE);
}
```

ftp_ffind

Summary

```
#include <net_config.h>

U16 ftp_ffind (
    U8   code,       /* Function request code. */
    U8* buf,         /* Pointer to the buffer to write to. */
    U8* mask,        /* Pattern to match files to. */
    U16 buflen );   /* Length of the output buffer. */
```

Description

The **ftp_ffind** function searches the File System directory for files matching the specified *mask* filter. Matching file information is stored to the output buffer specified with *buf*. The output data must be formatted in the FTP folder listing format.

Parameter *code* specifies the request type for the **ftp_ffind** function.

Code	Request Type
0	Read file size.
1	Read last-modified time of a file.
2	List file names only (first call).
3	List file directory in extended format (first call).
4	List file names only (subsequent call).
5	List file dorectory in extended format (subsequent call).

Parameter *buflen* specifies the size of output buffer *buf*.

The **ftp_ffind** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_ffind** function returns the number of bytes written to the *buf*.

See Also

ftp_fdelete, ftp_frename

Example

```
U16 ftp_ffind (U8 code, U8 *buf, U8 *mask, U16 len) {
  /* Find file names and other file information. */
  static FINFO info;
  U32 rlen,v;
  U8 *tp;

  if (code < 4) {
    /* First call to ffind, initialize the info. */
    info.fileID = 0;
  }

  rlen = 0;
next:
  if (ffind ((char *)mask, &info) == 0) {
    /* File found, print file information. */
    if (info.name[0] == '.') {
      if ((info.name[1] == 0) || (info.name[1] == '.' &&
info.name[2]) == 0) {
        /* Ignore the '.' and '..' folders. */
        goto next;
      }
    }
    switch (code) {
      case 0:
        /* Return file size as decimal number. */
        rlen = sprintf ((char *)buf,"%d\r\n", info.size);
        break;

      case 1:
        /* Return last-modified time in format "YYYYMMDDhhmmss".
*/
        rlen  = sprintf ((char *)buf,"%04d%02d%02d",
                         info.time.year, info.time.mon,
info.time.day);
        rlen += sprintf ((char *)&buf[rlen],"%02d%02d%02d\r\n",
                         info.time.hr, info.time.min,
info.time.sec);
        break;

      case 2:
      case 4:
        /* List file names only. */
        rlen = sprintf ((char *)buf,"%s\r\n", info.name);
        break;

      case 3:
      case 5:
        /* List directory in extended format. */
        rlen  = sprintf ((char *)buf,"%02d-%02d-%02d",
                         info.time.mon, info.time.day,
info.time.year%100);
        /* Convert time to "AM/PM" format. */
        v = info.time.hr % 12;
        if (v == 0) v = 12;
        if (info.time.hr < 12) tp = "AM";
        else                   tp = "PM";
        rlen += sprintf ((char *)&buf[rlen],"
%02d:%02d%s",v,info.time.min,tp);
        if (info.attrib & ATTR_DIRECTORY) {
          rlen += sprintf ((char *)&buf[rlen],"%-21s","
<DIR>");
        }
        else {
          rlen += sprintf ((char *)&buf[rlen],"%21d", info.size);
        }
        rlen += sprintf ((char *)&buf[rlen]," %s\r\n",
info.name);
        break;
    }
  }
  return (rlen);
}
```

ftp_fopen

Summary

```
#include <net_config.h>

void *ftp_fopen (
    U8* fname,     /* Pointer to name of file to open. */
    U8* mode);     /* Pointer to mode of operation.    */
```

Description

The **ftp_fopen** function opens a file for reading or writing. The argument *fname* specifies the name of the file to open. The *mode* defines the type of access permitted for the file. It can have one of the following values:

Mode	Description
"r"	Opens the file for reading. If the file does not exist, **fopen** fails.
"w"	Opens an empty file for writing if the file does not exist. If the file already exists, its contents are cleared.

The **ftp_fopen** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_fopen** function returns a pointer to the opened file. The function returns NULL if it cannot open the file.

See Also

ftp_fclose, ftp_fread, ftp_fwrite

Example

```
void *ftp_fopen (U8 *fname, U8 *mode) {
  /* Open file 'fname' for reading or writing. */
  return (fopen ((const char *)fname, (const char *)mode));
}
```

ftp_fread

Summary

```
#include <net_config.h>

U16 ftp_fread (
    FILE* file,      /* Pointer to the file to read from. */
    U8*   buf,       /* Pointer to buffer, to store the data. */
    U16   len );     /* Number of bytes to read. */
```

Description

The **ftp_fread** reads *len* bytes from the file identified by the *file* stream pointer in the function argument. The argument *buf* is a pointer to the buffer where the function stores the read data.

The **ftp_fread** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

note

- The **ftp_fread** function must read *len* bytes. The FTP Server stops reading and closes the file if the return value is less than *len* bytes.

Return Value

The **ftp_fread** function returns the number of bytes read from the file.

See Also

ftp_fclose, **ftp_fopen**, **ftp_fwrite**

Example

```
U16 ftp_fread (void *file, U8 *buf, U16 len) {
  /* Read 'len' bytes from file to buffer 'buf'. The file will be
*/
  /* closed, when the number of bytes read is less than 'len'.
*/
  return (fread (buf, 1, len, file));
}
```

ftp_frename

Summary

```
#include <net_config.h>

BOOL ftp_frename (
    U8* fname,      /* Pointer to old filename. */
    U8* newn );     /* Pointer to new filename. */
```

Description

The **ftp_frename** function changes the filename of the *fname* to *newn*. *fname* must be the name of an existing file and *newn* must be a valid filename that is not the name of an existing file.

The **ftp_frename** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_frename** function returns __TRUE when the file is successfully renamed. It returns __FALSE on failure.

See Also

ftp_fdelete, ftp_ffind

Example

```
BOOL ftp_frename (U8 *fname, U8 *newn) {
  /* Rename a file, return __TRUE on success. */
  if (frename((char *)fname, (char *)newn) == 0) {
    return (__TRUE);
  }
  return (__FALSE);
}
```

ftp_fwrite

Summary

```
#include <net_config.h>

U16 ftp_fwrite (
    FILE* file,      /* Pointer to the file to write to. */
    U8*   buf,       /* Pointer to the buffer containing the data.
*/
    U16   len );     /* Number of bytes to write. */
```

Description

The **ftp_fwrite** function writes a block of data to the file identified by the *file* stream pointer. The argument *buf* points to the buffer containing the data that is to be written to the file. The argument *len* specifies the number of bytes to write to the file.

The **ftp_fwrite** function is in the FTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **ftp_fwrite** function returns the number of bytes written to the file.

See Also

ftp_fclose, ftp_fopen, ftp_fread

Example

```
U16 ftp_fwrite (FILE *file, U8 *buf, U16 len) {
  /* Write 'len' bytes from buffer 'buf' to a file. */
  return (fwrite (buf, 1, len, file));
}
```

funinit

Summary

```
#include <rtl.h>

int funinit (
    const char *drive);      /* drive to uninitialize */
```

Description

The **funinit** function uninitializes the Flash File System. It can be used if during the application run time the drive volume to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of the drive.

The argument *drive* specifies the drive to be uninitialized. The following options are allowed for the *drive* argument:

drive	UnInitialized Drives
NULL	All enabled drives
""	Default system drive
"**F:**" or "**F0:**"	Flash drive
"**S:**" or "**S0:**"	SPI Flash drive
"**R:**" or "**R0:**"	RAM drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The **funinit** function is in the RL-FlashFS library. The prototype is defined in rtl.h.

Return Value

The **finit** function returns a value of 0 if successful. A non-zero return value indicates an error.

See Also

finit

Example

```
#include <rtl.h>

void main (void) {
  FILE *f;

  /* Initialize the M: drive. */
  if (finit ("M:") == 0) {
    /* Update a log file on SD card. */
    f = fopen ("M:\\Logs\\Test_file.log","a");
    if (f == NULL) {
      printf ("Failed to create a file!\n");
    }
    else {
      // write data to file
      fclose (f);
    }
    /* The drive is no more needed. */
    funinit ("M:");
  }
   ..
}
```

fwrite

Summary

```
#include <stdio.h>

U32 fwrite (
    const void *buffer,     /* data to write to file */
    U32 size,               /* size of each item */
    U32 count,              /* number of items to write */
    FILE *stream);          /* file stream to write to */
```

Description

The **fwrite** function writes up to *count* items of *size* bytes from *buffer* to the file *stream*. The file pointer associated with *stream* is increased by the number of bytes actually written.

The **fwrite** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **fwrite** function returns the number of complete items actually written. This number may be less than *count* if an error occurs.

See Also

fputc, fputs, fread

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_fwrite (void) {
  int count[10];
  FILE *fout;

  fout = fopen ("Counter.log","w");
  if (fout == NULL) {
    printf ("File open error!\n");
  }
  else {
    fwrite (&count[0], sizeof (int), 10, fout);
    fclose (fout);
  }
}
```

get_host_by_name

Summary

```
#include <rtl.h>

U8 get_host_by_name (
    U8* hostn,                                /* Pointer to the
hostname. */
    void (*cbfunc)(U8 event, U8* host_ip) );  /* Function to call
when    */
                                              /* the DNS request
ends.    */
```

Description

The **get_host_by_name** function resolves the IP address of a host from a hostname. The argument *hostn* is a pointer to a NULL terminated string that specifies the hostname. The function *get_host_by_name* starts the DNS client on TCPnet, to send a request to the DNS server.

The argument *cbfunc* specifies a user provided callback function that the DNS client calls when an event ends the DNS session. The DNS client specifies the event and the host IP address (in case of DNS_EVT_SUCCESS) when calling the *cbfunc* function.

Event	Description
DNS_EVT_SUCCESS	The IP address has been resolved. The argument *host_ip* of the function cbfunc points to a 4-byte buffer containing the IP address in dotted decimal notation.
DNS_EVT_NONAME	The hostname was not found in the DNS server.
DNS_EVT_TIMEOUT	The timeout has expired before the IP address could be resolved.
DNS_EVT_ERROR	A DNS communication protocol error has occurred. This can also result from misplaced dots in the hostname or incorrect name labels.

The **get_host_by_name** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- The hostname argument hostn can also point to the dotted decimal IP address in sting format (for example "192.168.0.100"). In this case, the DNS client calls the *cbfunc* function immediately with the IP address.

Return Value

The **get_host_by_name** function returns a code that specifies either the state of the dns resolving process or an error. If the return code is an error, then the current DNS request is ignored:

Return code	Description
DNS_RES_OK	The DNS resolving process has started.
DNS_ERROR_BUSY	The DNS resolving process is still busy.
DNS_ERROR_LABEL	The hostname label is too long.
DNS_ERROR_NAME	The hostname is too long.
DNS_ERROR_NOSRV	The DNS server IP address has not been specified.
DNS_ERROR_UDPSEND	The UDP socket cannot send packets.

Example

```
static void dns_cbfunc (U8 event, U8 *ip);

void resolve_host (void) {
  U8 res;

  res = get_host_by_name ("www.keil.com",dns_cbfunc);
  switch (res) {
    case DNS_RES_OK:
      break;
    case DNS_ERROR_BUSY:
      printf("DNS Resolver is still busy. Request ignored.\n");
      break;
    case DNS_ERROR_LABEL:
      printf("Host name label too long.\n");
      break;
    case DND_ERROR_NAME:
      printf("Host name too long.\n");
      break;
    case DNS_ERROR_NOSRV:
      printf("DNS Server IP address not specified.\n");
      break;
    case DNS_ERROR_UDPSEND:
      printf("Error sending UDP packet.\n");
      break;
  }
}
```

HID_GetReport

Summary

```
#include <hiduser.h>

bool HID_GetReport  (
  void);
```

Description

The **HID_GetReport** function sends the requested report data to the host by writing the data into the endpoint 0 buffer (EP0Buf).

Call **GetInReport** to update the report variable (InReport). Modify the **HID_GetReport** function to obtain your report data and copy them into the endpoint buffer. The function supports the **HID_REPORT_INPUT** request only.

The **HID_GetReport** function is part of the USB Function Driver layer of the RL-USB Software Stack.

Note

- If you modify the **HID_GetReport** function, then you must also modify the corresponding endpoint function, **USB_EndPoint1**, because the host might use either one of these functions to obtain the report data.

Return Value

The **HID_GetReport** function returns:

- **__TRUE** if the host request is supported.
- **__FLASE** in any other case.

See Also

HID_SetReport

Example

```
#include <hiduser.h>

bool HID_GetReport (void)  {

  switch (SetupPacket.wValue.WB.H)  {
    case HID_REPORT_INPUT:
      GetInReport ();            // your function to update the
report vars.
      EP0Buf [0] = InReport;     // copy report vars to the
endpoint buffer.
      break;
    …
  }
  return (__TRUE);
}
```

HID_SetReport

Summary

```
#include <hiduser.h>

bool HID_GetReport  (
  void);
```

Description

The **HID_SetReport** function obtains report data from the host by copying them from the endpoint 0 buffer (EP0Buf).

The function calls **SetOutReport** to update other application variables. Modify the **HID_SetReport** function to obtain as many bytes as your application needs from the host. The **HID_SetReport** function supports the request **HID_REPORT_OUTPUT** only.

The **HID_SetReport** function is part of the USB Function Driver layer of the RL-USB Software Stack.

Note

- You must copy all your report data from the endpoint buffer, otherwise the report data might be lost.

Return Value

The **HID_SetReport** function returns:

- **__TRUE** if the host request is supported.
- **__FLASE** in any other case.

See Also

HID_GetReport

Example

```
#include <hiduser.h>

bool HID_SetReport (void)  {

  switch (SetupPacket.wValue.WB.H)  {
    case HID_REPORT_INPUT:
      OutReport = EPoBuf [0];     // copy report vars to the
endpoint buffer.
      SetInReport ();             // your function to update the
report vars.
      break;
    …
  }
  return (__TRUE);
}
```

http_accept_host

Summary

```
#include <net_config.h>

BOOL http_accept_host (
    U8* rem_ip,       /* IP address of the remote host. */
    U16 rem_port );   /* Remote port used for communication. */
```

Description

The **http_accept_host** function checks if a connection from the remote host is allowed or not. This allows remote host filtering. You can selectively decide which hosts are allowed to connect to the web server and which are not allowed.

The argument *rem_ip* points to a buffer containing the four octets that make up the ip address of the remote machine.The argument *rem_port* specifies the port on the remote machine.

The **http_accept_host** function is in the HTTP_CGI.c module. The prototype is defined in net_config.h.

note

- This function is **optional**. If it does not exist in the project, the library default function is used instead. The library default function accepts all remote hosts.

Return Value

The **http_accept_host** function returns __TRUE if the connection from the remote host is allowed. If the connection is not allowed, this function shall return __FALSE.

Example

```
BOOL http_accept_host (U8 *rem_ip, U16 rem_port) {

  if (rem_ip[0] == 192  &&
      rem_ip[1] == 168  &&
      rem_ip[2] == 1    &&
      rem_ip[3] == 1) {
    /* Accept a connection. */
    return (__TRUE);
  }
  /* Deny a connection. */
  return (__FALSE);
}
```

http_date

Summary

```
#include <net_config.h>

U32 http_date (
    RL_TIME* time );   /* Modification time in RL format. */
```

Description

The **http_date** function converts the RL time format to **UTC** time format. This time is used by the Web server to control the browser local caching.

The argument *time* is the time provided in RL time format.

```
typedef struct {             /* RL Time format          */
  U8  hr;                    /* Hours    [0..23]        */
  U8  min;                   /* Minutes  [0..59]        */
  U8  sec;                   /* Seconds  [0..59]        */
  U8  day;                   /* Day      [1..31]        */
  U8  mon;                   /* Month    [1..12]        */
  U16 year;                  /* Year     [1980..2107] */
} RL_TIME;
```

The **http_date** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

Return Value

The **http_date** function returns **UTF** time in binary seconds.

Example

```
U32 http_finfo (U8 *fname) {
  FINFO *info;
  U32 utc;

  info = (FINFO *)alloc_mem (sizeof (FINFO));
  info->fileID = 0;
  utc = 0;
  if (ffind ((const char *)fname, info) == 0) {
    /* File found, save creation date in UTC format. */
    utc = http_date (&info->time);
  }
  free_mem ((OS_FRAME *)info);
  return (utc);
}
```

http_fclose

Summary

```
#include <net_config.h>

void http_fclose (
    FILE* file );     /* Pointer to the file to close. */
```

Description

The **http_fclose** function closes the file identified by the *file* stream pointer in the function argument.

The **http_fclose** function is in the HTTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **http_fclose** function does not return any value.

See Also

http_fopen

Example

```
void http_fclose (void *file) {
  /* Close the file opened for reading. */
  fclose (file);
}
```

http_fgets

Summary

```
#include <net_config.h>

BOOL http_fgets (
    FILE* file,     /* Pointer to the file to read from. */
    U8*   buf,      /* Pointer to buffer, to store the read data.
*/
    U16   size );   /* Maximum length of the string to read. */
```

Description

The **http_fgets** reads up-to *size* bytes from the file identified by the *file* stream pointer in the function argument. The argument *buf* is a pointer to the buffer where the function stores the read data.

The **http_fgets** function is in the HTTP_uif.c module. The prototype is defined in net_config.h.

note

- The **http_fgets** function is used by the **script interpreter**.

Return Value

The **http_fgets** function returns __TRUE when the string was successfully read from the file. Othervise, the function returns __FALSE.

See Also

http_finfo, **http_fread**

Example

```
BOOL http_fgets (void *f, U8 *buf, U16 size) {
  /* Read a string from file to buffer 'buf'. The file will be
closed, */
  /* when this function returns __FALSE.
    */
  if (fgets ((char *)buf, size, f) == NULL) {
    return (__FALSE);
  }
  return (__TRUE);
}
```

http_finfo

Summary

```
#include <net_config.h>

U32 http_finfo (
    U8* fname );    /* Pointer to name of file. */
```

Description

The **http_finfo** reads the time, when the file identified by the *fname* was last modified.

The **http_finfo** function is in the HTTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **http_finfo** function returns last modification time in **UTC** format.

See Also

http_fgets, **http_fread**

Example

```
U32 http_finfo (U8 *fname) {
  /* Read last modification time of a file. Return time in UTC
format. */
  FINFO *info;
  U32 utc;

  info = (FINFO *)alloc_mem (sizeof (FINFO));
  info->fileID = 0;
  utc = 0;
  if (ffind ((const char *)fname, info) == 0) {
    /* File found, save creation date in UTC format. */
    utc = http_date (&info->time);
  }
  free_mem ((OS_FRAME *)info);
  return (utc);
}
```

http_fopen

Summary

```
#include <net_config.h>

void* http_fopen (
    U8* fname );    /* Pointer to name of file to open. */
```

Description

The **http_fopen** function opens a file for reading. The argument *fname* specifies the name of the file to open. If the file does not exist, **fopen** fails.

The **http_fopen** function is in the HTTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **http_fopen** function returns a pointer to the opened file. The function returns NULL if it cannot open the file.

See Also

http_fclose

Example

```
void *http_fopen (U8 *fname) {
  /* Open file 'fname' for reading. Return file handle. */
  return (fopen ((const char *)fname, "r"));
}
```

http_fread

Summary

```
#include <net_config.h>

U16 http_fread (
    FILE* file,      /* Pointer to the file to read from. */
    U8*   buf,       /* Pointer to buffer, to store the read data.
*/
    U16   len );     /* Number of bytes to read. */
```

Description

The **http_fread** reads *len* bytes from the file identified by the *file* stream pointer in the function argument. The argument *buf* is a pointer to the buffer where the function stores the read data.

The **http_fread** function is in the HTTP_uif.c module. The prototype is defined in net_config.h.

note

- The **http_fread** function must read *len* bytes. The Web Server stops reading and closes the file if the return value is less than *len* bytes.

Return Value

The **http_fread** function returns the number of bytes read from the file.

See Also

http_fgets, http_finfo

Example

```
U16 http_fread (void *f, U8 *buf, U16 len) {
  /* Read 'len' bytes from file to buffer 'buf'. The file will be
*/
  /* closed, when the number of read bytes is less than 'len'.
*/
  return (fread (buf, 1, len, f));
}
```

http_get_content_type

Summary

```
#include <net_config.h>

U8 *http_get_content_type (void);
```

Description

The **http_get_content_type** function returns a pointer to the *Content-Type* html header, which was received in the xml post request. You can use this function to check for the content type, which was submitted by the **Silverlight** web service application

The **http_get_content_type** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

note

- When a Silverlight web service application sends a request to a web server, it specifies the *Content-Type* in the HTTP header that is sent to the web server. This information is processed by TCPnet and stored internally.

Return Value

The **http_get_content_type** function returns a pointer to the content type header, which is a null-terminated string.

See Also

cgi_process_data, cgx_content_type

Example

```
void cgi_process_data (U8 code, U8 *dat, U16 len) {
  U8 *pType;

  switch (code) {
    case 4:
      /* XML encoded content type, last packet. */
      pType = http_get_content_type ();
      if (strcmp (pType,"text/xml; charset=utf-8") == 0) {
        /* 'utf-8' character set */
         ..
      }
      else {
        /* Not 'utf-8' */
         ..
      }
      return;

    case 5:
      /* XML encoded as under 4, but with more to follow. */
      return;

    default:
      /* Ignore all other codes. */
      return;
  }
}
```

http_get_info

Summary

```
#include <net_config.h>

void http_get_info (
    REMOTEM* info );    /* Pointer to memory where the IP address
gets stored */
```

Description

The **http_get_info** function obtains information about the remote machine and records the remote machine's IP address and MAC address in the memory pointed to by the *info* argument.

The **http_get_info** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

Note

- You can use the IP address or MAC address information to restrict which remote machines are allowed to perform system changes.
- For PPP and SLIP links, the function records 00-00-00-00-00-00 as the MAC address in *info*.

Return Value

The **http_get_info** function does not return any value, but the information about the remote machine gets stored in the *info* argument.

See Also

cgi_func

Example

```
U16 cgi_func (U8 *env, U8 *buf, U16 buflen, U32 *pcgi) {
  REMOTEM info;
  U16 len = 0;

   ..
  http_get_info (&info);
  /* Print a remote IP address. */
  len  = sprintf(buf,"Your IP: %d.%d.%d.%d",info.IpAdr[0],
info.IpAdr[1],
                                            info.IpAdr[2],
infn.IpAdr[3]);
  /* Print a remote ethernet MAC address. */
  len += sprintf(buf+len,"Your MAC:
%02x-%02x-%02x-%02x-%02x-%02x",
                                            info.HwAdr[0],
info.HwAdr[1],
                                            info.HwAdr[2],
info.HwAdr[3],
                                            info.HwAdr[4],
info.HwAdr[5]);
   ..
  return (len);
}
```

http_get_lang

Summary

```
#include <net_config.h>

U8* http_get_lang (void);
```

Description

The **http_get_lang** function retrieves the preferred language setting from the browser. You can use this information to implement automatic language selection for your embedded web pages.

The **http_get_lang** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

Note

- When a web browser requests a web page, it specifies the preferred language in the HTTP header that is sent to the web server. This information is processed by TCPnet and stored internally.
- You can set the language preference in Internet Explorer by selecting Tools -> Internet Options -> Languages. In Mozilla Firefox, you can set the language preference by selecting Tools -> Options -> Content -> Languages.

Return Value

The **http_get_lang** function returns a pointer to the language code, which is a null-terminated string.

See Also

cgi_func

Example

```
U16 cgi_func (U8 *env, U8 *buf, U16 buflen, U32 *pcgi) {
  U16 len = 0;
  U8 *lang;

  switch (env[0]) {
     ..
    case 'e':
      /* Browser Language - file 'language.cgi' */
      lang = http_get_lang();
      if (strcmp (lang, "en") == 0) {
        lang = "English";
      }
      else if (strcmp (lang, "en-us") == 0) {
        lang = "English USA";
      }
      else if (strcmp (lang, "de") == 0) {
        lang = "German";
      }
      else if (strcmp (lang, "de-at") == 0) {
        lang = "German AT";
      }
      else if (strcmp (lang, "fr") == 0) {
        lang = "French";
      }
      else {
        lang = "Unknown";
      }
      len = sprintf(buf,&env[2],lang,http_get_lang());
      break;
  }
  return (len);
}
```

http_get_session

Summary

```
#include <net_config.h>

U8 http_get_session (void);
```

Description

The **http_get_session** returns the current session number of the HTTP server running on TCPnet. The session number can be any value between 0 and HTTP_NUMSESS. You can customize the defined value of HTTP_NUMSESS in net_config.h.

The **http_get_session** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

note

- Storing the HTTP query string in a single global variable can result in the HTTP server sending the wrong reply to a client when several clients try to access a dynamic HTTP page at the same time. Hence it is better to have a separate variable for each session, for example an array of query strings.

Return Value

The **http_get_session** function returns the current session number of the HTTP server.

See Also

cgi_func

Example

```
/* Should be the same value as set in 'Net_Config.c' file. */
#define HTTP_NUMSESS 10

U32 answer[HTTP_NUMSESS];

U16 cgi_func (U8 *env, U8 *buf, U16 buflen, U32 *pcgi) {
  U16 len = 0;
  U32 http_session;

   ..
  http_session = http_get_session ();
  len += sprintf (buf+len,"Answer is: %d",answer[http_session]);
   ..
  return (len);
}
```

http_get_var

Summary

```
#include <net_config.h>

U8* http_get_var (
    U8*    env,          /* Pointer to a string of environment
variables */
    void* ansi,          /* Buffer to store the environment variable
value */
    U16    maxlen );     /* Maximum length of environment variable
value */
```

Description

The **http_get_var** function processes the string *env*, which contains the environment variables, and identifies where the first variable ends. The function obtains and stores the first variable and its value into the buffer pointed by *ansi*, in ansi format.

The *maxlen* specifies the maximum length that can be stored in the *ansi* buffer. If the decoded environment variable value is longer than this limit, the function truncates it to *maxlen* to fit it into the buffer.

The **http_get_var** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

Note

- The **http_get_var** function can process environment variables from the GET and POST methods.
- You can call this function from the HTTP_CGI.c module.
- The web browser uses environment variables to return user-entered information that is requested in the HTTP input form.

Return Value

The **http_get_var** function returns a pointer to the remaining environment variables to process. It returns NULL if there are no more environment variables to process.

See Also

cgi_process_data, cgi_process_var

Example

```
void cgi_process_var (U8 *qs) {
  U8 var[40];

  do {
    /* Loop through all the parameters. */
    qs = http_get_var (qs, var, 40);
    /* Check the returned string, 'qs' now points to the next. */
    if (var[0] != 0) {
      /* Returned string is non 0-length. */
      if (str_scomp (var, "ip=") == __TRUE) {
        /* My IP address parameter. */
        sscanf (&var[3],
"%bd.%bd.%bd.%bd",&LocM.IpAdr[0],&LocM.IpAdr[1],

&LocM.IpAdr[2],&LocM.IpAdr[3]);
      }
      else if (str_Scomp (var, "msk=") == __TRUE) {
        /* Net mask parameter. */
        sscanf (&var[4],
"%bd.%bd.%bd.%bd",&LocM.NetMask[0],&LocM.NetMask[1],

&LocM.NetMask[2],&LocM.NetMask[3]);
      }
      else if (str_scomp (var, "gw=") == __TRUE) {
       ..
      }
    }
  } while (qs);
}
```

igmp_join

Summary

```
#include <rtl.h>

BOOL igmp_join (
    U8   *group_ip);      /* IP Address of the Host Group. */
```

Description

The **igmp_join** function requests that this host become a member of the host group identified by *group_ip*. Before any datagrams destined to a particular group can be received, an upper-layer protocol must ask the IP module to join that group.

The **igmp_join** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **igmp_join** function returns __TRUE when this host successfully joined a host group. Otherwise, the function returns __FALSE.

See Also

igmp_leave, **udp_mcast_ttl**, **udp_send**

Example

```
U8 sgroup[4] = { 238, 0, 100, 1};
 ..
if (igmp_join (sgroup) == __TRUE) {
  printf ("This Host is a member of group: %d.%d.%d.%d\n",
           sgroup[0],sgroup[1],sgroup[2],sgroup[3]);
}
else {
  printf ("Failed to join a host group.\n");
}
 ..
```

igmp_leave

Summary

```
#include <rtl.h>

BOOL igmp_leave (
    U8  *group_ip);      /* IP Address of the Host Group. */
```

Description

The **igmp_leave** function requests that this host gives up its membership in the host group identified by *group_ip*. After the upper-layer has requested to leave the host group, datagrams destined to a particular group can not be received, but are silently discarded by the IP-layer.

The **igmp_leave** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **igmp_leave** function returns __TRUE when this host successfully left a host group. Otherwise, the function returns __FALSE.

See Also

igmp_join, **udp_mcast_ttl**, **udp_send**

Example

```
U8 sgroup[4] = { 238, 0, 100, 1};
 ..
if (igmp_leave (sgroup) == __TRUE) {
  printf ("This Host has left the group: %d.%d.%d.%d\n",
           sgroup[0],sgroup[1],sgroup[2],sgroup[3]);
}
else {
  printf ("Failed to leave a group, this host is not a
member.\n");
}
```

init_ethernet

Summary

```
#include <net_config.h>

void init_ethernet (void);
```

Description

The **init_ethernet** function initializes the ethernet controller. The function:

- enables the sending and receiving of packets
- sets the MAC ethernet address
- sets the interrupt configuration registers of the ethernet controller if the ethernet controller is configured for interrupt mode.

The **init_ethernet** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- The MAC address of the ethernet interface is typically configured by writing the six byte MAC address into the dedicated registers of the ethernet controller.
- You must provide the **init_ethernet** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- The TcpNet system frequently calls **init_ethernet** during its startup.

Return Value

The **init_ethernet** function does not return any value.

See Also

interrupt_ethernet, poll_ethernet, send_frame

Example

```
void init_ethernet (void) {
  /* Initialize the EMAC ethernet controller. */
  U32 regv,tout,id1,id2;

  /* Power Up the EMAC controller. */
  PCONP |= 0x40000000;

  /* Enable P1 Ethernet Pins. */
  if (MAC_MODULEID == OLD_EMAC_MODULE_ID) {
    /* For the first silicon rev.'-' ID P1.6 should be set. */
    PINSEL2 = 0x50151105;
  }
  else {
    /* on rev. 'A' and later, P1.6 should NOT be set. */
    PINSEL2 = 0x50150105;
  }
  PINSEL3 = (PINSEL3 & ~0x0000000F) | 0x00000005;

  /* Reset all EMAC internal modules. */
  MAC_MAC1 = MAC1_RES_TX | MAC1_RES_MCS_TX | MAC1_RES_RX |
MAC1_RES_MCS_RX |
             MAC1_SIM_RES | MAC1_SOFT_RES;
  MAC_COMMAND = CR_REG_RES | CR_TX_RES | CR_RX_RES;

  /* A short delay after reset. */
  for (tout = 100; tout; tout--);

  /* Initialize MAC control registers. */
  MAC_MAC1 = MAC1_PASS_ALL;
  MAC_MAC2 = MAC2_CRC_EN | MAC2_PAD_EN;
  MAC_MAXF = ETH_MAX_FLEN;
  MAC_CLRT = CLRT_DEF;
  MAC_IPGR = IPGR_DEF;

  /* Enable Reduced MII interface. */
  MAC_COMMAND = CR_RMII | CR_PASS_RUNT_FRM;

  /* Reset Reduced MII Logic. */
  MAC_SUPP = SUPP_RES_RMII;
  for (tout = 100; tout; tout--);
  MAC_SUPP = 0;

  /* Put the DP83848C in reset mode */
  write_PHY (PHY_REG_BMCR, 0x8000);

  /* Wait for hardware reset to end. */
  for (tout = 0; tout < 0x100000; tout++) {
    regv = read_PHY (PHY_REG_BMCR);
    if (!(regv & 0x8800)) {
      /* Reset complete, device not Power Down. */
      break;
    }
  }

  /* Check if this is a DP83848C PHY. */
  id1 = read_PHY (PHY_REG_IDR1);
  id2 = read_PHY (PHY_REG_IDR2);

  if (((id1 << 16) | (id2 & 0xFFF0)) == DP83848C_ID) {
    /* Configure the PHY device */
#if defined (_10MBIT_)
    /* Connect at 10MBit */
    write_PHY (PHY_REG_BMCR, PHY_FULLD_10M);
#elif defined (_100MBIT_)
    /* Connect at 100MBit */
    write_PHY (PHY_REG_BMCR, PHY_FULLD_100M);
#else
    /* Use autonegotiation about the link speed. */
    write_PHY (PHY_REG_BMCR, PHY_AUTO_NEG);
    /* Wait to complete Auto_Negotiation. */
    for (tout = 0; tout < 0x100000; tout++) {
      regv = read_PHY (PHY_REG_BMSR);
      if (regv & 0x0020) {
        /* Autonegotiation Complete. */
        break;
      }
    }
#endif
  }

  /* Check the link status. */
  for (tout = 0; tout < 0x10000; tout++) {
    regv = read_PHY (PHY_REG_STS);
    if (regv & 0x0001) {
      /* Link is on. */
      break;
    }
  }

  /* Configure Full/Half Duplex mode. */
  if (regv & 0x0004) {
    /* Full duplex is enabled. */
    MAC_MAC2    |= MAC2_FULL_DUP;
    MAC_COMMAND |= CR_FULL_DUP;
    MAC_IPGT     = IPGT_FULL_DUP;
  }
  else {
    /* Half duplex mode. */
    MAC_IPGT = IPGT_HALF_DUP;
  }

  /* Configure 100MBit/10MBit mode. */
  if (regv & 0x0002) {
    /* 10MBit mode. */
    MAC_SUPP = 0;
  }
  else {
    /* 100MBit mode. */
    MAC_SUPP = SUPP_SPEED;
  }

  /* Set the Ethernet MAC Address registers */
  MAC_SA0 = ((U32)own_hw_adr[5] << 8) | (U32)own_hw_adr[4];
  MAC_SA1 = ((U32)own_hw_adr[3] << 8) | (U32)own_hw_adr[2];
  MAC_SA2 = ((U32)own_hw_adr[1] << 8) | (U32)own_hw_adr[0];

  /* Initialize Tx and Rx DMA Descriptors */
  rx_descr_init ();
  tx_descr_init ();

  /* Receive Broadcast, Multicast and Perfect Match Packets */
  MAC_RXFILTERCTRL = RFC_MCAST_EN | RFC_BCAST_EN |
RFC_PERFECT_EN;

  /* Enable EMAC interrupts. */
  MAC_INTENABLE = INT_RX_DONE | INT_TX_DONE;
  /* Reset all interrupts */
  MAC_INTCLEAR  = 0xFFFF;

  /* Enable receive and transmit mode of MAC Ethernet core */
  MAC_COMMAND  |= (CR_RX_EN | CR_TX_EN);
  MAC_MAC1     |= MAC1_REC_EN;

  /* Configure VIC for EMAC interrupt. */
  VICVectAddr21 = (U32)interrupt_ethernet;
}
```

init_modem

Summary

```
#include <net_config.h>

void init_modem (void);
```

Description

The **init_modem** function initializes the modem driver. The function:

- sets the default state of the RTS and DTR control lines if they are used
- initializes the driver status variables and buffers.

The **init_modem** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- The TcpNet system calls **init_modem** during its startup.

Return Value

The **init_modem** function does not return any value.

See Also

modem_dial, **modem_hangup**, **modem_listen**

Example

```
void init_modem (void) {
  /* Initializes the modem variables and control signals DTR &
RTS. */
  mlen = 0;
  mem_set (mbuf, 0, sizeof(mbuf));
  wait_for  = 0;
  wait_conn = 0;
  modem_st = MODEM_IDLE;
}
```

init_serial

Summary

```
#include <net_config.h>

void init_serial (void);
```

Description

The **init_serial** function initializes the serial driver. The function:

- sets the baud rate
- initializes the UART
- enables the interrupts for transmission and reception of data.

The **init_serial** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- The TcpNet system calls **init_serial** during its startup.
- You must provide the **init_serial** function if the serial controller you use is different from the ones provided in the TCPnet source.

Return Value

The **init_serial** function does not return any value.

See Also

com_getchar, com_putchar, com_tx_active

Example

```
void init_serial (void) {
  /* Initialize the serial interface */
  rbuf.in    = 0;
  rbuf.out   = 0;
  tbuf.in    = 0;
  tbuf.out   = 0;
  tx_active = __FALSE;

  /* Enable RxD1 and TxD1 pins. */
  PINSEL0 = 0x00050000;
  /* 8-bits, no parity, 1 stop bit */
  U1LCR = 0x83;
  /* 19200 Baud Rate @ 15MHz VPB Clock */
  U1DLL = 49;
  U1DLM = 0;
  U1LCR = 0x03;
  /* Enable RDA and THRE interrupts. */
  U1IER = 0x03;
  /* Enable UART1 interrupts. */
  VICVectAddr14  = (U32)handler_UART1;
  VICVectCntl14  = 0x27;
  VICIntEnable   = 1 << 7;
}
```

init_TcpNet

Summary

```
#include <rtl.h>

void init_TcpNet (void);
```

Description

The **init_TcpNet** function initializes the TCPnet system resources, protocols, and applications.

The **init_TcpNet** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **init_TcpNet** function at system startup to properly initialize the TCPnet environment.

Return Value

The **init_TcpNet** function does not return any value.

See Also

main_TcpNet

Example

```
#include <rtl.h>

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

int_disable_eth

Summary

```
#include <net_config.h>

void int_disable_eth (void);
```

Description

The **int_disable_eth** function disables the ethernet controller interrupts.

The **int_disable_eth** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- You must provide the **int_disable_eth** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- You must provide the **int_disable_eth** function only if you want to use the ethernet driver in interrupt mode.
- The TCPnet system calls **int_disable_eth** to disable the ethernet interrupts when entering critical or non-reentrant sections such as memory management functions.

Return Value

The **int_disable_eth** function does not return any value.

See Also

int_enable_eth, interrupt_ethernet

Example

```
void int_disable_eth (void) {
  /* Ethernet Interrupt Disable function. */
  VICIntEnClr = 1 << 21;
}
```

int_enable_eth

Summary

```
#include <net_config.h>

void int_enable_eth (void);
```

Description

The **int_enable_eth** function enables the ethernet controller interrupts.

The **int_enable_eth** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- You must provide the **int_enable_eth** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- You must provide the **int_enable_eth** function only if you want to use the ethernet driver in interrupt mode.
- The TCPnet system calls **int_enable_eth** to re-enable the ethernet interrupts when exiting critical sections such as memory management functions.

Return Value

The **int_enable_eth** function does not return any value.

See Also

int_disable_eth, interrupt_ethernet

Example

```
void int_enable_eth (void) {
  /* Ethernet Interrupt Enable function. */
  VICIntEnable = 1 << 21;
}
```

interrupt_ethernet

Summary

```
#include <net_config.h>

void interrupt_ethernet (void) __irq
```

Description

The **interrupt_ethernet** function is the ethernet controller interrupt request function that gets called immediately when the ethernet controller receives a new frame. The function first allocates a block of memory. It then reads and copies the packet from the ethernet controller into the allocated memory. The function puts the memory block pointer into the received frames queue by calling the **put_in_queue** function.

The **interrupt_ethernet** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- You must provide the **interrupt_ethernet** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- You must provide the **interrupt_ethernet** function only if you want to use the ethernet controller in interrupt mode.
- When you call **alloc_mem** to allocate the block of memory, ensure that the most significant bit in its argument *size* is set to 1. This is to prevent system hang error when it is out of memory.

Return Value

The **interrupt_ethernet** function does not return any value.

See Also

init_ethernet, send_frame

Example

```
static void interrupt_ethernet (void) __irq {
  /* EMAC Ethernet Controller Interrupt function. */
  OS_FRAME *frame;
  U32 idx,int_stat,RxLen,info;
  U32 *sp,*dp;

  while ((int_stat = (MAC_INTSTATUS & MAC_INTENABLE)) != 0) {
    MAC_INTCLEAR = int_stat;
    if (int_stat & INT_RX_DONE) {
      /* Packet received, check if packet is valid. */
      idx = MAC_RXCONSUMEINDEX;
      while (idx != MAC_RXPRODUCEINDEX) {
        info = Rx_Stat[idx].Info;
        if (!(info & RINFO_LAST_FLAG)) {
          goto rel;
        }

        RxLen = (info & RINFO_SIZE) - 3;
        if (RxLen > ETH_MTU || (info & RINFO_ERR_MASK)) {
          /* Invalid frame, ignore it and free buffer. */
          goto rel;
        }
        /* Flag 0x80000000 to skip sys_error() call when out of
memory. */
        frame = alloc_mem (RxLen | 0x80000000);
        /* if 'alloc_mem()' has failed, ignore this packet. */
        if (frame != NULL) {
          dp = (U32 *)&frame->data[0];
          sp = (U32 *)Rx_Desc[idx].Packet;
          for (RxLen = (RxLen + 3) >> 2; RxLen; RxLen--) {
            *dp++ = *sp++;
          }
          put_in_queue (frame);
        }
rel:    if (++idx == NUM_RX_FRAG) idx = 0;
        /* Release frame from EMAC buffer. */
        MAC_RXCONSUMEINDEX = idx;
      }
    }
    if (int_stat & INT_TX_DONE) {
      /* Frame transmit completed. */
    }
  }
  /* Acknowledge the interrupt. */
  VICVectAddr = 0;
}
```

ioc_getcb

Summary

```
#include <file_config.h>

FAT_VI *ioc_getcb (
    const char *drive);      /* Drive to get control block of */
```

Description

The **ioc_getcb** function retrieves a handle for the FAT Media control interface. It must be called before any other ioc functions are called.

The argument *drive* specifies the drive letter to retrieve the handle for. The following values are allowed for the *drive* argument:

drive	Initialized Drives
"" or **NULL**	Default system drive
"**M:**" or "**M0:**"	Memory Card drive 0
"**M1:**"	Memory Card drive 1
"**U:**" or "**U0:**"	USB Flash drive 0
"**U1:**"	USB Flash drive 1
"**N:**" or "**N0:**"	NAND Flash drive

The **ioc_getcb** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.

Return Value

The **ioc_getcb** function returns a pointer to media control block on success. A NULL return value indicates an error.

See Also

ioc_init, ioc_read_info, ioc_read_sect, ioc_uninit, ioc_write_sect

Example

```
#include <File_Config.h>

FAT_VI *mc0;                                /* Media Control Block */

void main (void) {
   ..
  mc0 = ioc_getcb (NULL);
  if (ioc_init (mc0) == 0) {
    ioc_read_info (&info, mc0);             /* Default drive is M0:
*/
    USBD_MSC_BlockSize  = 512;
    USBD_MSC_BlockGroup = mc0->CaSize + 2; /* Cache buffer from
File_Config.c */
    USBD_MSC_BlockCount = info.block_cnt * (info.read_blen /
512);
    USBD_MSC_MemorySize = USBD_MSC_BlockCount * info.read_blen;
    USBD_MSC_BlockBuf   = (U8 *)mc0->CaBuf;
    usbd_connect(__TRUE);                   /* USB Connect */
    set_cursor (0, 1);
    lcd_print ("  PC Interface  ");         /* PC Interface */
  }
  else {
    LED_On(LED_MSK);                        /* Card Failure! */
    set_cursor (0, 1);
    lcd_print (" Card Failure!  ");
    }
      . .
}
```

ioc_init

Summary

```
#include <file_config.h>

int ioc_init (
    FAT_VI *vi);            /* Pointer to media control block */
```

Description

The **ioc_init** function initializes the FAT Media Device. This function calls the **Init** function of the underlying SPI, MCI or NAND media driver. It must be called before any ioc read or ioc write functions are called.

The argument *vi* specifies the media control block. The value for this argument was retrieved in **ioc_getcb** function.

The **ioc_init** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.

Return Value

The **ioc_init** function returns a value of 0 if successful. A non-zero return value indicates an error.

See Also

ioc_getcb, ioc_read_info, ioc_read_sect, ioc_uninit, ioc_write_sect

Example

```
#include <File_Config.h>

FAT_VI *mc0;                              /* Media Control Block */

int main (void) {
  Media_INFO info;
   ..
  mc0 = ioc_getcb (NULL);                 /* Default drive is M0:
*/
  if (ioc_init (mc0) == 0) {
    ioc_read_info (&info, mc0);
    USBD_MSC_BlockSize  = 512;
    USBD_MSC_BlockCount = info.block_cnt * (info.read_blen /
512);
    USBD_MSC_MemorySize = USBD_MSC_BlockCount * info.read_blen;
    usbd_connect(__TRUE);
    lcd_print ("PC Interface");
  }
  else {
    lcd_print ("Card Failure!");
  }
   ..
}
```

ioc_read_info

Summary

```
#include <file_config.h>

BOOL ioc_read_info (
    Media_INFO *cfg);       /* Structure where to write the read
info */
    FAT_VI *vi);            /* Pointer to media control block */
```

Description

The **ioc_read_info** function reads the media configuration info to a structure. This information is used by host file system to check if the Storage Media is compatible, or for formatting the Storage Media.

The host file system is usually a PC in a configuration, where the embedded system is running USB Device stack, and SD Card is used as a Mass Storage Media for the PC.

The argument *vi* specifies the media control block. The value for this argument was retrieved in **ioc_getcb** function.

The **ioc_read_info** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.

Return Value

The **ioc_read_info** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

ioc_getcb, ioc_init, ioc_read_sect, ioc_uninit, ioc_write_sect

Example

```
#include <File_Config.h>

FAT_VI *mc0;                                /* Media Control Block */

int main (void) {
  Media_INFO info;
   ..
  mc0 = ioc_getcb (NULL);                   /* Default drive is M0:
*/
  if (ioc_init (mc0) == 0) {
    ioc_read_info (&info, mc0);
    USBD_MSC_BlockSize  = 512;
    USBD_MSC_BlockCount = info.block_cnt * (info.read_blen /
512);
    USBD_MSC_MemorySize = USBD_MSC_BlockCount * info.read_blen;
    usbd_connect(__TRUE);
    lcd_print ("PC Interface");
  }
  else {
    lcd_print ("Card Failure!");
  }
   ..
}
```

ioc_read_sect

Summary

```
#include <file_config.h>

BOOL ioc_read_sect (
    U32 sect,           /* Absolute sector address. */
    U8 *buf,            /* Location where to write the read data.
*/
    U32 cnt,            /* Number of sectors to read. */
    FAT_VI *vi);        /* Pointer to media control block */
```

Description

The **ioc_read_sect** function reads reads one or more sectors from the FAT Media Device to a buffer.

The argument *buf* is a pointer to the buffer that stores the data. The argument *sect* specifies the starting sector from where the data are read. The argument *cnt* specifies the number of sectors to be read.

The argument *vi* specifies the media control block. The value for this argument was retrieved in **ioc_getcb** function.

The **ioc_read_sect** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.
- Sector size is 512 bytes.

Return Value

The **ioc_read_sect** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

ioc_getcb, **ioc_init**, **ioc_read_info**, **ioc_uninit**, **ioc_write_sect**

Example

```
#include <File_Config.h>

FAT_VI *mc0;                              /* Media Control Block */

void usbd_msc_read_sect (U32 block, U8 *buf, U32 num_of_blocks) {
  ioc_read_sect (block, buf, num_of_blocks, mc0);
}
```

ioc_uninit

Summary

```
#include <file_config.h>

int ioc_uninit (
    FAT_VI *vi);            /* Pointer to media control block */
```

Description

The **ioc_uninit** function uninitializes the FAT Media Device. This function calls the **UnInit** function of the underlying SPI, MCI or NAND media driver. It must be called before the Media control is returned back to the RL-FlashFS FAT library.

The argument *vi* specifies the media control block. The value for this argument was retrieved in **ioc_getcb** function.

The **ioc_uninit** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.

Return Value

The **ioc_uninit** function returns a value of 0 if successful. A non-zero return value indicates an error.

See Also

ioc_getcb, **ioc_init**, **ioc_read_info**, **ioc_read_sect**, **ioc_write_sect**

Example

```
#include <File_Config.h>

FAT_VI *mc0;                            /* Media Control Block */

int main (void) {
  Media_INFO info;
   ..
  mc0 = ioc_getcb (NULL);               /* Default drive is M0:
*/
  if (ioc_init (mc0) == 0) {
    ioc_read_info (&info, mc0);
    lcd_print ("PC Interface");
     ..
    ioc_uninit (mc0);                   / Stop using SD Card for
Mass Storage. */
  }
  else {
    lcd_print ("Card Failure!");
  }
   ..
}
```

ioc_write_sect

Summary

```
#include <file_config.h>

BOOL ioc_write_sect (
    U32 sect,            /* Absolute sector address. */
    U8 *buf,             /* Pointer to buffer with the data to write
*/
    U32 cnt,             /* Number of sectors to write. */
    FAT_VI *vi);         /* Pointer to media control block */
```

Description

The **ioc_write_sect** function writes one or more sectors from a buffer to the FAT Media Device.

The argument *sect* specifies the starting sector to where data are written. The argument *buf* is a pointer to the buffer that holds the data that should be written. The argument *cnt* specifies the number of sectors to be written.

The argument *vi* specifies the media control block. The value for this argument was retrieved in **ioc_getcb** function.

The **ioc_write_sect** function is in the RL-FlashFS library. The prototype is defined in file_config.h.

Note

- If the **IOC Media** interface is available for FAT media only.
- Sector size is 512 bytes.

Return Value

The **ioc_write_sect** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

ioc_getcb, **ioc_init**, **ioc_read_info**, **ioc_read_sect**, **ioc_uninit**

Example

```
#include <File_Config.h>

FAT_VI *mc0;                                /* Media Control Block */

void usbd_msc_write_sect (U32 block, U8 *buf, U32 num_of_blocks)
{
  ioc_write_sect(block, buf, num_of_blocks, mc0);
}
```

isr_evt_set

Summary

```
#include <rtl.h>

void isr_evt_set (
    U16    event_flags,     /* Bit pattern of event flags to set */
    OS_TID task );           /* The task that the events apply to */
```

Description

The **isr_evt_set** function sets the event flags for the *task* identified by the function argument. The function only sets the event flags whose corresponding bit is set to 1 in the *event_flags* argument.

The **isr_evt_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can call the **isr_evt_set** function only from the IRQ interrupt service routine. You cannot call it from the FIQ interrupt service routine.
- When the **isr_evt_set** function is called too frequently, it forces too many tick timer interrupts and the **task manager** task scheduler is executed most of the time. It might happen that two **isr_evt_set** functions for the same task are called before the task gets a chance to run from one of the event waiting functions (os_evt_wait_). Of course one event is lost because event flags are not counting objects.

Return Value

The **isr_evt_set** function does not return any value.

See Also

os_evt_clr, os_evt_set, os_evt_wait_and, os_evt_wait_or

Example

```
#include <rtl.h>

void timer1 (void) __irq {
   ..
  isr_evt_set (0x0008, tsk1);
   ..
}
```

isr_mbx_check

Summary

```
#include <rtl.h>

OS_RESULT isr_mbx_check (
    OS_ID mailbox );     /* The mailbox to check for free space */
```

Description

The **isr_mbx_check** function determines the number of messages that can still be added into the *mailbox* identified by the function argument. You can avoid losing the message by calling the **isr_mbx_check** function to check for available space in the mailbox before calling the **isr_mbx_send** function to send a message.

The **isr_mbx_check** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can call the **isr_mbx_check** function only from the IRQ interrupt service routine. You cannot call it from the FIQ interrupt service routine.

Return Value

The **isr_mbx_check** function returns the number of message entries in the mailbox that are free.

See Also

isr_mbx_send, os_mbx_declare, os_mbx_init

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

void timer1 (void) __irq {
   ..
  if (isr_mbx_check (mailbox1) != 0) {
    isr_mbx_send (mailbox1, msg);
  }
   ..
}
```

isr_mbx_receive

Summary

```
#include <rtl.h>

OS_RESULT isr_mbx_receive (
    OS_ID  mailbox,       /* The mailbox to put the message in */
    void** message );     /* Location to store the message pointer
*/
```

Description

The **isr_mbx_receive** function gets a pointer to a message from the *mailbox* if the mailbox is not empty. The function puts the message pointer from the mailbox into the location pointed by the *message* argument. The **isr_mbx_receive** function does not cause the current task to sleep even if there is no message in the mailbox. Hence this function can be called from an interrupt function.

You can use the **isr_mbx_receive** function to receive a message or a protocol frame (for example TCP-IP, UDP, and ISDN) in an interrupt function.

The **isr_mbx_receive** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- You can call the **isr_mbx_receive** function only from IRQ interrupt functions. You cannot call it from the FIQ interrupt function.
- When you get the message from the mailbox, you must free the memory block containing the message to avoid running out of memory.
- When you get the message from the mailbox, space is created in the mailbox for a new message.

Return Value

The **isr_mbx_receive** function returns the completion value:

Return Value	Description
OS_R_MBX	A message was available and was read from the mailbox.
OS_R_OK	No message was available in the mailbox.

See Also

isr_mbx_send, os_mbx_declare, os_mbx_init, os_mbx_wait

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);
void *msg;

void EtherInt (void) __irq {
   ..
  if (isr_mbx_receive (mailbox1, &msg) == OS_R_MBX) {
    /* Transmit Ethernet frame. */
  }
  else {
    /* No message was available, stop transmitter */
  }
   ..
}
```

isr_mbx_send

Summary

```
#include <rtl.h>

void isr_mbx_send (
    OS_ID mailbox,          /* The mailbox to put the message in
*/
    void* message_ptr );    /* Pointer to the message */
```

Description

The **isr_mbx_send** function puts the pointer to the message *message_ptr* in the *mailbox* if the mailbox is not already full. The **isr_mbx_send** function does not cause the current task to sleep even if there is no space in the mailbox to put the message.

When an interrupt receives a protocol frame (for example TCP-IP, UDP, or ISDN), you can call the **isr_mbx_send** function from the interrupt to pass the protocol frame as a message to a task.

The **isr_mbx_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- You can call the **isr_mbx_send** function only from IRQ interrupt functions. You cannot call it from FIQ interrupt functions.
- If the *mailbox* is full, the RTX kernel ignores the message since it cannot be put into the mailbox, and calls the **error function**. Thus before sending a message using **isr_mbx_send**, you must use the **isr_mbx_check** function to check if the mailbox is full.

Return Value

The **isr_mbx_send** function does not return any value.

See Also

isr_mbx_check, isr_mbx_receive, os_mbx_declare, os_mbx_init

Example

```
#include <RTL.h>

os_mbx_declare (mailbox1, 20);

void timer1 (void) __irq {
   ..
  isr_mbx_send (mailbox1, msg);
   ..
}
```

isr_sem_send

Summary

```
#include <rtl.h>

void isr_sem_send (
    OS_ID semaphore );    /* The semaphore whose token count is
incremented */
```

Description

The **isr_sem_send** function increments the number of tokens in the *semaphore* object.

The **isr_sem_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.
- You can call the **isr_sem_send** function only from IRQ interrupt functions and not from the FIQ interrupt function.

Return Value

The **isr_sem_send** function does not return any value.

See Also

os_sem_init, os_sem_send, os_sem_wait

Example

```
#include <rtl.h>

OS_SEM semaphore1;

void timer1 (void) __irq {
   ..
  isr_sem_send (semaphore1);
   ..
}
```

isr_tsk_get

Summary

```
#include <rtl.h>

OS_TID isr_tsk_get (void);
```

Description

The **isr_tsk_get** function identifies the interrupted task by returning its task ID.

The **isr_tsk_get** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **isr_tsk_get** function returns the task identifier number (TID) of the interrupted task.

See Also

os_tsk_self

Example

```
#include <rtl.h>

void os_error (U32 err_code) {
  /* This function is called when a runtime error is detected. */
  OS_TID err_task;

  switch (err_code) {
    case OS_ERR_STK_OVF:
      /* Identify the task with stack overflow. */
      err_task = isr_tsk_get();
      break;
    case OS_ERR_FIFO_OVF:
      break;
    case OS_ERR_MBX_OVF:
      break;
  }
  for (;;);
}
```

main_TcpNet

Summary

```
#include <rtl.h>

void main_TcpNet (void);
```

Description

The **main_TcpNet** is the main TcpNet function. It handles:

- protocol timeouts
- ARP address cache
- polling of the ethernet controller for received data.

When **main_TcpNet** receives data from the remote machine, it calls the appropriate TCPnet protocol functions to process the data and then passes the resulting data to the user application.

The **main_TcpNet** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call **main_TcpNet** frequently. Otherwise the TCPnet system fails to run.

Return Value

The **main_TcpNet** function does not return any value.

See Also

init_TcpNet

Example

```
#include <rtl.h>

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

mci.BusMode

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL  (*BusMode)  (
    U32  mode);         /* SD/MMC Bus mode */
   ..
} const MCI_DRV;
```

Description

The **BusMode** function sets the bus mode of the MCI interface to **push-pull** or **open-drain**. The push-pull mode is used for SD and SDHC Memory Cards, while open-drain mode is used for MMC Memory Cards.

The argument *mode* specifies the requested bus mode:

Mode	Description
BUS_OPEN_DRAIN	Open drain bus mode.
BUS_PUSH_PULL	Push-pull bus mode.

The **BusMode** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **BusMode** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusSpeed, mci.BusWidth, mci.CheckMedia, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL BusMode (U32 mode) {
  /* Set MCI Bus mode to Open Drain or Push Pull. */

  switch (mode) {
    case BUS_OPEN_DRAIN:
      MCI_POWER |= 0x40;
      return (__TRUE);

    case BUS_PUSH_PULL:
      MCI_POWER &= ~0x40;
      return (__TRUE);
  }
  return (__FALSE);
}
```

mci.BusSpeed

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*BusSpeed) (
    U32 kbaud);        /* Bus speed in kilo-baud */
   ..
} const MCI_DRV;
```

Description

The **BusSpeed** function sets the transfer speed on the MCI interface to the requested baud rate. When SD/MMC Memory Card is initialized, low speed transfer (400 kBit/s maximum) is used. When the Card initialization is complete, the high speed MCI data transfer is used.

The argument *kbaud* specifies the requested baud rate.

The **BusSpeed** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

- It is important to set the actual MCI speed equal to (or less than) the requested baud rate *kbaud*, but **not higher** than the requested baud rate. The error might happen due to the integer math used for the calculation of a divide factor.

Return Value

The **BusSpeed** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, mci.BusWidth, mci.CheckMedia, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL BusSpeed (U32 kbaud) {
  /* Set a MCI clock speed to desired value. */
  U32 div;

  /* baud = MCLK / (2 x (div + 1)) */
  div = (__MCLK/2000 + kbaud - 1) / kbaud;
  if (div > 0)    div--;
  if (div > 0xFF) div = 0xFF;
  MCI_CLOCK = (MCI_CLOCK & ~0xFF) | 0x300 | div;
  return (__TRUE);
}
```

mci.BusWidth

Summary

```
#include <file_config.h>

typedef struct {
    ..
  BOOL  (*BusWidth)  (
    U32  width);        /* SD/MMC Bus width */
    ..
} const MCI_DRV;
```

Description

The **BusWidth** function sets the bus width of the MCI interface to **1-bit** or **4-bit** data bus. The 4-bit mode is used for SD and SDHC Memory Cards, while 1-bit mode is used for MMC Memory Cards.

The argument *width* specifies the requested bus mode:

Width	Description
1	1-bit bus width mode.
4	4-bit bus width mode.

The **BusWidth** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **BusWidth** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, mci.BusSpeed, mci.CheckMedia, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL BusWidth (U32 width) {
  /* Set MCI Bus width. */

  switch (width) {
    case 1:
      MCI_CLOCK &= ~0x0800;
      return (__TRUE);

    case 4:
      MCI_CLOCK |= 0x0800;
      return (__TRUE);
  }
  return (__FALSE);
}
```

mci.CheckMedia

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32   (*CheckMedia)  (void);           /* Optional, NULL if not
existing */
} const MCI_DRV;
```

Description

The **CheckMedia** is a user-provided routine that checks the SD/MMC Memory Card status. It reads the **Card Detect** (CD) and **Write Protect** (WP) digital inputs. If CD and WP digital inputs from SD Card socket are not connected, this function might be omitted. In this case enter the **NULL** value for CheckMedia into the MCI Driver control block. It is also possible to provide this function, which always returns **M_INSERTED** status.

The **CheckMedia** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **CheckMedia** function returns the or-ed status of the following values:

- **M_INSERTED**
  SD Card is inserted in the socket.
- **M_PROTECTED**
  SD Card is read-only. Lock slider is in position Locked.

See Also

mci.BusMode, mci.BusSpeed, mci.BusWidth, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                           /* Can be NULL if not
existing */
};

static U32 CheckMedia (void) {
  /* Read CardDetect and WriteProtect SD card socket pins. */
  U32 stat = 0;

  if (!(IOPIN0 & 0x04)) {
    /* Card is inserted (CD=0). */
    stat |= M_INSERTED;
  }
  if ((IOPIN0 & 0x20)) {
    /* Write Protect switch is active (WP=1). */
    stat |= M_PROTECTED;
  }
  return (stat);
}
```

mci.Command

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*Command) (
    U8   cmd,              /* Command code for SD Card    */
    U32  arg,              /* Command Argument            */
    U32  resp_type,       /* Command Response type       */
    U32  *rp);             /* Buffer to recive a response */
   ..
} const MCI_DRV;
```

Description

The **Command** function is a user-provided routine that sends a command to the flash memory card. SD/MMC commands are used to control the operation of the flash memory card such as: read block, write block, read card specific data, etc.

Parameter `cmd` is one of available SD/MMC Commands, parameter `arg` is a 32-bit SD Command Argument.

The *resp_type* argument specifies the expected response time:

Type	Description
RESP_NONE	No response expected.
RESP_SHORT	4-byte short response expected.
RESP_LONG	16-byte long response expected.

The argument `rp` is a pointer to a buffer where the received response should be written.

The **Command** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **Command** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, mci.BusSpeed, mci.BusWidth, mci.CheckMedia, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL Command (U8 cmd, U32 arg, U32 resp_type, U32 *rp) {
  /* Send a Command to Flash card and get a Response. */
  U32 cmdval,stat;

  cmd   &= 0x3F;
  cmdval = 0x400 | cmd;
  switch (resp_type) {
    case RESP_SHORT:
      cmdval |= 0x40;
      break;
    case RESP_LONG:
      cmdval |= 0xC0;
      break;
  }
  /* Send the command. */
  MCI_ARGUMENT = arg;
  MCI_COMMAND  = cmdval;

  if (resp_type == RESP_NONE) {
    /* Wait until command finished. */
    while (MCI_STATUS & MCI_CMD_ACTIVE);
    MCI_CLEAR = 0x7FF;
    return (__TRUE);
  }

  for (;;) {
    stat = MCI_STATUS;
    if (stat & MCI_CMD_TIMEOUT) {
      MCI_CLEAR = stat & MCI_CLEAR_MASK;
      return (__FALSE);
    }
    if (stat & MCI_CMD_CRC_FAIL) {
      MCI_CLEAR = stat & MCI_CLEAR_MASK;
      if ((cmd == SEND_OP_COND)       ||
          (cmd == SEND_APP_OP_COND)   ||
          (cmd == STOP_TRANS)) {
        MCI_COMMAND = 0;
        break;
      }
      return (__FALSE);
    }
    if (stat & MCI_CMD_RESP_END) {
      MCI_CLEAR = stat & MCI_CLEAR_MASK;
      break;
    }
  }
  if ((MCI_RESP_CMD & 0x3F) != cmd) {
    if ((cmd != SEND_OP_COND)      &&
        (cmd != SEND_APP_OP_COND)  &&
        (cmd != ALL_SEND_CID)      &&
        (cmd != SEND_CSD))          {
      return (__FALSE);
    }
  }
  /* Read MCI response registers */
  rp[0] = MCI_RESP0;
  if (resp_type == RESP_LONG) {
    rp[1] = MCI_RESP1;
    rp[2] = MCI_RESP2;
    rp[3] = MCI_RESP3;
  }
  return (__TRUE);
}
```

mci.Delay

Summary

```
#include <file_config.h>

typedef struct {
   ..
  void (*Delay) (
    U32 us);            /* Time to wait in micro seconds. */
   ..
} const MCI_DRV;
```

Description

The **Delay** function is a user-provided routine that delays the program execution in the MCI Driver. The argument *us* specifies the delay time in micro-seconds.

The **Delay** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **Delay** function does not return any value.

See Also

mci.BusMode, mci.BusSpeed, mci.BusWidth, mci.CheckMedia, mci.Command, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                         /* Can be NULL if not
existing */
};

static void Delay (U32 us) {
  /* Approximate delay in micro seconds. */
  U32 i;

  for (i = WAIT_CNT(__CPUCLK, us); i; i--);
}
```

mci.Init

Summary

```
#include <file_config.h>

typedef struct {
  BOOL (*Init) (void);
   ..
} const MCI_DRV;
```

Description

The **Init** function is a user-provided routine that initializes the Memory Card Interface controller. It is invoked by the **finit** function on system startup.

The **Init** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

note

- The Flash File System calls the **Init** function at system startup. The FlashFS might call the function again if SD/MMC Flash Memory Card Hot Swapping is used.

Return Value

The **Init** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, **mci.BusSpeed**, **mci.BusWidth**, **mci.CheckMedia**, **mci.Command**, **mci.Delay**, **mci.ReadBlock**, **mci.SetDma**, **mci.UnInit**, **mci.WriteBlock**

Example

```
/* USB-MSC Device Driver Control Block */
FAT_DRV usb0_drv = {
  Init,
  UnInit,
  ReadSector,
  WriteSector,
  ReadInfo,
  CheckMedia
};

static BOOL Init (U32 mode) {
  /* Initialize USB Host. */
  U32 cnt;

  if (mode == DM_IO) {
    /* Initialise USB hardware. */
    media_ok = __FALSE;
    return (usbh_init());
  }

  if (mode == DM_MEDIA) {
    for (cnt = 0; cnt < 1000; cnt++) {
      usbh_engine();
      if (usbh_msc_status () == __TRUE) {
        media_ok = __TRUE;
        return (__TRUE);
      }
      Delay (500);
    }
  }
  return (__FALSE);
}
```

mci.ReadBlock

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*ReadBlock) (
    U32 bl,             /* Absolute block (sector) address.   */
    U8 *buf,            /* Location where to write the data.  */
    U32 cnt);           /* Number of blocks (sectors) to read.*/
   ..
} const MCI_DRV;
```

Description

The **ReadBlock** function is a user-provided routine that reads the data from SD/MMC Memory Card to a buffer. The argument *buf* is a pointer to the buffer that stores the data. The argument *bl* specifies the starting block from where the data are read. The argument *cnt* specifies the number of block to be read.

When the RL-FlashFS library wants to read the data from SD Memory Card, it calls the MCI Driver functions in the following sequence:

1. **SetDma** - sets the DMA to receive data.
   This can be used if DMA must be set prior to sending an SD command. (like for Atmel MCI peripheral). If this function does not exist (a NULL pointer in the MCI Driver control block), then it is not called.
2. **Command** - sends a command READ_BLOCK or READ_MULT_BLOCK to SD Memory Card.
3. **ReadBlock** - reads a block of data from SD memory Card, or simply waits for the DMA transfer to finish.

The **ReadBlock** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **ReadBlock** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, **mci.BusSpeed**, **mci.BusWidth**, **mci.CheckMedia**, **mci.Command**, **mci.Delay**, **mci.Init**, **mci.SetDma**, **mci.UnInit**, **mci.WriteBlock**

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL ReadBlock (U32 bl, U8 *buf, U32 cnt) {
  /* Read one or more 512 byte blocks from Flash Card. */
  U32 i;

  /* Set MCI Transfer registers. */
  MCI_DATA_TMR  = DATA_RD_TOUT_VALUE;
  MCI_DATA_LEN  = cnt * 512;

  /* Start DMA Peripheral to Memory transfer. */
  DmaStart (DMA_READ, buf);
  MCI_DATA_CTRL = 0x9B;

  for (i = DMA_TOUT; i; i--) {
    if (GPDMA_RAW_INT_TCSTAT & 0x01) {
      /* Data transfer finished. */
      return (__TRUE);
    }
  }
  /* DMA Transfer timeout. */
  return (__FALSE);
}
```

mci.SetDma

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*SetDma)  ( /* Optional, NULL for local or non DMA */
    U32 mode,       /* DMA mode read or write             */
    U8  *buf,       /* Buffer location with/for the data */
    U32 cnt);       /* Number of blocks to read or write */
   ..
} const MCI_DRV;
```

Description

The **SetDma** function sets the DMA for sending or receiving data. It can be used to set the DMA mode before the SD Comand is sent to SD Memory Card. Some MCI controllers require to activate the DMA at the same time when the SD Comand is being sent to SD Memory Card.

The argument *mode* specifies the requested DMA mode:

Mode	Description
DMA_READ	Setup DMA for reading from SD Memory Card.
DMA_WRITE	Setup DMA for writing to SD Memory Card.

The argument *buf* is a pointer to the buffer that holds the data that should be written or where the data should be read to. The argument *cnt* specifies the number of block to be written or read.

When the RL-FlashFS library wants to write data to or read data from the SD Memory Card, it calls the MCI Driver functions in the following sequence:

1. **SetDma** - sets the DMA to write or read data.
   This can be used if DMA must be set prior to sending an SD command. (like for Atmel MCI peripheral). If this function does not exist (a NULL pointer in the MCI Driver control block), then it is not called.
2. **Command** - sends a write or read command to SD Memory Card.
3. **WriteBlock** - writes a block of data to SD Memory Card or
   ReadBlock - reads a block of data from SD Memory Card.

The **SetDma** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **SetDma** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, **mci.BusSpeed**, **mci.BusWidth**, **mci.CheckMedia**, **mci.Command**, **mci.Delay**, **mci.Init**, **mci.ReadBlock**, **mci.UnInit**, **mci.WriteBlock**

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  SetDma,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL SetDma (U32 mode, U8 *buf, U32 cnt) {
  /* Configure DMA for read or write. */

  pMCI->MCI_CR    = AT91C_MCI_MCIDIS;
  pMCI->MCI_PTCR  = AT91C_PDC_TXTDIS  | AT91C_PDC_RXTDIS;
  pMCI->MCI_BLKR  = (512 << 16) | cnt;
  if (mode == DMA_READ) {
    /* Transfer data from card to memory. */
    pMCI->MCI_RPR  = (U32)buf;
    pMCI->MCI_RCR  = (512 >> 2) * cnt;
    pMCI->MCI_PTCR = AT91C_PDC_RXTEN;
  }
  else {
    /* Transfer data from memory to card. */
    pMCI->MCI_TPR  = (U32)buf;
    pMCI->MCI_TCR  = (512 >> 2) * cnt;
  }
  return (__TRUE);
}
```

mci.UnInit

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*UnInit) (void);
   ..
} const MCI_DRV;
```

Description

The **UnInit** function is a user-provided routine in the **MCI driver** that uninitializes the MCI controller. It is invoked by the **funinit** function.

It can be used if during the application run time the embedded flash or SD Card drive to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of the drive.

The **UnInit** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **UnInit** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, mci.BusSpeed, mci.BusWidth, mci.CheckMedia, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.WriteBlock

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                         /* Can be NULL if not
existing */
};

static BOOL UnInit (void) {
  /* Reset the MCI peripheral to default state. */

  /* Power down, switch off VCC for the Flash Card. */
  MCI_POWER = 0x00;

  /* Clear all pending interrupts. */
  MCI_COMMAND   = 0;
  MCI_DATA_CTRL = 0;
  MCI_CLEAR     = 0x7FF;

  /* Disable MCI pins. */
  PINSEL1 &= ~0x00003FC0;
  PINSEL4 &= ~0x0FC00000;

  /* Power Down the MCI controller. */
  PCONP   &= ~0x10000000;
  return (__TRUE);
}
```

mci.WriteBlock

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*WriteBlock) (
    U32 bl,           /* Absolute block (sector) address. */
    U8 *buf,          /* Location with the data to write. */
    U32 cnt);         /* Number of blocks to write.       */
   ..
} const MCI_DRV;
```

Description

The **WriteBlock** function is a user-provided routine that writes data blocks to SD/MMC Memory Card. The argument *bl* specifies the starting block to where data are written. The argument *buf* is a pointer to the buffer that holds the data that should be written. The argument *cnt* specifies the number of blocks to be written.

When the RL-FlashFS library wants to write data to SD Memory Card, it calls the MCI Driver functions in the following sequence:

1. **SetDma** - sets the DMA to send data.
   This can be used if DMA must be set prior to sending an SD command. (like for Atmel MCI peripheral). If this function does not exist (a NULL pointer in the MCI Driver control block), then it is not called.
2. **Command** - sends a command WRITE_BLOCK or WRITE_MULT_BLOCK to SD Memory Card.
3. **WriteBlock** - writes a block of data to SD Memory Card, or simply waits for the DMA transfer to finish.

The **WriteBlock** function is in the **MCI driver**. The prototype is defined in file_config.h. You have to customize the function in your own MCI driver.

Return Value

The **WriteBlock** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

mci.BusMode, mci.BusSpeed, mci.BusWidth, mci.CheckMedia, mci.Command, mci.Delay, mci.Init, mci.ReadBlock, mci.SetDma, mci.UnInit

Example

```
/* MCI Device Driver Control Block */
MCI_DRV mci0_drv = {
  Init,
  UnInit,
  Delay,
  BusMode,
  BusWidth,
  BusSpeed,
  Command,
  ReadBlock,
  WriteBlock,
  NULL,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL WriteBlock (U32 bl, U8 *buf, U32 cnt) {
  /* Write a cnt number of 512 byte blocks to Flash Card. */
  U32 i,j;

  for (j = 0; j < cnt; buf += 512, j++) {
    /* Set MCI Transfer registers. */
    MCI_DATA_TMR   = DATA_WR_TOUT_VALUE;
    MCI_DATA_LEN   = 512;

    /* Start DMA Memory to Peripheral transfer. */
    DmaStart (DMA_WRITE, buf);
    MCI_DATA_CTRL = 0x99;

    for (i = DMA_TOUT; i; i--) {
      if (GPDMA_RAW_INT_TCSTAT & 0x01) {
        /* Data transfer finished. */
        break;
      }
    }

    if (i == 0) {
      /* DMA Data Transfer timeout. */
      return (__FALSE);
    }

    if (cnt == 1) {
      break;
    }

    /* Wait until Data Block sent to Card. */
    while (MCI_STATUS != (MCI_DATA_END | MCI_DATA_BLK_END)) {
      if (MCI_STATUS & (MCI_DATA_CRC_FAIL | MCI_DATA_TIMEOUT)) {
        /* Error while Data Block sending occured. */
        return (__FALSE);
      }
    }
    /* Wait 2 SD clocks */
    for (i = WAIT_2SD_CLK(__CPUCLK); i; i--);
  }
  return (__TRUE);
}
```

modem_dial

Summary

```
#include <net_config.h>

void modem_dial (
    U8* dialnum );     /* Pointer to string containing the number
to dial. */
```

Description

The **modem_dial** function dials the target number of the remote modem to connect to the remote host. The function:

- resets and initializes the local modem
- dials the remote modem
- waits for the response "CONNECT"
- ends the connection (hangup) if a timeout occurs.

The argument *dialnum* points to a null terminated ascii string containing the phone number of remote modem.

The **modem_dial** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- The TCPnet system calls **modem_dial** when it wants to initiate a serial modem connection to a remote host.
- The **modem_dial** function sends commands to the local modem to perform the steps needed to dial the remote modem. If the local modem does not respond with the message "OK", the function resends the commands a few times before giving up on trying to get connected.
- The modem driver functions must not use waiting loops because loops block the TCPnet system.

Return Value

The **modem_dial** function does not return any value.

See Also

init_modem, modem_hangup, modem_listen

Example

```
void modem_dial (U8 *dialnum) {
  /* Modem dial target number 'dialnum' */

  dial_num = dialnum;
  listen_mode = 0;
  step = 0;
  proc_dial ();
}
```

modem_hangup

Summary

```
#include <net_config.h>

void modem_hangup (void);
```

Description

The **modem_hangup** function disconnects the modem connection. The function:

- sends an escape sequence to the modem and activates modem command mode
- sends a hangup command to the modem
- waits for further calls if TCPnet is in listening mode.

The **modem_hangup** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- The TcpNet system calls **modem_hangup** when it wants to disconnect the serial modem connection either from client mode or from server mode.
- The **modem_hangup** function sends commands to the local modem to perform the steps needed to disconnect. If the local modem does not respond with the message "OK", the function resends the commands a few times before giving up on trying to get disconnected. If the modem does not respond correctly to the hangup command, the modem driver tries to reset the local modem.
- The modem driver functions must not use waiting loops because loops block the TCPnet system.

Return Value

The **modem_hangup** function does not return any value.

See Also

[init_modem](), [modem_dial](), [modem_listen]()

Example

```
void modem_hangup (void) {
  /* This function clears DTR to force the modem to hang up if
*/
  /* it was on line and/or make the modem to go to command mode.
*/
  step = 0;
  proc_hangup ();
}
```

modem_listen

Summary

```
#include <net_config.h>

void modem_listen (void);
```

Description

The **modem_listen** function puts the local modem in answering mode to accept any incoming call. The function:

- resets and initializes the local modem
- waits for the message "RING"
- accepts the incoming call
- waits for further calls if the line gets disconnected.

The **modem_listen** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- The TCPnet system calls **modem_listen** when the PPP or SLIP network daemon is running in server mode.
- The **modem_listen** function sends commands to the local modem to perform the steps needed to listen for incoming calls. If the local modem does not respond with the message "OK", the function resends the commands a few times before giving up on trying to get into listen mode.
- The modem driver functions must not use waiting loops because loops block the TCPnet system.

Return Value

The **modem_listen** function does not return any value.

See Also

init_modem, modem_dial, modem_hangup

Example

```
void modem_listen (void) {
  /* This function puts Modem into Answering Mode. */
  step = 0;
  listen_mode = 1;
  proc_listen ();
}
```

modem_online

Summary

```
#include <net_config.h>

BOOL modem_online (void);
```

Description

The **modem_online** function checks to see if the local modem is connected (online) to a remote modem.

The **modem_online** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

Return Value

The **modem_online** function returns __TRUE if the local modem is online. Otherwise, it returns __FALSE.

See Also

modem_process, **modem_run**

Example

```
BOOL modem_online (void) {
  /* Checks if the modem is online. Return false when not. */
  if (modem_st == MODEM_ONLINE) {
    return (__TRUE);
  }
  return (__FALSE);
}
```

modem_process

Summary

```
#include <net_config.h>

BOOL modem_process (
    U8 ch );    /* New character sent by the modem. */
```

Description

The **modem_process** function processes characters that the local modem sends to the TCPnet system. The function stores each character in a buffer and checks the buffer for a valid modem response. The argument *ch* is the new character that is available from the modem.

The **modem_process** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- TCPnet calls **modem_process** when a new character is available from the local modem when TCPnet is in modem command mode.
- TCPnet is in modem command mode when it is in the process of instructing the local modem to dial, listen, or disconnect (hangup) from the remote modem.
- The modem driver functions must not use waiting loops because loops block the TCPnet system.

Return Value

The **modem_process** function returns __TRUE when it receives the response "CONNECT" from the local modem. This means that the local modem has connected to the remote modem. Otherwise, it returns __FALSE.

See Also

modem_online, **modem_run**

Example

```
BOOL modem_process (U8 ch) {
  /* Modem character process event handler. This function is
called when */
  /* a new character has been received from the modem in command
mode    */

  if (modem_st == MODEM_IDLE) {
    mlen = 0;
    return (__FALSE);
  }
  if (mlen < sizeof(mbuf)) {
    mbuf[mlen++] = ch;
  }
  /* Modem driver is processing a command */
  if (wait_for) {
    /* 'modem_run()' is waiting for modem reply */
    if (str_scomp (mbuf,reply) == __TRUE) {
      wait_for = 0;
      delay    = 0;
      if (wait_conn) {
        /* OK, we are online here. */
        wait_conn = 0;
        modem_st  = MODEM_ONLINE;
        /* Inform the parent process we are online now. */
        return (__TRUE);
      }
    }
  }
  /* Watch the modem disconnect because we do not use CD line */
  if (mem_comp (mbuf,"NO CARRIER",10) == __TRUE) {
    set_mode ();
  }
  if (ch == '\r' || ch == '\n') {
    flush_buf ();
  }
  /* Modem not connected, return FALSE */
  return (__FALSE);
}
```

modem_run

Summary

```
#include <net_config.h>

void modem_run (void);
```

Description

The **modem_run** function is the main thread that performs the command sending actions needed to dial, listen, or disconnect from the remote modem (when TCPnet and the local modem are in command mode). The **modem_run** function also performs the timeout delays that are necessary when waiting for a response from the local modem and when sending successive commands.

The **modem_run** function for the null modem is in the RL-TCPnet library. The prototype is defined in net_config.h. If you want to use a standard modem connection, you must copy std_modem.c into your project directory.

note

- The TCPnet system calls **modem_run** on every system timer tick interrupt, which occurs at 100 ms intervals by default. Because of this regular interval, the **modem_run** function can implement delays using a simple counter.
- The modem driver functions must not use waiting loops because loops block the TCPnet system. Hence, calling the **modem_run** function every timer tick interval is ideal.
- The functions **modem_dial**, **modem_listen**, and **modem_hangup** only initiate the command sending process with the modem. The **modem_run** function then continues the process until completion.

Return Value

The **modem_run** function does not return any value.

See Also

modem_online, modem_process

Example

```
void modem_run (void) {
  /* This is a main thread for MODEM Control module. It is called
on every */
  /* system timer timer tick to implement delays easy. By default
this is  */
  /* every 100ms. The 'sytem tick' timeout is set in
'Net_Config.c'        */

  if (delay) {
    if (--delay) {
      return;
    }
  }

  switch (modem_st) {
    case MODEM_IDLE:
    case MODEM_ERROR:
      /* Modem idle or in error */
      break;

    case MODEM_ONLINE:
      /* Modem is online - connected */
       break;
    case MODEM_DIAL:
      /* Dial target number */
      proc_dial ();
      break;

    case MODEM_LISTEN:
      /* Activate answering mode */
      proc_listen ();
      break;

    case MODEM_HANGUP:
      /* Hangup and reset the modem */
      proc_hangup ();
      break;
  }
}
```

MSC_Inquiry

Summary

```
#include <mscuser.h>

void MSC_Inquiry (
  void);
```

Description

Sends the vendor ID, product ID, and product revision number to the host.

Return Value

None.

See Also

MSC_MemoryRead, MSC_MemoryVerify, MSC_MemoryWrite

Example

```
#include <mscuser.h>

void MSC_Inquiry (void)  {
…

  BulkBuf [4] = "K";      // Vendor ID = Keil. Modify it for your
product.
  BulkBuf [4] = "e";
  BulkBuf [4] = "i";
  BulkBuf [4] = "l";
  BulkBuf [4] = " ";
  BulkBuf [4] = " ";
  BulkBuf [4] = " ";
  BulkBuf [4] = " ";
…
  DataInTransfer ();      // Send the BulkBuf buffer to the host.
}
```

MSC_MemoryRead

Summary

```
#include <mscuser.h>

void MSC_MemoryRead  (
  void);
```

Description

The function **MSC_MemoryRead** sends data from the onboard RAM to the host. The number of bytes it sends is either the maximum packet size or the number of bytes requested by the host, whichever is lower. Modify this function to suit the application requirements.

The function **MSC_MemoryRead** is part of the USB Function Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

MSC_MemoryVerify, MSC_MemoryWrite

Example

```
#include <mscuser.h>

void MSC_MemoryRead (void)  {
U32 n;


  …
  // Call your function to read from the flash card.
  Flash_MemoryRead(Offset, n, &Memory);

  // Send the data through the endpoint.
  USB_WriteEP(MSC_EP_IN, &Memory, n);
  …
}
```

MSC_MemoryVerify

Summary

```
#include <mscuser.h>

void MSC_MemoryVerify (
  void);
```

Description

The function **MSC_MemoryVerify** checks whether the data written to the onboard RAM is identical to the data sent by the host. The number of bytes it checks is the number of bytes sent by the host. Modify this function to suit the application requirements.

The function **MSC_MemoryVerify** is part of the USB Function Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

MSC_MemoryRead, MSC_MemoryWrite

Example

```
#include <mscuser.h>

void MSC_MemoryVerify (void)  {
U32 n;
  …
  // Call your function to read from the flash card.
  Flash_MemoryRead(Offset, BulkLen, &Memory);

  for (n=0; n < BulkLen; n++)  {
    if (Memory [n] != BulkBuf [n])  {
      MemOK = __FALSE;
      break;
    }
  }
  …
}
```

MSC_MemoryWrite

Summary

```
#include <mscuser.h>

void MSC_MemoryWrite (
  void);
```

Description

The **MSC_MemoryWrite** function copies data from the host to the onboard RAM. Modify this function to suit the application requirements. The **MSC_MemoryWrite** function is part of the USB Function Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

MSC_MemoryRead, **MSC_MemoryVerify**

Example

```
#include <mscuser.h>

void MSC_MemoryWrite (void)  {
U32 n;


…
  // Copy the buffer BulkBuf to the flash card.
  Flash_MemoryWrite (Offset, BulkLen, &BulkBuf);
…
}
```

nand.BlockErase

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32  (*BlockErase)  (
    U32  row,              /* Block starting address */
    NAND_DRV_CFG *cfg);   /* Device configuration    */
} const NAND_DRV;
```

Description

The **BlockErase** function is a user-provided routine that erases the NAND flash block. The argument *row* specifies the starting address of flash block to be erased.

The *cfg* argument specifies the device configuration for the NAND driver. This configuration contains also device specific default data positions as described in Page data Layout.

The **BlockErase** function is in the **NAND driver**. The prototype is defined in file_config.h. You have to customize the function in your own NAND driver.

Return Value

The **BlockErase** function returns the following values:

- **RTV_NOERR**
  Flash block erased successfully.
- **ERR_NAND_HW_TOUT**
  Hardware data transfer timeout.
- **ERR_NAND_ERASE**
  Block erase has failed.

See Also

nand.Init, **nand.PageRead**, **nand.PageWrite**, **nand.UnInit**

Example

```
/* NAND Device Driver Control Block */
NAND_DRV nand0_drv = {
  Init,
  UnInit,
  PageRead,
  PageWrite,
  BlockErase
};

static U32 BlockErase (U32 row, NAND_DRV_CFG *cfg) {

  if (!StatusFlag (NAND_CON_READY)) {        /* Wait for
controller ready    */
    return ERR_NAND_HW_TOUT;
  }
  MLC_CMD   = NAND_CMD_ERASE1ST;             /* Erase command 1
         */
  SetAddr (row, cfg->AddrCycles);            /* Set address
         */
  MLC_CMD   = NAND_CMD_ERASE2ND;             /* Erase command 2
         */
  if (!StatusFlag (NAND_CON_READY)) {        /* Wait for
controller ready    */
    return ERR_NAND_HW_TOUT;
  }

  if (!StatusFlag (NAND_CHIP_BUSY)) {        /* Wait while NAND
busy          */
    return ERR_NAND_HW_TOUT;
  }

  MLC_CMD = NAND_CMD_STATUS;                 /* Write Read Status
command    */
  if ((U8)MLC_DATAX(0) & NAND_STAT_FAIL) {   /* Check if command
successful   */
    return ERR_NAND_ERASE;                   /* Block Erase Failed
         */
  }
  return RTV_NOERR;
}
```

nand.Init

Summary

```
#include <file_config.h>

typedef struct {
  U32  (*Init) (
    NAND_DRV_CFG *cfg);  /* Device configuration */
   ..
} const NAND_DRV;
```

Description

The **Init** function is a user-provided routine that initializes the Flash programming algorithm for a NAND flash memory device. It is invoked by the **finit** function on system startup.

The *cfg* argument specifies the device configuration for the NAND driver. This configuration contains also device specific default data positions as described in Page data Layout. The **Init** function can overwrite those data positions, if they are different from default.

The **Init** function is in the **NAND driver**. The prototype is defined in file_config.h. You have to customize the function in your own NAND driver.

Return Value

The **Init** function returns the following values:

- **RTV_NOERR**
  NAND Flash initialized successfully.
- **ERR_NAND_HW_TOUT**
  NAND Flash Reset Command failed.
- **ERR_NAND_UNSUPPORTED**
  NAND Flash Page size invalid.

See Also

nand.BlockErase, **nand.PageRead**, **nand.PageWrite**, **nand.UnInit**

Example

```
/* NAND Device Driver Control Block */
NAND_DRV nand0_drv = {
  Init,
  UnInit,
  PageRead,
  PageWrite,
  BlockErase
};

static U32 Init (NAND_DRV_CFG *cfg) {
  U32 pgSz, cyc;

  pgSz = cfg->PageSize;
  switch (pgSz) {
    case 528:
    case 2112:
      break;
    default:
      return ERR_NAND_UNSUPPORTED;
  }

  FLASHCLK_CTRL = 0x00000022;               /* Setup NAND Flash
Clock Control */
  MLC_CEH       = 0;                          /* Force nCE assert
         */
  MLC_CMD       = NAND_CMD_RESET;             /* Reset NAND Flash
         */

  if (!StatusFlag (NAND_CHIP_BUSY)) {         /* Wait while NAND
busy           */
    return ERR_NAND_HW_TOUT;
  }

  cyc = cfg->AddrCycles;                      /* Get address cycles
         */

  MLC_LOCK_PR = 0xA25E;                       /* Unlock MLC_ICR
register        */
  MLC_ICR     = ((cyc  ==    4) << 1) |
                ((pgSz == 2112) << 2);

  MLC_LOCK_PR  = 0xA25E;                      /* Unlock MLC_TIME
register       */
  MLC_TIME_REG = (3 << 24) | (11 << 19) | (4 << 16) | (2 << 12) |
                 (4 << 8)  | ( 3 <<  4) | (4 <<  0);

  P3_OUTP_SET |= (1 << 19);                     /* Disable write
protect        */
  return RTV_NOERR;
}
```

nand.PageRead

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32 (*PageRead) (
    U32 row,              /* Page address           */
    U8 *buf,              /* Pointer to data buffer */
    NAND_DRV_CFG *cfg);   /* Device configuration   */
   ..
} const NAND_DRV;
```

Description

The **ReadPage** function is a user-provided routine that reads the data from NAND Flash device to a buffer. The argument *buf* is a pointer to the buffer that receives the read data. The argument *row* specifies the starting page from where the data are read.

The *cfg* argument specifies the device configuration for the NAND driver. This configuration contains also device specific default data positions as described in Page data Layout.

The **PageRead** function is in the **NAND driver**. The prototype is defined in file_config.h. You have to customize the function in your own NAND driver.

Return Value

The **PageRead** function returns the following values:

- **RTV_NOERR**
  Page read successfully.
- **ERR_NAND_HW_TOUT**
  Hardware data transfer timeout.
- **ERR_ECC_COR**
  ECC corrected the error in page data. Read page data is valid.
- **ERR_ECC_UNCOR**
  ECC was not able to correct the error in page data. Data read contains errors.

See Also

nand.BlockErase, **nand.Init**, **nand.PageWrite**, **nand.UnInit**

Example

```
/* NAND Device Driver Control Block */
NAND_DRV nand0_drv = {
  Init,
  UnInit,
  PageRead,
  PageWrite,
  BlockErase
};

static U32 PageRead (U32 row, U8 *buf, NAND_DRV_CFG *cfg) {
  U32 i, sec, ecc;
  U32 *p = (U32 *)buf;

  MLC_CMD = NAND_CMD_READ1ST;                /* Read command (1st
cycle)      */
  if (cfg->PageSize > 528) {
    MLC_CMD = NAND_CMD_READ2ND;              /* Read command (2nd
cycle)     */
  }
  SetAddr (row << 8, cfg->AddrCycles);      /* Set address
         */

  ecc = ECC_NOERR;

  for (sec = 0; sec < cfg->SectorsPerPage; sec++) {
    MLC_ECC_AUTO_DEC_REG = 0x00;            /* Auto Decode
         */
    if (!StatusFlag (NAND_CON_READY)) {     /* Wait for
controller ready    */
      return ERR_NAND_HW_TOUT;
    }
    if (!StatusFlag (NAND_ECC_READY)) {     /* Wait for ECC ready
         */
      return ERR_NAND_HW_TOUT;
    }

    if (MLC_ISR & NAND_ERR_DET) {           /* Check for decode
error       */
      ecc |= ECC_CORRECTED;
    }
    if (MLC_ISR & NAND_DEC_FAIL) {
      ecc |= ECC_UNCORRECTED;
    }
    for (i = 0; i < (528 >> 2); i++) {
      *p++ = MLC_BUFFX (i);                 /* Read main + spare
area       */
    }
  }
  if (ecc & ECC_UNCORRECTED) {
    /* ECC was not able to correct the data within page */
    return ERR_ECC_UNCOR;
  }
  if (ecc & ECC_CORRECTED) {
    /* ECC corrected the data within page*/
    return ERR_ECC_COR;
  }
  return RTV_NOERR;
}
```

nand.PageWrite

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32  (*PageWrite)  (
    U32 row,              /* Page address           */
    U8 *buf,              /* Pointer to data buffer */
    NAND_DRV_CFG *cfg);   /* Device configuration   */
   ..
} const NAND_DRV;
```

Description

The **PageWrite** function is a user-provided routine that writes the data from buffer to a NAND Flash device. The argument *buf* is a pointer to the buffer that contains the data. The argument *row* specifies the starting page to where the data are written.

The *cfg* argument specifies the device configuration for the NAND driver. This configuration contains also device specific default data positions as described in Page data Layout.

The **PageWrite** function is in the **NAND driver**. The prototype is defined in file_config.h. You have to customize the function in your own NAND driver.

Return Value

The **PageWrite** function returns the following values:

- **RTV_NOERR**
  Page written successfully.
- **ERR_NAND_HW_TOUT**
  Hardware data transfer timeout.
- **ERR_NAND_PROG**
  Page write has failed.

See Also

nand.BlockErase, **nand.Init**, **nand.PageRead**, **nand.UnInit**

Example

```
/* NAND Device Driver Control Block */
NAND_DRV nand0_drv = {
  Init,
  UnInit,
  PageRead,
  PageWrite,
  BlockErase
};

static U32 PageWrite (U32 row, U8 *buf, NAND_DRV_CFG *cfg) {
  U32 i, sec;

  U32 *p = (U32 *)buf;

  if (!StatusFlag (NAND_CON_READY)) {        /* Wait for
controller ready    */
    return ERR_NAND_HW_TOUT;
  }
  MLC_CMD = NAND_CMD_PROG1ST;               /* Programm command 1
         */
  SetAddr (row << 8, cfg->AddrCycles);      /* Set address
         */

  for (sec = 0; sec < cfg->SectorsPerPage; sec++) {
    MLC_ECC_ENC_REG = 0;                    /* Start Encode Cycle
         */
    for (i = 0; i < (528 >> 2); i++)        /* Write main + spare
area      */
      MLC_BUFFX(i) = *p++;
    MLC_ECC_AUTO_ENC_REG = 0;               /* Auto encode
         */
    if (!StatusFlag (NAND_CON_READY)) {     /* Wait for
controller ready    */
      return ERR_NAND_HW_TOUT;
    }
  }
  MLC_CMD = NAND_CMD_PROG2ND;               /* Programm command 2
         */

  if (!StatusFlag (NAND_CON_READY)) {       /* Wait for
controller ready    */
    return ERR_NAND_HW_TOUT;
  }
  if (!StatusFlag (NAND_CHIP_BUSY)) {       /* Wait while NAND
busy          */
    return ERR_NAND_HW_TOUT;
  }

  MLC_CMD = NAND_CMD_STATUS;                /* Send status
command          */
  if ((U8)MLC_DATAX(0) & NAND_STAT_FAIL) {
    return ERR_NAND_PROG;                   /* Programming failed
         */
  }
  return RTV_NOERR;
}
```

nand.UnInit

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32   (*UnInit)  (
    NAND_DRV_CFG *cfg);   /* Device configuration */
   ..
} const NAND_DRV;
```

Description

The **UnInit** function is a user-provided routine that uninitializes the Flash programming algorithm for a NAND flash memory device. It is invoked by the **funinit** function.

The *cfg* argument specifies the device configuration for the NAND driver. This configuration contains also device specific default data positions as described in Page data Layout.

It can be used if during the application run time the embedded flash needs to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of embedded flash device.

The **UnInit** function is in the **NAND driver**. The prototype is defined in file_config.h. You have to customize the function in your own NAND driver.

Return Value

The **UnInit** function returns the following values:

- **RTV_NOERR**
  NAND Flash uninitialized successfully.

See Also

nand.BlockErase, **nand.Init**, **nand.PageRead**, **nand.PageWrite**

Example

```
/* NAND Device Driver Control Block */
NAND_DRV nand0_drv = {
  Init,
  UnInit,
  PageRead,
  PageWrite,
  BlockErase
};

static U32 UnInit (NAND_DRV_CFG *cfg) {
  return RTV_NOERR;
}
```

os_dly_wait

Summary

```
#include <rtl.h>

void os_dly_wait (
    U16 delay_time );     /* Length of time to pause */
```

Description

The **os_dly_wait** function pauses the calling task. The argument *delay_time* specifies the length of the pause and is measured in number of system_ticks. You can set the *delay_time* to any value between 1 and 0xFFFE.

The **os_dly_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You cannot intermix with a single task the wait method **os_itv_wait ()** and **os_dly_wait ()**.

Return Value

The **os_dly_wait** function does not return any value.

See Also

os_itv_set, os_itv_wait

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_dly_wait (20);
   ..
}
```

os_evt_clr

Summary

```
#include <rtl.h>

void os_evt_clr (
    U16    clear_flags,    /* Bit pattern of event flags to clear
*/
    OS_TID task );          /* The task that the events apply to */
```

Description

The **os_evt_clr** function clears the event flags for the *task* identified by the function argument. The function only clears the event flags whose corresponding bit is set to 1 in the *clear_flags* argument.

The **os_evt_clr** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_evt_clr** function does not return any value.

See Also

isr_evt_set, **os_evt_set**, **os_evt_wait_and**, **os_evt_wait_or**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_evt_clr (0x0002, tsk2);
   ..
}
```

os_evt_get

Summary

```
#include <rtl.h>

U16 os_evt_get (void);
```

Description

You can use the **os_evt_get** function to identify the event that caused the **os_evt_wait_or** function to complete.

The **os_evt_get** function identifies this event by setting the corresponding flag in the returned value. If more than one event occurred simultaneously, all their flags are set in the returned value.

The **os_evt_get** function is in the RL-RTX library. The prototype is defined in rtl.h.

- When the **os_evt_wait_or** function has been waiting on more than one event, it is not immediately known which event caused the **os_evt_wait_or** function to return. This is why the **os_evt_get** function is useful.

Return Value

The **os_evt_get** function returns a bit pattern that identifies the events that caused the **os_evt_wait_or** function to complete.

See Also

os_evt_wait_or

Example

```
#include <RTL.h>

__task void task1 (void) {
  U16 ret_flags;

  if (os_evt_wait_or (0x0003, 500) == OS_R_EVT) {
    ret_flags = os_evt_get ();
    printf("Events %04x received.\n",ret_flags);
  }
   ..
}
```

os_evt_set

Summary

```
#include <rtl.h>

void os_evt_set (
    U16    event_flags,     /* Bit pattern of event flags to set */
    OS_TID task );           /* The task that the events apply to */
```

Description

The **os_evt_set** function sets the event flags for the *task* identified by the function argument. The function only sets the event flags whose corresponding bit is set to 1 in the *event_flags* argument.

The **os_evt_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_evt_set** function does not return any value.

See Also

isr_evt_set, **os_evt_clr**, **os_evt_wait_and**, **os_evt_wait_or**

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_evt_set (0x0003, tsk2);
   ..
}
```

os_evt_wait_and

Summary

```
#include <rtl.h>

OS_RESULT os_evt_wait_and (
    U16 wait_flags,     /* Bit pattern of events to wait for */
    U16 timeout );      /* Length of time to wait for event */
```

Description

The **os_evt_wait_and** function waits for all the events specified in the *wait_flags* to occur. The function only waits on events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The function can wait on as many as 16 different events.

You can use the *timeout* argument to specific the length of time after which the function must return even if none of the events have occurred. You can use any value of timeout with the exception of 0xFFFF, which you can use to specify an indefinite timeout. The unit of measure of the *timeout* argument is the number of system intervals.

The **os_evt_wait_and** function returns when all of the events specified in the *wait_flags* have occurred or when the timeout expires. If all events specified in *wait_flags* have arrived, this function clears them before the function returns. The function actually clears the events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The other event flags are not changed.

The **os_evt_wait_and** function is in the RL-RTX library. The prototype is defined in rtl.h.

note

- Each task has its own 16 bit wait flag.

Return Value

The **os_evt_wait_and** function returns a value to indicate whether an event occurred or the timeout expired.

Return Value	Description
OS_R_EVT	All the flags specified by **wait_flags** have been set.
OS_R_TMO	The timeout has expired.

See Also

os_evt_get, os_evt_wait_or

Example

```
#include <rtl.h>

__task void task1 (void) {
  OS_RESULT result;

  result = os_evt_wait_and (0x0003, 500);
  if (result == OS_R_TMO) {
    printf("Event wait timeout.\n");
  }
  else {
    printf("Event received.\n");
  }
   ..
}
```

os_evt_wait_or

Summary

```
#include <rtl.h>

OS_RESULT os_evt_wait_or (
    U16 wait_flags,     /* Bit pattern of events to wait for */
    U16 timeout );      /* Length of time to wait for event */
```

Description

The **os_evt_wait_or** function waits for one of the events specified in the *wait_flags* to occur. The function only waits on events whose corresponding flags have been set to 1 in the *wait_flags* parameter. The function can wait on as many as 16 different events.

You can use the *timeout* argument to specific the length of time after which the function must return even if none of the events have occurred. You can use any value of timeout with the exception of 0xFFFF, which you can use to specify an indefinite timeout. The unit of measure of the *timeout* argument is the number of system intervals.

The **os_evt_wait_or** function returns when at least one of the events specified in the *wait_flags* has occurred or when the timeout expires. The event flag or flags that caused the **os_evt_wait_or** function to complete are cleared before the function returns. You can identify those event flags with **os_evt_get** function later.

The **os_evt_wait_or** function is in the RL-RTX library. The prototype is defined in rtl.h.

note

- Each task has its own 16 bit wait flag.

Return Value

The **os_evt_wait_or** function returns a value to indicate whether an event occurred or the timeout expired.

Return Value	Description
OS_R_EVT	At least one of the flags specified by **wait_flags** has been set.
OS_R_TMO	The timeout has expired.

See Also

os_evt_get, os_evt_wait_and

Example

```
#include <rtl.h>

__task void task1 (void) {
  OS_RESULT result;

  result = os_evt_wait_or (0x0003, 500);
  if (result == OS_R_TMO) {
    printf("Event wait timeout.\n");
  }
  else {
    printf("Event received.\n");
  }
   ..
}
```

os_itv_set

Summary

```
#include <rtl.h>

void os_itv_set (
    U16 interval_time );     /* Time interval for periodic wake-up
*/
```

Description

The **os_itv_set** function sets up the calling task for periodic wake-up after a time interval specified by *interval_time*. You must use the **os_itv_wait** function to wait for the completion of the time interval. The time interval is measured in units of system ticks, and you can set it to any value between 1 and 0xFFFE.

The **os_itv_set** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_itv_set** function does not return any value.

See Also

os_dly_wait, os_itv_wait

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_itv_set (50);
   ..
}
```

os_itv_wait

Summary

```
#include <rtl.h>

void os_itv_wait (void);
```

Description

The **os_itv_wait** function waits for a periodic time interval after which the RTX kernel wakes up the calling task. You must set the time interval using the **os_itv_set** function.

you can use the **os_itv_wait** function to perform a job at regular intervals independent of the execution time of the task.

The **os_itv_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You cannot intermix with a single task the wait method **os_itv_wait ()** and **os_dly_wait ()**.

Return Value

The **os_itv_wait** function does not return any value.

See Also

os_dly_wait, os_itv_set

Example

```
#include <rtl.h>

__task void task1 (void) {
   ..
  os_itv_set (20);
  for (;;) {
    os_itv_wait ();
    /* do some actions at regular time intervals */
  }
}
```

os_mbx_check

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_check (
    OS_ID mailbox );     /* The mailbox to check for free space */
```

Description

The **os_mbx_check** function determines the number of messages that can still be added into the *mailbox* identified by the function argument. You can avoid blocking the current task by calling the **os_mbx_check** function to check for available space in the mailbox before calling the **os_mbx_send** function to send a message.

The **os_mbx_check** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_mbx_check** function returns the number of message entries in the mailbox that are free.

See Also

isr_mbx_check, **os_mbx_declare**, **os_mbx_send**

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  if (os_mbx_check (mailbox1) == 0) {
    printf("Mailbox is full.\n");
  }
   ..
}
```

os_mbx_declare

Summary

```
#include <rtl.h>

#define os_mbx_declare( \
    name,               \ /* Name of the mailbox */
    cnt )               \ /* Number of message entries */
U32 name [4 + cnt]
```

Description

The **os_mbx_declare** macro defines a mailbox object. The argument *name* is the name of the mailbox object. The argument *cnt* is the number of messages that can be entered into the mailbox object. A cnt value of 20 is sufficient in most cases.

The **os_mbx_declare** macro is part of RL-RTX. The definition is in rtl.h.

Return Value

The **os_mbx_declare** macro does not return any value.

See Also

os_mbx_init

Example

```
#include <rtl.h>

/* Declare a mailbox for 20 messages. */
os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  os_mbx_init (mailbox1, sizeof(mailbox1));
   ..
}
```

os_mbx_init

Summary

```
#include <rtl.h>

void os_mbx_init (
    OS_ID mailbox,        /* The mailbox to initialize */
    U16   mbx_size );     /* Number of bytes in the mailbox */
```

Description

The **os_mbx_init** function initializes the *mailbox* object identified by the function argument.

The argument *mbx_size* specifies the size of the mailbox, in bytes. However, the number of message entries in the mailbox is defined by the **os_mbx_declare** macro.

The **os_mbx_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox before you perform any operation on it.

Return Value

The **os_mbx_init** function does not return any value.

See Also

os_mbx_declare

Example

```
#include <rtl.h>

/* Declare a mailbox for 20 messages. */
os_mbx_declare (mailbox1, 20);

__task void task1 (void) {
   ..
  os_mbx_init (mailbox1, sizeof(mailbox1));
   ..
}
```

os_mbx_send

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_send (
    OS_ID mailbox,          /* The mailbox to put the message in */
    void* message_ptr,     /* Pointer to the message */
    U16   timeout );        /* Wait time for mailbox to be free */
```

Description

The **os_mbx_send** function puts the pointer to a message, *message_ptr*, in the *mailbox*, if the mailbox is not already full.

If the mailbox is full, the RTX kernel puts the calling task to sleep. The *timeout* specifies the length of time the task can wait for a space to become available in the mailbox. The kernel wakes up the task either when the timeout has expired or when a space becomes available in the mailbox.

You can set the timeout to any value between 0 and 0xFFFE. You can set the timeout to 0xFFFF for an indefinite timeout.

The **os_mbx_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- The unit of measure of the *timeout* argument is numbers of system intervals.
- The *message_ptr* points to a block of allocated memory holding a message of any type. The block of memory is allocated when the message is created and freed by the destination task when the message is received.

Return Value

The function returns the completion value:

Return Value	Description
OS_T_TMO	The timeout has expired.
OS_R_OK	The message has been put in the mailbox.

See Also

isr_mbx_send, os_mbx_declare, os_mbx_init, os_mbx_wait

Example

```
#include <RTL.h>

os_mbx_declare (mailbox1, 20);
OS_TID tsk1, tsk2;

__task void task1 (void);
__task void task2 (void);

__task void task1 (void) {
  void *msg;
   ..
  tsk2 = os_tsk_create (task2, 0);
  os_mbx_init (mailbox1, sizeof(mailbox1));
  msg = alloc();
  /* set message content here*/
  os_mbx_send (mailbox1, msg, 0xFFFF);
   ..
}

__task void task2 (void) {
  void *msg;
   ..
  os_mbx_wait (mailbox1, &msg, 0xffff);
  /* process message content here */
  free (msg);
   ..
}
```

os_mbx_wait

Summary

```
#include <rtl.h>

OS_RESULT os_mbx_wait (
    OS_ID   mailbox,       /* The mailbox to get message from */
    void** message,        /* Location to store the message pointer
*/
    U16     timeout );     /* Wait time for message to become
available */
```

Description

The **os_mbx_wait** function gets a pointer to a message from the *mailbox* if the mailbox is not empty. The function puts the message pointer from the mailbox into the location pointed by the *message* argument.

If the mailbox is empty, the RTX kernel puts the calling task to sleep. The *timeout* specifies the length of time the task can wait for a message. The kernel wakes up the task either when the timeout expires or when a message becomes available in the mailbox.

You can set the timeout to any value between 0 and 0xFFFE. You can set the timeout to 0xFFFF for an indefinite timeout. If you specify the *timeout* to 0, the calling task continues immediately even if there are higher priority tasks in the ready list, irrespective of whether a message is present in the mailbox or not.

The **os_mbx_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must declare and initialize the mailbox object before you perform any operation on it.
- The unit of measure of the *timeout* argument is numbers of system intervals.
- When you get the message from the mailbox, you must free the memory block containing the message to avoid running out of memory.
- When you get the message from the mailbox, space is created in the mailbox for a new message.

Return Value

The **os_mbx_wait** function returns a completion value:

Return Value	Description
OS_R_MBX	The task has waited until a message was put in the mailbox.
OS_R_TMO	The timeout specified by timeout has expired before a message was available in the mailbox.
OS_R_OK	A message was available in the mailbox, and the task continues without waiting.

See Also

isr_mbx_receive, os_mbx_declare, os_mbx_init, os_mbx_send

Example

```
#include <rtl.h>

os_mbx_declare (mailbox1, 20);

__task void task1 (void){
  void *msg;
   ..
  if (os_mbx_wait (mailbox1, &msg, 10) == OS_R_TMO) {
    printf ("Wait message timeout!\n");
    }
  else {
    /* process message here */
    free (msg);
  }
   ..
}
```

os_mut_init

Summary

```
#include <rtl.h>

void os_mut_init (
    OS_ID mutex );     /* The mutex to initialize */
```

Description

The **os_mut_init** function initializes the *mutex* object identified by the function argument.

The **os_mut_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must define the mutex object of type OS_MUT. You can use the name of the mutex object to identify it during operation.

Return Value

The **os_mut_init** function does not return any value.

See Also

os_mut_release, **os_mut_wait**

Example

```
#include <rtl.h>

OS_MUT mutex1;

__task void task1 (void) {
   ..
  os_mut_init (mutex1);
   ..
}
```

os_mut_release

Summary

```
#include <rtl.h>

OS_RESULT os_mut_release (
    OS_ID mutex );     /* The mutex to release */
```

Description

The **os_mut_release** function decrements the internal counter of the *mutex* identified by the function argument in order to release the mutex. Only when the internal counter of the mutex is zero, the mutex is really free to be acquired by another task.

The mutex object knows the task that currently owns it. Hence the owning task can acquire and lock the mutex as many times as needed using the **os_mut_wait** function. When a task that owns a mutex tries to acquire it again, the task does not get blocked, but the mutex's internal counter is incremented. The task that acquired the mutex must release the mutex as many times as it was acquired, so that the internal counter of the mutex is decremented to 0.

This function also restores the original task's priority if **priority inheritance** has been applied to the owning task of the mutex and his priority has been temporary raised.

The **os_mut_release** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the mutex object using the **os_mut_init** function before you can perform any operation on it.

Return Value

The **os_mut_release** function returns the completion value:

Return Value	Description
OS_R_OK	The mutex was successfully released
OS_R_NOK	An error occurred. This can be either because the internal counter of the mutex is not 0 or because the calling task is not the owner of the mutex.

See Also

os_mut_init, os_mut_wait

Example

```
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (mutex1);
}

void f2 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 2 */
   ..
  os_mut_release (mutex1);
}

__task void task1 (void) {
   ..
  os_mut_init (mutex1);
  f1 ();
   ..
}

__task void task2 (void) {
   ..
  f2 ();
   ..
}
```

os_mut_wait

Summary

```
#include <rtl.h>

OS_RESULT os_mut_wait (
    OS_ID mutex,         /* The mutex to acquire */
    U16   timeout );     /* Length of time to wait */
```

Description

The **os_mut_wait** function tries to acquire the *mutex* identified by the function argument. If the mutex has not be locked by another task, the calling task acquires and locks the mutex and might continue immediately. If the mutex has been locked by another task, then the RTX kernel puts the calling task to sleep until the mutex becomes unlocked or until the *timeout* expires.

You can specify any value between 0 and 0xFFFE for the *timeout* argument. You must set *timeout* to 0xFFFF for an indefinite timeout period.

If you specify a value of 0 for the *timeout*, the calling task continues immediately even if there is a higher priority task in the ready list.

This function also raises the priority of the owning task of the mutex, if it is lower than the priority of the calling task. This programming method is called **priority inheritance** and is used to eliminate priority inversion problems. When a mutex is released, the original priority of the owning task will be restored.

The **os_mut_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Timeout is measured in number of system intervals.
- You must initialize the mutex object using the **os_mut_init** function before you can perform any operation on it.

Return Value

The **os_mut_wait** function returns the completion value:

Return Value	Description
OS_R_MUT	The task waited until the mutex was released and has now acquired and locked the mutex.
OS_R_TMO	The timeout has expired.
OS_R_OK	The mutex was available and the **os_mut_wait** function returned to the calling task immediately.

See Also

os_mut_init, os_mut_release

Example

```
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
  os_mut_wait (mutex1, 0xffff);
   ..
  /* Critical region 1 */
   ..
  /* f2() will not block the task. */
  f2 ();
  os_mut_release (mutex1);
}

void f2 (void) {
  os_mut_wait (mutex1, 0xffff);
```

```
   ..
  /* Critical region 2 */
   ..
  os_mut_release (mutex1);
}

__task void task1 (void) {
   ..
  os_mut_init (mutex1);
  f1 ();
   ..
}

__task void task2 (void) {
   ..
  f2 ();
   ..
}
```

os_sem_init

Summary

```
#include <rtl.h>

void os_sem_init (
    OS_ID semaphore,         /* The semaphore object to initialize
*/
    U16   token_count );     /* Initial number of tokens */
```

Description

The **os_sem_init** function initializes the *semaphore* object identified by the function argument.

The argument *token_count* determines the number of tokens stored in the semaphore initially.

The **os_sem_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must define the semaphore object of type OS_SEM. You can use the name of the semaphore object to identify it during operation.

Return Value

The **os_sem_init** function does not return any value.

See Also

isr_sem_send, os_sem_send, os_sem_wait

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
   ..
  os_sem_init (semaphore1, 0);
  os_sem_send (semaphore1);
   ..
}
```

os_sem_send

Summary

```
#include <rtl.h>

OS_RESULT os_sem_send (
    OS_ID semaphore );     /* The semaphore whose token count is
incremented */
```

Description

The **os_sem_send** function increments the number of tokens in the *semaphore* object identified by the function argument.

The **os_sem_send** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.

Return Value

The **os_sem_send** function always returns OS_R_OK.

See Also

isr_sem_send, os_sem_init, os_sem_wait

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
   ..
  os_sem_init (semaphore1, 0);
  os_sem_send (semaphore1);
   ..
}
```

os_sem_wait

Summary

```
#include <rtl.h>

OS_RESULT os_sem_wait (
    OS_ID semaphore,    /* The semaphore to get the token from */
    U16   timeout );    /* Length of time to wait for the token */
```

Description

The **os_sem_wait** function requests a token from the *semaphore* identified by the function argument. If the token count in the semaphore is more than zero, the function gives a token to the calling task and decrements the token count in the semaphore. The calling task might then continue immediately or is put in the ready list depending on the priorities of other tasks in the ready list and the value of *timeout*.

If the token count in the semaphore is zero, the calling task is put to sleep by the RTX kernel. When a token becomes available in the semaphore or when the timeout period expires, the RTX kernel wakes the task and puts it in the ready list .

You can specify any value between 0 and 0xFFFE for the *timeout* argument. You must set *timeout* to 0xFFFF for an indefinite timeout period.

If you specify a value of 0 for the *timeout*, the calling task continues immediately even if there is a higher priority task in the ready list.

The **os_sem_wait** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Timeout is measured in number of system intervals.
- You must initialize the semaphore object using the **os_sem_init** function before you can perform any operation on the semaphore.

Return Value

The **os_sem_wait** function returns a completion value:

Return Value	Description
OS_R_SEM	The calling task has waited until a semaphore became available.
OS_R_TMO	The timeout expired before the token became available.
OS_R_OK	A token was available and the function returned immediately.

See Also

isr_sem_send, **os_sem_init**, **os_sem_send**

Example

```
#include <rtl.h>

OS_SEM semaphore1;

__task void task1 (void) {
   ..
  os_sem_wait (semaphore1, 0xffff);
   ..
}
```

os_sys_init

Summary

```
#include <rtl.h>

void os_sys_init (
    void (*task)(void) );   /* Task to start */
```

Description

The **os_sys_init** function initializes and starts the Real-Time eXecutive (RTX) kernel.

The *task* argument points to the task function to start after the kernel is initialized. The RTX kernel gives the task a default priority of 1.

The **os_sys_init** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init** function must be called from the **main** C function.
- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.

Return Value

The **os_sys_init** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init_prio, **os_sys_init_user**

Example

```
#include <rtl.h>

void main (void) {
  os_sys_init (task1);   /* start the kernel */
  while(1);
}
```

os_sys_init_prio

Summary

```
#include <rtl.h>

void os_sys_init_prio (
    void (*task)(void),    /* Task to start */
    U8 priority);         /* Task priority (1-254) */
```

Description

The **os_sys_init_prio** function initializes and starts the *Real-Time eXecutive* (RTX) kernel.

The *task* argument points to the task to start after the kernel is initialized.

The *priority* argument specifies the priority for the *task*. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the *priority*, it is automatically replaced with a value of 1. Priority 255 is also reserved.

The **os_sys_init_prio** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init_prio** function must be called from the **main** C function.
- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.
- Priority value of 255 represents the most important task.

Return Value

The **os_sys_init_prio** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init, os_sys_init_user

Example

```
#include <rtl.h>

void main (void) {
  os_sys_init_prio (task1, 10);
  while(1);
}
```

os_sys_init_user

Summary

```
#include <rtl.h>

void os_sys_init_user (
    void (*task)(void),     /* Task to start */
    U8    priority,         /* Task priority (1-254) */
    void* stack,            /* Task stack */
    U16   size);            /* Stack size */
```

Description

The **os_sys_init_user** function initializes and starts the *Real-Time eXecutive* (RTX) kernel. Use this function when you must specify a large stack for the starting task.

The *task* argument points to the task function to start after the kernel is initialized.

The *priority* argument specifies the priority for the *task*. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the *priority*, it is automatically replaced with a value of 1. Priority 255 is also reserved.

The *stack* argument points to a memory block reserved for the stack to use for the *task*. The *size* argument specifies the size of the stack in bytes.

The **os_sys_init_user** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The **os_sys_init_user** function must be called from the **main** C function.
- The stack must be aligned at an 8-byte boundary and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.
- Priority value of 255 represents the most important task.

Return Value

The **os_sys_init_user** function does not return. Program execution continues with the task identified by the *task* argument.

See Also

os_sys_init, os_sys_init_prio

Example

```
#include <rtl.h>

static U64 stk1[400/8];    /* 400-byte stack */

void main (void) {
  os_sys_init_user (task1, 10, &stk1, sizeof(stk1));
  while(1);
}
```

os_tmr_call

Summary

```
void os_tmr_call (
    U16 info );    /* Identification of an expired timer. */
```

Description

The **os_tmr_call** function is a user defined function that gets called by the RTX kernel's **task manager** task scheduler, when the user defined timer expires. After the **os_tmr_call** function returns, the task manager deletes this user timer.

The *info* argument contains the value that was specified when the timer was created using **os_tmr_create**.

The **os_tmr_call** function is part of RL-RTX. The prototype is defined in rtl.h. You can customize the function in rtx_config.c.

Note

- You can call any of the **isr_** system functions from the **os_tmr_call** function, but you cannot call any of the **os_** system functions.
- Do not call **os_tmr_kill** for an expired user timer.

Return Value

The **os_tmr_call** function does not return any value.

See Also

os_tmr_create, os_tmr_kill

Example

```
void os_tmr_call (U16 info) {
  switch (info) {
    case 1:           /* The supervised task is locked, */
                      /* recovery actions required.     */
      break;
    case 2:           /* The second task is locked.     */

      break;
       ..
  }
}
```

os_tmr_create

Summary

```
#include <rtl.h>

OS_ID os_tmr_create (
    U16 tcnt,       /* Length of the timer. */
    U16 info );     /* Argument to the callback function. */
```

Description

The **os_tmr_create** function sets up and starts a timer. When the timer expires, the RTX kernel calls the user defined **os_tmr_call** callback function and passes *info* as an argument to the **os_tmr_call** function.

The *tcnt* argument specifies the length of timer, in number of system ticks. You can specify tcnt to any value between 1 and 0xFFFF.

The **os_tmr_create** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_tmr_create** function returns a timer ID if the timer was successfully created. Otherwise, it returns NULL.

See Also

os_tmr_call, **os_tmr_kill**

Example

```
#include <rtl.h>

OS_TID tsk1;
OS_ID  tmr1;

__task void task1 (void) {
   ..
  tmr1 = os_tmr_create (10, 1);
  if (tmr1 == NULL) {
    printf ("Failed to create user timer.\n");
  }
   ..
}
```

os_tmr_kill

Summary

```
#include <rtl.h>

OS_ID os_tmr_kill (
    OS_ID timer );     /* ID of the timer to kill */
```

Description

The **os_tmr_kill** function deletes the *timer* identified by the function argument. *timer* is a user timer that was created using the *os_tmr_create* function. If you delete the timer before it expires, the *os_tmr_call* callback function does not get called.

The **os_tmr_kill** function is in the RL-RTX library. The prototype is defined in rtl.h.

- Do not call **os_tmr_kill** for an expired user timer. It has already been deleted by the system.

Return Value

The **os_tmr_kill** function returns NULL if the timer is killed successfully. Otherwise, it returns the *timer* value.

See Also

os_tmr_call, os_tmr_create

Example

```
#include <rtl.h>

OS_TID tsk1;
OS_ID  tmr1;

__task void task1 (void) {
   ..
  if (os_tmr_kill (tmr1) != NULL) {
    printf ("\nThis timer is not on the list.");
  }
  else {
    printf ("\nTimer killed.");
  }
   ..
}
```

os_tsk_create

Summary

```
#include <rtl.h>

OS_TID os_tsk_create (
    void (*task)(void),     /* Task to create */
    U8    priority );       /* Task priority (1-254) */
```

Description

The **os_tsk_create** function creates the task identified by the *task* function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The **os_tsk_create** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.
- Priority value of 255 represents the most important task.

Return Value

The **os_tsk_create** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create_ex, os_tsk_create_user, os_tsk_create_user_ex

Example

```
#include <rtl.h>

OS_TID tsk1, tsk2;

__task void task1 (void) {
   ..
  tsk2 = os_tsk_create (task2, 1);
   ..
}

__task void task2 (void) {
   ..
}
```

os_tsk_create_ex

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_ex (
    void (*task)(void *),     /* Task to create */
    U8    priority,          /* Task priority (1-254) */
    void* argv );             /* Argument to the task */
```

Description

The **os_tsk_create_ex** function creates the task identified by the *task* function pointer argument and adds the task to the ready queue. The function dynamically assigns a task identifier value (TID) to the new task. The **os_tsk_create_ex** function is an extension to the **os_tsk_create** function that enables you to pass an argument to the task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *argv* argument is passed directly to the task when it starts. An argument to a task can be useful to differentiate between multiple instances of the same task.

The **os_tsk_create_ex** function is in the RL-RTX library. The prototype is defined in rtl.h.

note

- The RTK kernel uses the default stack size, which is defined in **rtx_config.c**, for the task.

Return Value

The **os_tsk_create_ex** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, os_tsk_create_user, os_tsk_create_user_ex

Example

```
#include <rtl.h>

OS_TID tsk1, tsk2_0, tsk2_1;
int param[2] = {0, 1};

__task void task1 (void) {
   ..
  tsk2_0 = os_tsk_create_ex (task2, 1, &param[0]);
  tsk2_1 = os_tsk_create_ex (task2, 1, &param[1]);
   ..
}

__task void task2 (void *argv) {
   ..
  switch (*(int *)argv) {
    case 0:
      printf("This is a first instance of task2.\n");
      break;
    case 1:
      printf("This is a second instance of task2.\n");
      break;
    }

      . .
}
```

os_tsk_create_user

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_user(
    void (*task)(void),     /* Task to create */
    U8    priority,        /* Task priority (1-254) */
    void* stk,              /* Pointer to the task's stack */
    U16   size );           /* Number of bytes in the stack */
```

Description

The **os_tsk_create_user** function creates the task identified by the task function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task. This function enables you to provide a separate stack for the task. This is useful when a task needs a bigger stack for its local variables.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *stk* argument is a pointer to the memory block reserved for the stack of this task. The *size* argument specifies the number of bytes in the stack.

The **os_tsk_create_user** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The stack must be aligned at an 8-byte boundary, and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.

Return Value

The **os_tsk_create_user** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, os_tsk_create_ex, os_tsk_create_user_ex

Example

```
#include <rtl.h>

OS_TID tsk1,tsk2;
static U64 stk2[400/8];

__task void task1 (void) {
   ..
  /* Create task 2 with a bigger stack */
  tsk2 = os_tsk_create_user (task2, 1, &stk2, sizeof(stk2));
   ..
}

__task void task2 (void) {
  /* We need a bigger stack here. */
  U8 buf[200];
   ..
}
```

os_tsk_create_user_ex

Summary

```
#include <rtl.h>

OS_TID os_tsk_create_user_ex (
    void (*task)(void *),     /* Task to create */
    U8     priority,         /* Task priority (1-254) */
    void* stk,                /* Pointer to the task's stack */
    U16    size,              /* Size of stack in bytes */
    void* argv );             /* Argument to the task */
```

Description

The **os_tsk_create_user_ex** function creates the task identified by the *task* function pointer argument and then adds the task to the ready queue. It dynamically assigns a task identifier value (TID) to the new task. This function enables you to provide a separate stack for the task. This is useful when a task needs a bigger stack for its local variables. The **os_tsk_create_user_ex** function is an extension to the **os_tsk_create_user** function that enables you to pass an argument to the task.

The *priority* argument specifies the priority for the task. The default task priority is 1. Priority 0 is reserved for the Idle Task. If a value of 0 is specified for the priority, it is automatically replaced with a value of 1. Priority 255 is also reserved. If the new task has a higher priority than the currently executing task, then a task switch occurs immediately to execute the new task.

The *stk* argument is a pointer to the memory block reserved for the stack of this task. The *size* argument specifies the number of bytes in the stack.

The *argv* argument is passed directly to the task when it starts. An argument to a task can be useful to differentiate between multiple instances of the same task. Multiple instances of the same task can behave differently based on the argument.

The **os_tsk_create_user_ex** function is in the RL-RTX library. The prototype is defined in rtl.h.

note

- The stack *stk* must be aligned at an 8-byte boundary and must be declared as an array of type U64 (unsigned long long).
- The default stack size is defined in **rtx_config.c**.

Return Value

The **os_tsk_create_user_ex** function returns the task identifier value (TID) of the new task. If the function fails, for example due to an invalid argument, it returns 0.

See Also

os_tsk_create, os_tsk_create_ex, os_tsk_create_user

Example

```
#include <rtl.h>

OS_TID tsk1,tsk2_0,tsk2_1;
static U64 stk2[2][400/8];

__task void task1 (void) {
   ..
  /* Create task 2 with a bigger stack */
  tsk2_0 = os_tsk_create_user_ex (task2, 1,
                                  &stk2[0], sizeof(stk2[0]),
                                  (void *)0);
  tsk2_1 = os_tsk_create_user_ex (task2, 1,
                                  &stk2[1], sizeof(stk2[1]),
                                  (void *)1);
   ..
}

__task void task2 (void *argv) {
  /* We need a bigger stack here. */
  U8 buf[200];

   ..
  switch ((int)argv) {
    case 0:
      printf("This is a first instance of task2.\n");
      break;
    case 1:
      printf("This is a second instance of task2.\n");
      break;
  }
   ..
}
```

os_tsk_delete

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_delete (
    OS_TID task_id );     /* Id of the task to delete */
```

Description

If a task has finished all its work or is not needed anymore, you can terminate it using the **os_tsk_delete** function. The **os_tsk_delete** function stops and deletes the task identified by *task_id*.

The **os_tsk_delete** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- If *task_id* has a value of 0, the task that is currently running is stopped and deleted. The program execution continues with the task with the next highest priority in the ready queue.
- Deleting a task frees all dynamic memory resources allocated to that task.

Return Value

The **os_tsk_delete** function returns OS_R_OK if the task was successfully stopped and deleted. In all other cases, for example if the task with *task_id* does not exist or is not running, the function returns OS_R_NOK.

See Also

os_tsk_delete_self

Example

```
#include <rtl.h>
OS_TID tsk3;

__task void task2 (void) {
  tsk3 = os_tsk_create (task3, 0);
   ..
  if (os_tsk_delete (tsk3) == OS_R_OK) {
    printf("\n'task 3' deleted.");
  }
  else {
    printf ("\nFailed to delete 'task 3'.");
  }
}

__task void task3 (void) {
   ..
}
```

os_tsk_delete_self

Summary

```
#include <rtl.h>

void os_tsk_delete_self (void);
```

Description

The **os_tsk_delete_self** function stops and deletes the currently running task. The program execution continues with the task with the next highest priority in the ready queue.

The **os_tsk_delete_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- Deleting a task frees all dynamic memory resources allocated to that task.

Return Value

The **os_tsk_delete_self** function does not return. The program execution continues with the task with the next highest priority in the ready queue.

See Also

os_tsk_delete

Example

```
#include <rtl.h>

__task void task2 (void) {
   ..
  os_tsk_delete_self();
}
```

os_tsk_pass

Summary

```
#include <rtl.h>

void os_tsk_pass (void);
```

Description

The **os_tsk_pass** function passes control to the next task of the same priority in the ready queue. If there is no task of the same priority in the ready queue, the current task continues and no task switching occurs.

The **os_tsk_pass** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- You can use this function to implement a task switching system between several tasks of the same priority.

Return Value

The **os_tsk_pass** function does not return any value.

See Also

os_tsk_prio, os_tsk_prio_self

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
   ..
  os_tsk_pass();
   ..
}
```

os_tsk_prio

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_prio (
    OS_TID task_id,        /* ID of the task */
    U8     new_prio );     /* New priority of the task (1-254) */
```

Description

The **os_tsk_prio** function changes the execution priority of the task identified by the argument *task_id*.

If the value of *new_prio* is higher than the priority of the currently executing task, a task switch occurs to enable the task identified by *task_id* to run. If the value of *new_prio* is lower than the priority of the currently executing task, then the currently executing task resumes its execution.

If the value of *task_id* is 0, the priority of the currently running task is changed to *new_prio*.

The **os_tsk_prio** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The value of *new_prio* can be anything from 1 to 254.
- The new priority stays in effect until you change it.
- Priority 0 is reserved for the idle task. If priority 0 is specified to the function, it is automatically replaced with the value of 1 by the RTX kernel. Priority 255 is also reserved.
- A higher value for *new_prio* indicates a higher priority.

Return Value

The **os_tsk_prio** function returns one of these values:

Return Value	Description
OS_R_OK	The priority of a task has been successfully changed.
OS_R_NOK	The task with `task_id` does not exist or has not been started.

See Also

os_tsk_pass, os_tsk_prio_self

Example

```
#include <RTL.h>

OS_TID tsk1,tsk2;

__task void task1 (void) {
   ..
  os_tsk_prio_self (5);

  /* Changing the priority of task2 will cause a task switch. */
  os_tsk_prio(tsk2, 10);
   ..
}

__task void task2 (void) {
   ..
  /* Change priority of this task will cause task switch. */
  os_tsk_prio_self (1);
   ..
}
```

os_tsk_prio_self

Summary

```
#include <rtl.h>

OS_RESULT os_tsk_prio_self (
    U8 new_prio );     /* New priority of task (1-254) */
```

Description

The **os_tsk_prio_self** macro changes the priority of the currently running task to *new_prio*.

The **os_tsk_prio_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- The value of *new_prio* can be anything from 1 to 254.
- The new priority stays in effect until you change it.
- Priority 0 is reserved for the idle task. If priority 0 is specified to the function, it is automatically replaced with the value of 1 by the RTX kernel. Priority 255 is also reserved.
- A higher value for *new_prio* indicates a higher priority.

Return Value

The **os_tsk_prio_self** function always returns OS_R_OK.

See Also

os_tsk_pass, **os_tsk_prio**

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
   ..
  os_tsk_prio_self(10);  /* Increase its priority, for the
critical section */
   ..                    /* This is a critical section */
   ..
  os_tsk_prio_self(2);   /* Decrease its priority at end of
critical section */
   ..
}
```

os_tsk_self

Summary

```
#include <rtl.h>

OS_TID os_tsk_self (void);
```

Description

The **os_tsk_self** function identifies the currently running task by returning its task ID.

The **os_tsk_self** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **os_tsk_self** function returns the task identifier number (TID) of the currently running task.

See Also

isr_tsk_get, os_tsk_create, os_tsk_create_user

Example

```
#include <rtl.h>

OS_TID tsk1;

__task void task1 (void) {
  tsk1 = os_tsk_self();
   ..
}
```

poll_ethernet

Summary

```
#include <net_config.h>

void poll_ethernet (void);
```

Description

The **poll_ethernet** function polls the ethernet status register for any received ethernet packets. If there is a new packet, it allocates a block of memory. It then reads and copies the packet from the ethernet controller into the allocated memory. The function puts the pointer to the memory block into the received frames queue by calling the **put_in_queue** function.

The **poll_ethernet** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- You must provide the **poll_ethernet** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- You must provide the **poll_ethernet** function only if you want to use the ethernet driver in poll mode.
- The TcpNet system frequently calls **poll_ethernet** to poll for any received ethernet packet.

Return Value

The **poll_ethernet** function does not return any value.

See Also

init_ethernet, send_frame

Example

```
void poll_ethernet (void) {
  /* Poll the Ethernet controller for received frames. If the
Ethernet   */
  /* controller runs in interrupt mode, this function must be
empty.     */
  OS_FRAME *frame;
  U32 State, RxLen;
  U32 val, *dp;

  LREG (U16, BSR) = 2;
  val = LREG (U8, B2_IST);
  if (!(val & (IST_RCV | IST_RX_OVRN))) {
    /* Nothing received yet. */
    return;
  }

  if (val & IST_RX_OVRN) {
    /* Clear the RX overrun bit. */
    LREG (U8, B2_ACK) = ACK_RX_OVRN;
    return;
  }

  State = LREG (U16, B2_FIFO);
  if (State & FIFO_REMPTY) {
    /* Check if empty packet. */
    return;
  }
  /* Read status and packet length */
  LREG (U16, B2_PTR) = PTR_RCV | PTR_AUTO_INCR | PTR_READ;
  val = LREG (U32, B2_DATA);
  State = val & 0xFFFF;
  RxLen = (val >> 16) - 6;
  if (State & RFS_ODDFRM) {
    /* Odd number of bytes in a frame. */
    RxLen++;
  }

  if (RxLen > ETH_MTU) {
    /* Packet too big, ignore it and free MMU. */
    LREG (U16, BSR)      = 2;
    LREG (U16, B2_MMUCR) = MMU_REMV_REL_RX;
    return;
  }

  frame = alloc_mem (RxLen);

  /* Make sure that block is 4-byte aligned */
  RxLen = (RxLen + 3) >> 2;

  dp = (U32 *)&frame->data[0];
  for (  ; RxLen; RxLen--) {
    *dp++ = LREG (U32, B2_DATA);
  }

  /* MMU free packet. */
  LREG (U16, BSR)      = 2;
  LREG (U16, B2_MMUCR) = MMU_REMV_REL_RX;

  put_in_queue (frame);
}
```

ppp_close

Summary

```
#include <rtl.h>

void ppp_close (void);
```

Description

The **ppp_close** function disconnects the PPP link between the two modems.

The **ppp_close** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You can call the **ppp_close** function when the PPP network daemon is either in client mode or in server mode.
- Since it is the PPP client that starts a dial-up connection, it is also the PPP client that usually disconnects the PPP link.
- The **ppp_close** function does not change the running mode of the PPP daemon. If the PPP daemon was in server mode, PPP daemon re-initializes the modem driver to accept further incoming calls.

Return Value

The **ppp_close** function does not return any value.

See Also

ppp_connect, ppp_is_up, ppp_listen

Example

```
#include <rtl.h>

void disconnect_link (void) {
  /* Disconnect a dial-up link. */
  ppp_close ();
}
```

ppp_connect

Summary

```
#include <rtl.h>

void ppp_connect (
    char const* dialnum,     /* Pointer to string containing the
number to dial. */
    char const* user,        /* Username for authentication. */
    char const* passw );     /* Password for authentication. */
```

Description

The **ppp_connect** function starts a dial-up connection to the remote PPP server by starting the PPP daemon in client mode.

The argument *dialnum* points to a null terminated ASCII string containing the phone number of the remote PPP server.

The argument *user* points to the username, and the argument *passw* points to the password. The TcpNet system uses username and password for remote user authentication using the Password Authentication Protocol (PAP). Both arguments are null terminated ASCII strings.

The **ppp_connect** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **ppp_connect** function when you want to establish a dial-up connection using PPP.
- If you use a direct cable connection, you can set *dialnum* to NULL because the Null_Modem device driver ignores the argument *dialnum*.

Return Value

The **ppp_connect** function does not return any value.

See Also

ppp_close, ppp_is_up, ppp_listen

Example

```
#include <rtl.h>

void dial_remote (void) {
  /* Dial remote PPP Server. */
  ppp_connect ("04213372", "Keil", "test");
}
```

ppp_is_up

Summary

```
#include <rtl.h>

BOOL ppp_is_up (void);
```

Description

The **ppp_is_up** function determines the state of PPP link between the two modems. It returns __TRUE if the PPP link state is "network" and IP frames can be exchanged.

The **ppp_is_up** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You can call the **ppp_is_up** function when the PPP network daemon is either in client mode or in server mode.
- You can also use the **ppp_is_up** function to continuously monitor the state of the PPP link.

Return Value

The **ppp_is_up** function returns __TRUE if the PPP link between the modems is up and functional. The function returns __FALSE if the PPP link is down.

See Also

ppp_close, ppp_connect, ppp_listen

Example

```
#include <rtl.h>

void connect_soc (void) {
  /* Connect TCP socket when PPP is up. */
  if(ppp_is_up()) {
    tcp_connect (socket_tcp, remip, 1000, 0);
  }
}
```

ppp_listen

Summary

```
#include <rtl.h>

void ppp_listen (
    char const* user,        /* Username for authentication. */
    char const* passw );     /* Password for authentication. */
```

Description

The **ppp_listen** function configures the PPP interface to accept incoming PPP connections by starting the PPP daemon in server mode.

The argument *user* points to the username, and the argument *passw* points to the password. The TcpNet system uses username and password for remote user authentication using the Password Authentication Protocol (PAP). Both arguments are null-terminated ASCII strings.

The **ppp_listen** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- It is common to call the **ppp_listen** function at system startup.

Return Value

The **ppp_listen** function does not return any value.

See Also

ppp_close, **ppp_connect**, **ppp_is_up**

Example

```
#include <rtl.h>

void main (void) {
  init_TcpNet ();
  /* Activate the PPP interface. */
  ppp_listen ("Keil", "test");
  while (1) {
    timer_poll ();
    main_TcpNet ();
  }
}
```

rewind

Summary

```
#include <stdio.h>

void rewind (
    FILE *stream);    /* file stream to rewind */
```

Description

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. Error and end-of-file indicators are cleared.

The **rewind** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Return Value

The **rewind** function does not return any value.

See Also

fseek, **ftell**

Example

```
#include <rtl.h>
#include <stdio.h>

void tst_rewind (void) {
  char line[80];
  FILE *fin;

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    while (fgets (&line, sizeof (line), fin) != NULL);
    // read again the first line
    rewind (fin);
    fgets (&line, sizeof (line), fin);
    fclose (fin);
  }
}
```

send_frame

Summary

```
#include <net_config.h>

void send_frame (
    OS_FRAME* frame );     /* Pointer to the ethernet frame to
send. */
```

Description

The **send_frame** function sends the ethernet frame packet using the ethernet controller. The argument *frame* points to the ethernet frame constructed by the TCPnet system. The component *frame->length* holds the total length of the frame that needs to be transferred.

The **send_frame** function is part of RL-TCPnet. The prototype is defined in net_config.h.

note

- You must provide the **send_frame** function if the ethernet controller you use is different from the ones provided in the TCPnet source.
- The TCPnet system calls **send_frame** when it has constructed the ethernet frame that needs to be sent to the network.

Return Value

The **send_frame** function does not return any value.

See Also

init_ethernet, poll_ethernet

Example

```
void send_frame (OS_FRAME *frame) {
  /* Send frame to EMAC ethernet controller */
  U32 idx,len;
  U32 *sp,*dp;

  idx = MAC_TXPRODUCEINDEX;
  sp  = (U32 *)&frame->data[0];
  dp  = (U32 *)Tx_Desc[idx].Packet;

  /* Copy frame data to EMAC packet buffers. */
  for (len = (frame->length + 3) >> 2; len; len--) {
    *dp++ = *sp++;
  }
  Tx_Desc[idx].Ctrl = (frame->length-1) | (TCTRL_INT |
TCTRL_LAST);

  /* Start frame transmission. */
  if (++idx == NUM_TX_FRAG) idx = 0;
  MAC_TXPRODUCEINDEX = idx;
}
```

slip_close

Summary

```
#include <rtl.h>

void slip_close (void);
```

Description

The **slip_close** function disconnects the SLIP link between the two modems.

The **slip_close** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You can call the **slip_close** function when the SLIP network daemon is either in client mode or in server mode.
- Since it is the SLIP client that starts a dial-up connection, it is also the SLIP client that usually disconnects the SLIP link.
- The **slip_close** function does not change the running mode of the SLIP daemon. If the SLIP daemon was in server mode, SLIP daemon re-initializes the modem driver to accept further incoming calls.

Return Value

The **slip_close** function does not return any value.

See Also

slip_connect, slip_is_up, slip_listen

Example

```
#include <rtl.h>

void disconnect_link (void) {
  /* Disconnect a dial-up link. */
  slip_close ();
}
```

slip_connect

Summary

```
#include <rtl.h>

void slip_connect (
    char const* dialnum );     /* Pointer to string containing the
number to dial. */
```

Description

The **slip_connect** function starts a dial-up connection to the remote SLIP server by starting the SLIP daemon in client mode.

The argument *dialnum* points to a null terminated ASCII string containing the phone number of the remote SLIP server.

The **slip_connect** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **slip_connect** function when you want to establish a dial-up connection using SLIP.
- If you use a direct cable connection, you can set *dialnum* to NULL because the Null_Modem device driver ignores the argument *dialnum*.

Return Value

The **slip_connect** function does not return any value.

See Also

slip_close, slip_is_up, slip_listen

Example

```
#include <rtl.h>

void dial_remote (void) {
  /* Dial remote SLIP Server. */
  slip_connect ("04213372");
}
```

slip_is_up

Summary

```
#include <rtl.h>

BOOL slip_is_up (void);
```

Description

The **slip_is_up** function determines the state of SLIP link between the two modems. It returns __TRUE if IP frames can be exchanged over the SLIP link.

The **slip_is_up** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You can call the **slip_is_up** function when the SLIP network daemon is either in client mode or in server mode.
- You can also use the **slip_is_up** function to continuously monitor the state of the SLIP link.

Return Value

The **slip_is_up** function returns __TRUE if the SLIP link between the modems is up and functional. The function returns __FALSE if the SLIP link is down.

See Also

slip_close, slip_connect, slip_listen

Example

```
#include <rtl.h>

void connect_soc (void) {
  /* Connect TCP socket when SLIP is up. */
  if(slip_is_up()) {
    tcp_connect (socket_tcp, remip, 1000, 0);
  }
}
```

slip_listen

Summary

```
#include <rtl.h>

void slip_listen (void);
```

Description

The **slip_listen** function configures the SLIP interface to accept incoming SLIP connections by starting the SLIP daemon in server mode.

The **slip_listen** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- It is common to call the **slip_listen** function at system startup.

Return Value

The **slip_listen** function does not return any value.

See Also

slip_close, **slip_connect**, **slip_is_up**

Example

```
#include <rtl.h>

void main (void) {
  init_TcpNet ();
  /* Activate the SLIP interface. */
  slip_listen ();
  while (1) {
    timer_poll ();
    main_TcpNet ();
  }
}
```

smtp_accept_auth

Summary

```
#include <net_config.h>

BOOL smtp_accept_auth (
    U8* srv_ip );       /* IP address of SMTP server. */
```

Description

The **smtp_accept_auth** function informs the TCPnet if the SMTP client should log on to SMTP Server when sending e-mails. The TCPnet library calls this function and asks the user what to do if the SMTP Server has advertised the user authentication. It is now on the user to accept the authentication or not.

The argument *srv_ip* points to a buffer containing the four octets that make up the ip address of the remote SMTP server.

The **smtp_accept_auth** function is part of RL-TCPnet. The prototype is defined in net_config.h. You can customize the function in smtp_uif.c.

Return Value

The **smtp_accept_auth** function returns __TRUE if the authentication is accepted and __FALSE, when the user does not want to log on to SMTP server to send e-mails.

If this function returns __FALSE, the **smtp_cbfunc** with the code 0 and 1 is not called.

See Also

smtp_cbfunc

Example

```
BOOL smtp_accept_auth (U8 *srv_ip) {

  if (srv_ip[0] == 192  &&
      srv_ip[1] == 168  &&
      srv_ip[2] == 1    &&
      srv_ip[3] == 253) {
    /* Our local SMTP server, no authentication. */
    return (__FALSE);
  }
  /* Log on to SMTP Server. */
  return (__TRUE);
}
```

smtp_cbfunc

Summary

```
#include <net_config.h>

U16 smtp_cbfunc (
    U8    code,       /* Type of data requested by the SMTP Client.
*/
    U8*   buf,        /* Location where to write the requested data.
*/
    U16   buflen,     /* Number of bytes in the output buffer. */
    U32* pvar );      /* Pointer to a storage variable. */
```

Description

The **smtp_cbfunc** function provides the email headers and data, in SMTP format, when requested by the SMTP client running on the TCPnet system. The SMTP client calls this function several times to compose the email message.

The argument *code* specifies the type of email section (user, header or data) that the SMTP Client requires. This is shown in the table.

Code	Email section	Description
0	Username:	Username for SMTP authentication
1	Password:	Password for SMTP authentication
2	From:	Email address of the sender
3	To:	Email address of the recipient
4	Subject:	Subject of the email
5	Data:	Email body in plain ASCII format

The argument *buf* is a pointer to the output buffer where the **smtp_cbfunc** function writes the requested data into. The argument *buflen* specifies the length of the output buffer in bytes.

The argument *pvar* is a pointer to a variable that never gets altered by the SMTP client. You can use **pvar* as a repeat counter or simply to distinguish between different calls of the **smtp_cbfunc** function.

The **smtp_cbfunc** function is part of RL-TCPnet. The prototype is defined in net_config.h. You must customize the function in smtp_uif.c to compose the email message correctly.

note

- The length of the output buffer, *buflen* might vary because buffer length is determined by the TCP socket Maximum Segment Size (MSS) negotiation. The buffer length is normally around 1400 bytes for local LAN. But this can be reduced to 500 bytes or even less.
- If the **smtp_cbfunc** function writes more bytes than buflen into the output buffer, then a system crash resulting from corruption of memory link pointers is highly likely.
- The argument *pcgi* is private to each SMTP Session. The SMTP Client clears the data in the *pcgi* pointer, to 0, before the **smtp_cbfunc** function is called for the first time.

Return Value

The **smtp_cbfunc** function returns the number of bytes written to the output buffer, and it writes the repeat flag value in the most significant bit of the return value.

If the return value's most significant bit is set to 1, the SMTP client running on TCPnet calls the **smtp_cbfunc** function again with the same value for the argument *code*, and *pcgi*, which holds the same content as previously set. The function **smtp_cbfunc** can then enter more data into the buffer *buf*.

See Also

smtp_accept_auth, smtp_connect

Example

```
typedef struct {
  U8  id;
  U16 idx;
} MY_BUF;
#define MYBUF(p)        ((MY_BUF *)p)

U16 smtp_cbfunc (U8 code, U8 *buf, U16 buflen, U32 *pvar) {
  U32 i,len = 0;

  switch (code) {
    case 0:
      /* Enter Username for SMTP Server authentication. */
      len = str_copy (buf, "user");
      break;

    case 1:
      /* Enter Password for SMTP Server authentication. */
      len = str_copy (buf, "password");
      break;

    case 2:
      /* Enter email address of the sender. */
      len = str_copy (buf, "mcb@keil.com");
      break;

    case 3:
      /* Enter email address of the recipient. */
      len = str_copy (buf, "somebody@keil.com");
      break;

    case 4:
      /* Enter email subject. */
      len = str_copy (buf, "Reported measurements");
      break;

    case 5:
      /* Enter email data. */
      switch (MYBUF(pvar)->id) {
        case 0:
          /* First call, enter an email header text. */
          len = str_copy (buf, "Here is the log file:\r\n\r\n");
          MYBUF(pvar)->id  = 1;
          MYBUF(pvar)->idx = 1;
          goto rep;

        case 1:
          /* Add email message body. */
          for (len = 0; len < buflen-150; ) {
            /* Let's use as much of the buffer as possible. */
            /* This will produce less packets and speedup the
transfer. */
```

```
            len += sprintf ((char *)(buf+len), "%d.
",MYBUF(pvar)->idx);
            for (i = 0; i < 8; i++) {
              len += sprintf ((char *)(buf+len), "AD%d= %d
",i,AD_in(i));
            }
            len += str_copy (buf+len, "\r\n");
            if (++MYBUF(pvar)->idx > 500) {
              MYBUF(pvar)->id = 2;
              break;
            }
          }
          /* Request a repeated call, bit 15 is a repeat flag. */
rep:      len |= 0x8000;
          break;

        case 2:
          /* Last one, add a footer text to this email. */
          len = str_copy (buf, "OK, that is all. \r\nBye..\r\n");
          break;
      }
  }
  return ((U16)len);
}
```

smtp_connect

Summary

```
#include <rtl.h>

BOOL smtp_connect (
    U8*    ipadr,                      /* IP address of the SMTP
server. */
    U16    port,                       /* Port number of SMTP server.
*/
    void (*cbfunc)(U8 event) );        /* Function to call when the
SMTP session ends. */
```

Description

The **smtp_connect** function starts the SMTP client on the TCPnet system. This causes the SMTP client to then start an SMTP session by connecting to an SMTP server on the TCP *port* specified in the function argument.

The argument *ipadr* points to an array of 4 bytes containing the dotted-decimal notation of the IP address of the SMTP server to connect to.

The argument *cbfunc* points to a function that the SMTP client running on TCPnet calls when the SMTP session ends. The **cbfunc** is an event callback function that uses the *event* argument of the **cbfunc** function to signal one of the following SMTP events:

Event	Description
SMTP_EVT_SUCCESS	The email has been successfully sent.
SMTP_EVT_TIMEOUT	SMTP Server response has timed out, and hence the SMTP client has aborted the operation. The email has not been sent.
SMTP_EVT_ERROR	Protocol error occurred when sending the email. The email has not been sent.

The **smtp_connect** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- The standard SMTP port is TCP port 25.
- Your application, running on TCPnet, can call the **smtp_connect** function to connect to an SMTP server to send emails.

Return Value

The **smtp_connect** function returns __TRUE if the SMTP client has been successfully started. Otherwise it returns __FALSE.

See Also

smtp_cbfunc

Example

```
static void smtp_cback (U8 event);

void send_email (void) {
  U8 smtpip[4] = {192,168,2,253};

  if (smtp_connect (smtpip, 25, smtp_cback) == 0) {
    printf("E-mail not sent, SMTP Client not ready.\n");
  }
  else {
    printf("SMTP Client started.\n");
  }
}
```

```
static void smtp_cback (U8 event) {
  switch (event) {
    case SMTP_EVT_SUCCESS:
      printf ("Email successfully sent\n");
      break;
    case SMTP_EVT_TIMEOUT:
      /* Timeout, try again. */
      printf ("Mail Server timeout.\n");
      break;
    case SMTP_EVT_ERROR:
      /* Error, try again. */
      printf ("Error sending email.\n");
      break;
  }
}
```

snmp_set_community

Summary

```
#include <net_config.h>

BOOL snmp_set_community (
    const char *community);    /* Pointer to a Community string. */
```

Description

The **snmp_set_community** function changes the SNMP community to a new community identified with a parameter *community*.

Parameter *community* is a pointer to a 0-terminated string. The maximum length of the community string is limited to **18 characters**.

Return Value

The **snmp_set_community** function returns __TRUE when the SNMP community has been changed successfully. On error, this function returns __FALSE.

See Also

snmp_trap

Example

```
void send_alarm (void) {
  /* Send a trap message when alarm input is activated.*/
  U16 obj[2];

  /* Change the community to "private". */
  snmp_set_community ("private");

  /* Add "KeyIn" value to trap message. */
  obj[0] = 1;
  obj[1] = 8;     /* Index of "KeyIn" entry in MIB table. */
  snmp_trap (NULL, 6, 1, obj);

  /* Restore the community to "public". */
  snmp_set_community ("public");
}
```

snmp_trap

Summary

```
#include <net_config.h>

BOOL snmp_trap (
    U8*   man_ip,        /* Pointer to the IP address of Trap
Manager. */
    U8    gen_trap,      /* Generic Trap value. */
    U8    spec_trap,     /* Specific Trap value. */
    U16* obj_list );     /* Pointer to the list of objects. */
```

Description

The **snmp_trap** function sends a trap message to the Trap Manager. Parameter *man_ip* specifies the IP address of the Trap server, where the trap message is destined to. If the IP address of the Trap Manager is not specified (parameter *man_ip* is **NULL**), the IP address of Trap Server configured in **Net_Config.c** is used instead.

Parameter *gen_trap* specifies the **generic** trap type:

Type	Description
0	coldStart trap
1	warmStart trap
2	linkDown trap
3	linkUp trap
4	authenticationFailure trap
5	egpNeighborLoss trap
6	enterpriseSpecific trap

Parameter *spec_trap* specifies the **specific** trap type. It must be set to 0 for all generic traps from 0 to 5. It defines a specific trap type for generic **enterpriseSpecific** trap.

Parameter *obj_list* specifies the objects from the **MIB** table, which will be included in the trap message **variable-bindings**. This parameter is a pointer to the **object list** array. This array is of variable size. The first element specifies the count of objects in the **object list** array, followed by the object MIB index values.

Array Index	Array Value
obj[0]	number of objects **n**
obj[1]	MIB index of first object
obj[2]	MIB index of second object
..	..
obj[**n**]	MIB index of last object

If *obj_list* parameter is **NULL**, or **obj[0] = 0**, no object values will be binded to the trap message.

The **snmp_trap** function is a system function that is in the RL-TCPnet library. The prototype is defined in net_config.h.

- The maximum number of objects that can be binded to the trap message is limited to **20 objects**.

Return Value

The **snmp_trap** function returns __TRUE when the SNMP trap message has been sent successfully. On error, this function returns __FALSE.

See Also

snmp_set_community

Example

```
void send_alarm (void) {
  /* Send a trap message when alarm input is activated.*/
  U16 obj[2];
  /* Add "KeyIn" value to trap message. */
  obj[0] = 1;
  obj[1] = 8;     /* Index of "KeyIn" entry in MIB table. */
  snmp_trap (NULL, 6, 1, obj);
}
```

spi.BusSpeed

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*BusSpeed) (
    U32 kbaud);       /* Bus speed in kilo-baud */
   ..
} const SPI_DRV;
```

Description

The **BusSpeed** function sets the transfer speed on the SPI interface to requested baud rate. When SD/MMC Flash Cards are initialized from native to SPI mode, low speed transfer (400 kBit/s maximum) is used. When the Card initialization is complete, the high speed SPI data transfer is used.

The argument *kbaud* specifies the requested baud rate.

The **BusSpeed** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

- It is important to set the actual SPI speed equal to (or less than) the requested baud rate *kbaud*, but **not higher** than the requested baud rate. The error might happen due to the integer math used for the calculation of a divide factor.

Return Value

The **BusSpeed** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.CheckMedia, spi.Init, spi.RecBuf, spi.Send, spi.SendBuf, spi.SetSS, spi.UnInit

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL BusSpeed (U32 kbaud) {
  /* Set an SPI clock to required baud rate. */
  U32 div;

  div = (__PCLK/1000 + kbaud - 1) / kbaud;
  if (div == 0)   div = 0x02;
  if (div & 1)    div++;
  if (div > 0xFE) div = 0xFE;
  SSPCPSR = div;
  return (__TRUE);
}
```

spi.CheckMedia

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U32   (*CheckMedia)  (void);           /* Optional, NULL if not
existing    */
} const SPI_DRV;
```

Description

The **CheckMedia** is a user-provided routine that checks the SD/MMC Memory Card status. It reads the **Card Detect** (CD) and **Write Protect** (WP) digital inputs. If CD and WP digital inputs from SD Card socket are not connected, this function might be omitted. In this case enter the **NULL** value for CheckMedia into the SPI Driver control block. It is also possible to provide this function, which always returns **M_INSERTED** status.

The **CheckMedia** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

Return Value

The **CheckMedia** function returns the or-ed status of the following values:

- **M_INSERTED**
  SD Card is inserted in the socket.
- **M_PROTECTED**
  SD Card is read-only. Lock slider is in position Locked.

See Also

spi.BusSpeed, spi.Init, spi.RecBuf, spi.Send, spi.SendBuf, spi.SetSS, spi.UnInit

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static U32 CheckMedia (void) {
  /* Read CardDetect and WriteProtect SD card socket pins. */
  U32 stat = 0;

  if (!(IOPIN0 & 0x04)) {
    /* Card is inserted (CD=0). */
    stat |= M_INSERTED;
  }
  if ((IOPIN0 & 0x20)) {
    /* Write Protect switch is active (WP=1). */
    stat |= M_PROTECTED;
  }
  return (stat);
}
```

spi.Init

Summary

```
#include <file_config.h>

typedef struct {
  BOOL (*Init) (void);
   ..
} const SPI_DRV;
```

Description

The **Init** function is a user-provided routine that initializes the SPI serial interface. It is invoked by the **finit** function on system startup.

The **Init** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

Return Value

The **Init** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.BusSpeed, spi.CheckMedia, spi.RecBuf, spi.Send, spi.SendBuf, spi.SetSS, spi.UnInit

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL Init (void) {
  /* Initialize and enable the SSP Interface module. */

  /* SSEL is GPIO, output set to high. */
  IODIR0 |= 1<<20;
  IOSET0  = 1<<20;
  /* SCK1, MISO1, MOSI1 are SSP pins. */
  PINSEL1 = (PINSEL1 & ~0x000003FC) | 0x000000A8;

  /* Enable SPI in Master Mode, CPOL=0, CPHA=0. */
  SSPCR0  = 0x0007;
  SSPCR1  = 0x0002;
  SSPCPSR = 0xFE;
  return (__TRUE);
}
```

spi.RecBuf

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*RecBuf) (
    U8 *buf,     /* buffer to read data to   */
    U32 sz);     /* size of the data to read */
   ..
} const SPI_DRV;
```

Description

The **RecBuf** is a user-provided routine that receives the data from the SPI interface into the buffer *buf* for *sz* bytes.

This function is used to receive large data packets. Typically several 512-byte FAT sectors or large data buffers for the Embedded File System.

The **RecBuf** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

- This function might be implemented in **DMA mode** for faster transfer.

Return Value

The **RecBuf** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.BusSpeed, spi.CheckMedia, spi.Init, spi.Send, spi.SendBuf, spi.SetSS, spi.UnInit

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL RecBuf (U8 *buf, U32 sz) {
  /* Receive SPI data to buffer. */
  U32 i;

  for (i = 0; i < sz; i++) {
    SSPDR = 0xFF;
    /* Wait while Rx FIFO is empty. */
    while (!(SSPSR & RNE));
    buf[i] = SSPDR;
  }
  return (__TRUE);
}
```

spi.Send

Summary

```
#include <file_config.h>

typedef struct {
   ..
  U8   (*Send) (
    U8 outb);    /* Byte to send to SPI interface. */
   ..
} const SPI_DRV;
```

Description

The **Send** function sends one byte of data over the SPI interface, and it receives a byte of data from the SPI interface. The Flash File System calls the function to send or receive commands, status values, or data to the SD/MMC Flash Memory Card or to the SPI Data Flash Memory.

The argument *outb* is the command or data to transmit over the SPI interface.

The **Send** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

Return Value

The **Send** function returns a byte received on the SPI data input.

See Also

spi.BusSpeed, **spi.CheckMedia**, **spi.Init**, **spi.RecBuf**, **spi.SendBuf**, **spi.SetSS**, **spi.UnInit**

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static U8 Send (U8 outb) {
  /* Send and Receive a byte on SPI interface. */

  SSPDR = outb;
  /* Wait if RNE cleared, Rx FIFO is empty. */
  while (!(SSPSR & RNE));
  return (SSPDR);
}
```

spi.SendBuf

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*SendBuf) (
    U8 *buf,    /* buffer containing the data */
    U32 sz);    /* size of the data buffer    */
   ..
} const SPI_DRV;
```

Description

The **SendBuf** function is a user-provided routine that sends the contents of *buf* to the SPI interface for *sz* bytes.

This function is used to send large data packets. Typically several 512-byte FAT sectors or large data buffers for the Embedded File System.

The **SendBuf** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

- This function might be implemented in **DMA mode** for faster transfer.

Return Value

The **SendBuf** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.BusSpeed, **spi.CheckMedia**, **spi.Init**, **spi.RecBuf**, **spi.Send**, **spi.SetSS**, **spi.UnInit**

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL SendBuf (U8 *buf, U32 sz) {
  /* Send buffer to SPI interface. */
  U32 i;

  for (i = 0; i < sz; i++) {
    SSPDR = buf[i];
    /* Wait if Tx FIFO is full. */
    while (!(SSPSR & TNF));
    SSPDR;
  }
  /* Wait until Tx finished, drain Rx FIFO. */
  while (SSPSR & (BSY | RNE)) {
    SSPDR;
  }
  return (__TRUE);
}
```

spi.SetSS

Summary

```
#include <file_config.h>

typedef struct {
   ..
  BOOL (*SetSS) (
    U32 ss);    /* Enable/Disable SPI Slave Select */
   ..
} const SPI_DRV;
```

Description

The **SetSS** function enables or disables Slave Select on the SPI interface. The argument *ss* specifies state of the Slave Select signal.

The **SetSS** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

Return Value

The **SetSS** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.BusSpeed, spi.CheckMedia, spi.Init, spi.RecBuf, spi.Send, spi.SendBuf, spi.UnInit

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                          /* Can be NULL if not
existing */
};

static BOOL SetSS (U32 ss) {
  /* Enable/Disable SPI Chip Select (drive it high or low). */

  if (ss) {
    /* SSEL is GPIO, output set to high. */
    IOSET0 = 1<<20;
  }
  else {
    /* SSEL is GPIO, output set to low. */
    IOCLR0 = 1<<20;
  }
  return (__TRUE);
}
```

spi.UnInit

Summary

```
#include <file_config.h>

typedef struct {
    ..
  BOOL (*UnInit) (void);
    ..
} const SPI_DRV;
```

Description

The **UnInit** function is a user-provided routine in the **SPI driver** that uninitializes the SPI serial interface. It is invoked by the **funinit** function.

It can be used if during the application run time the embedded flash or SD Card drive to be disabled for whatever reason (for example lowering power consumption). After this function is called only the **finit** function should be called for reinitialization of the drive.

The **UnInit** function is in the **SPI driver**. The prototype is defined in file_config.h. You have to customize the function in your own SPI driver.

Return Value

The **UnInit** function returns a value of __TRUE if successful or a value of __FALSE if unsuccessful.

See Also

spi.BusSpeed, **spi.CheckMedia**, **spi.Init**, **spi.RecBuf**, **spi.Send**, **spi.SendBuf**, **spi.SetSS**

Example

```
/* SPI Device Driver Control Block */
SPI_DRV spi0_drv = {
  Init,
  UnInit,
  Send,
  SendBuf,
  RecBuf,
  BusSpeed,
  SetSS,
  CheckMedia                         /* Can be NULL if not
existing */
};

static BOOL UnInit (void) {
  /* Return SSP interface to default state. */

  IODIR0   &= ~(1<<20);
  PINSEL1  &= ~0x000003FC;
  SSPCR1   = 0x0000;
  SSPCR0   = 0x0000;
  SSPCPSR  = 0x00;
  return (__TRUE);
}
```

tcp_abort

Summary

```
#include <rtl.h>

BOOL tcp_abort (
    U8 socket);     /* TCP socket to abort. */
```

Description

The **tcp_abort** function closes the TCP connection immediately by sending a TCP frame with the RESET flag set to the remote machine. The argument *socket* specifies the handle of the socket whose connection is to be closed.

The **tcp_abort** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- TCPnet calls the **listener** callback function when the connection has aborted.
- After calling **tcp_abort** you cannot use the socket to send or receive any data.
- After calling **tcp_abort** the socket still remains allocated until you release it.

Return Value

The **tcp_abort** function returns __TRUE if the connection has been closed successfully. Otherwise, the function returns __FALSE.

See Also

tcp_close, tcp_release_socket

Example

```
#include <rtl.h>

void disconnect_tcp (U8 tcp_soc) {
   ..
  /* This TCP connection needs to close immediately */
  tcp_abort (tcp_soc);
  /* Socket will not be needed anymore */
  tcp_release_socket (tcp_soc);
}
```

tcp_check_send

Summary

```
#include <rtl.h>

BOOL tcp_check_send (
    U8 socket );    /* TCP socket to check whether it can send
data. */
```

Description

The **tcp_check_send** function determines if the TCP socket can send data. It does this by checking whether the TCP connection has been established and whether the socket has received an acknowledgement from the remote machine for the previously sent data. The argument *socket* specifies the handle of the socket to check.

The **tcp_check_send** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **tcp_check_send** function returns __TRUE if the socket is ready to send data. Otherwise, the function returns __FALSE.

See Also

tcp_get_state

Example

```
#include <rtl.h>

void send_data () {
  U8 remip[4] = {192,168,1,100};
  U8 *sendbuf;

  switch (tcp_get_state (socket_tcp)) {
    case TCP_STATE_FREE:
    case TCP_STATE_CLOSED:
      /* Connection idle, send Connect Request. */
      tcp_connect (socket_tcp, Rem_IP, 1001, 0);
      break;
    case TCP_STATE_CONNECT:
      /* We are connected, send command to remote peer. */
      if (tcp_check_send (socket_tcp)) {
        /* OK, socket is ready to send data. */
        sendbuf = tcp_get_buf (2);
        sendbuf[0] = BLINKLED;
        sendbuf[1] = p2;
        tcp_send (socket_tcp, sendbuf, SENDLEN);
      }
      break;
  }
}
```

tcp_close

Summary

```
#include <rtl.h>

BOOL tcp_close (
    U8 socket );     /* TCP socket to close. */
```

Description

The **tcp_close** function initiates the procedure to close the TCP connection. It might take some time to close the connection. The argument *socket* specifies the handle of the socket whose connection is to be closed.

The **tcp_close** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- TCPnet calls the **listener** callback function when the connection has closed.
- After calling **tcp_close** you cannot use the socket to send or receive any data.
- After calling **tcp_close** the socket still remains allocated until you release it.

Return Value

The **tcp_close** function returns __TRUE if the connection closing procedure has been started successfully. Otherwise, the function returns __FALSE.

See Also

tcp_abort, **tcp_release_socket**

Example

```
#include <rtl.h>

void disconnect_tcp (U8 tcp_soc) {
   ..
  /* This TCP connection is no longer needed */
  tcp_close (tcp_soc);
  /* Release TCP Socket in tcp event callback function */
  /* when event TCP_EVT_CLOSE occured.                 */
}
```

tcp_connect

Summary

```
#include <rtl.h>

BOOL tcp_connect (
    U8   socket,        /* Socket handle of the local machine. */
    U8*  remip,         /* Pointer to IP address of remote machine.
*/
    U16  remport,       /* Port number of remote machine. */
    U16  locport );     /* Port number of local machine. */
```

Description

The **tcp_connect** function initiates a connection to a remote server. The argument *socket* is a handle to the socket to use for communication on the local machine. The argument *locport* specifies the port to use for communication on the local machine. If *locport* is set to 0, TCPnet automatically allocates the first free TCP port.

The argument *remip* points to the buffer containing the ip address octets of the remote server to connect to. The argument *remport* specifies the TCP port number on the remote machine to communicate with.

The **tcp_connect** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- Only a socket of type TCP_TYPE_CLIENT or TCP_TYPE_CLIENT_SERVER can call the **tcp_connect** function.

Return Value

The **tcp_connect** function returns __TRUE when the connection establishment procedure has been started successfully. Otherwise, the function returns __FALSE.

See Also

tcp_abort, tcp_close, tcp_listen

Example

```
#include <RTL.h>

U8 tcp_soc;

U16 tcp_callback (U8 soc, U8 event, U8 *ptr, U16 par) {
  /* This function is called on TCP event */
   ..
  return (0);
}

void main (void) {
  U8 rem_ip[4] = {192,168,1,110};

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  tcp_soc = tcp_get_socket (TCP_TYPE_SERVER, 0, 30,
tcp_callback);
  if (tcp_soc != 0) {
    /* Start Connection */
    tcp_connect (tcp_soc, rem_ip, 80, 1000);
  }
  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

tcp_get_buf

Summary

```
#include <rtl.h>

U8* tcp_get_buf (
    U16 size );    /* Number of bytes to be sent. */
```

Description

The **tcp_get_buf** function allocates memory for the TCP send buffer into which your application can write the outgoing data packet. The argument *size* specifies the number of data bytes that the application wants to send.

After the TCP frame has been sent and an acknowledgement has been received from the remote host, TCPnet automatically de-allocates the memory used by the send buffer.

A default Maximum Segment Size of 1460 bytes is defined at startup. However, when establishing a connection with a remote machine, TCPnet might negotiate a different (smaller) value for the Maximum Segment Size.

The **tcp_get_buf** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- Your application must call the **tcp_get_buf** function each time it wants to send a TCP data packet.
- The size of the allocated memory must not exceed the TCP Maximum Segment Size (1460 bytes).
- Writing more data than the allocated *size* of the data buffer overwrites the Memory Manager Block links and causes TCPnet to crash.

Return Value

The **tcp_get_buf** function returns a pointer to the allocated memory. If memory allocation fails, TCPnet calls the **sys_error** function with the code ERR_MEM_ALLOC.

See Also

tcp_max_dsize, **tcp_send**

Example

```
#include <rtl.h>
#include <string.h>

void send_datalog () {
  U8 *sendbuf;
  U16 maxlen;

  maxlen = tcp_max_dlen (tcp_soc);
  sendbuf = tcp_get_buf (maxlen);
  memcpy (sendbuf, data_buf, maxlen);
  tcp_send (tcp_soc, sendbuf, maxlen);
}
```

tcp_get_socket

Summary

```
#include <rtl.h>

U8 tcp_get_socket (
    U8   type,              /* Type of TCP socket. */
    U8   tos,               /* Type Of Service. */
    U16  tout,              /* Idle timeout period before
disconnecting. */
    U16  (*listener)(      /* Function to call when a TCP event
occurs. */
        U8  socket,         /* Socket handle of the local machine. */
        U8  event,          /* TCP event such as connect, or close. */
        U8* ptr,            /* Pointer to IP address of remote
machine, */
                            /*   or to buffer containing received
data. */
        U16 par ));         /* Port number of remote machine, or
length */
                            /* of received data. */
```

Description

The **tcp_get_socket** function allocates a free TCP socket. The function initializes all the state variables of the TCP socket to the default state.

The argument *type* specifies the type of the TCP socket.

Socket Type	Description
TCP_TYPE_SERVER	The TCP socket is able to listen on the TCP port for incoming connections.
TCP_TYPE_CLIENT	The TCP socket is able to initiate a connection to a remote server.
TCP_TYPE_CLIENT_SERVER	The TCP socket is able to listen to incoming connections and to initiate a connection to a remote server.
TCP_TYPE_DELAY_ACK	This attribute improves the performance for applications sending large amounts of data like HTTP server. You can combine this attribute with the other attributes using the bitwise-or (|) operation.
TCP_TYPE_FLOW_CTRL	The TCP socket is able to control TCP Data Flow. You can combine this attribute with the other attributes using the bitwise-or (|) operation.
TCP_TYPE_KEEP_ALIVE	The TCP socket is able to send keep-alive packets when timeout expires. You can combine this attribute with the other attributes using the bitwise-or (|) operation.

The argument *tos* specifies the IP Type Of Service. The most common value for *tos* is 0.

The argument *tout* specifies the idle timeout in seconds. The TCP connection is supervised by the keep alive timer. When the connection has been idle for more than *tout* seconds, TCPnet disconnects the the TCP connection or sends a keep-alive packet if TCP_TYPE_KEEP_ALIVE attribute is set.

The argument ***listener*** is the event listening function of the TCP socket. TCPnet calls the ***listener*** function whenever a TCP event occurs. The arguments to the ***listener*** function are:

- *socket*: TCP socket handle of the local machine.
- *event*: Specifies the type of event that occurred as shown in the table

below.

- *ptr*: If event is TCP_EVT_DATA, *ptr* points to a buffer containing the received data. For all other events, *ptr* points to the IP address of the remote machine.
- *par*: If event is TCP_EVT_DATA, *par* specifies the number of bytes of data received. For all other events, *par* specifies the port number used by the remote machine.

TCPnet uses the return value of the callback function ***listener*** only when the event is TCP_EVT_CONREQ. It uses the return value to decide whether to accept or reject an incoming connection when the TCP socket is listening. If the ***listener*** function returns 1, TCPnet accepts the incoming connection. If the ***listener*** function returns 0, TCPnet rejects the incoming connection. You can thus define the ***listener*** function to selectively reject incoming connections from particular IP addresses.

Event Type	Description
TCP_EVT_CONREQ	A Connect Request has been received from a remote client that wants to connect to the server running on TCPnet.
TCP_EVT_CONNECT	The TCP socket has connected to the remote machine.
TCP_EVT_CLOSE	The TCP connection has been properly closed.
TCP_EVT_ABORT	The TCP connection has been aborted.
TCP_EVT_ACK	Acknowledgement has been received from the remote host for the previously sent data.
TCP_EVT_DATA	A TCP data packet has been received.

The **tcp_get_socket** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **tcp_get_socket** function before any other function calls to the TCP socket.
- You must define the ***listener*** function to use with the TCP socket.
- You must use the **TCP_TYPE_KEEP_ALIVE** attribute for a longstanding connection.

Return Value

The **tcp_get_socket** function returns the handle of the allocated TCP socket. If the function could not allocate a socket, it returns 0.

See Also

tcp_connect, tcp_listen, tcp_release_socket, tcp_reset_window

Example

```
#include <rtl.h>

U8 tcp_soc;

U16 tcp_callback (U8 soc, U8 event, U8 *ptr, U16 par) {
  /* This function is called on TCP event */
   ..
  switch (event) {
    case TCP_EVT_CONREQ:
      /* Remote host is trying to connect to our TCP socket. */
      /* 'ptr' points to Remote IP, 'par' holds the remote port.
*/
        ..
      /* Return 1 to accept connection, or 0 to reject connection
*/
      return (1);
    case TCP_EVT_ABORT:
      /* Connection was aborted */
       ..
      break;
    case TCP_EVT_CONNECT:
      /* Socket is connected to remote peer. */
       ..
      break;
    case TCP_EVT_CLOSE:
      /* Connection has been closed */
       ..
      break;
    case TCP_EVT_ACK:
      /* Our sent data has been acknowledged by remote peer */
       ..
      break;
    case TCP_EVT_DATA:
      /* TCP data frame has been received, 'ptr' points to data
*/
      /* Data length is 'par' bytes */
       ..
      break;
  }
  return (0);
}

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  tcp_soc = tcp_get_socket (TCP_TYPE_SERVER, 0, 30,
tcp_callback);
  if (tcp_soc != 0) {
    /* Start listening on TCP port 80 */
    tcp_listen (tcp_soc, 80);
  }

  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

tcp_get_state

Summary

```
#include <rtl.h>

U8 tcp_get_state (
    U8 socket );    /* TCP socket to get the state of. */
```

Description

The **tcp_get_state** function determines the current state of the TCP socket. Your application can use the **tcp_get_state** function to monitor the progress when establishing a connection or when closing the connection. The most useful state values are TCP_STATE_CLOSED, TCP_STATE_LISTEN, and TCP_STATE_CONNECT. The argument *socket* specifies the handle of the socket to get the state of.

The **tcp_get_state** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **tcp_get_state** function returns the current state of the TCP socket.

State	Description
TCP_STATE_FREE	Socket is free and not allocated yet. The function cannot return this value.
TCP_STATE_CLOSED	Socket is allocated to an application but the connection is closed.
TCP_STATE_LISTEN	Socket is listening for incoming connections.
TCP_STATE_SYN_REC	Socket has received a TCP packet with the flag SYN set.
TCP_STATE_SYN_SENT	Socket has sent a TCP packet with the flag SYN set.
TCP_STATE_FINW1	Socket has sent a FIN packet, to start the closing of the connection.
TCP_STATE_FINW2	Socket has received acknowledgement from the remote machine for the FIN packet it sent out from the local machine. Socket is now waiting for a FIN packet from the remote machine.
TCP_STATE_CLOSING	Socket has received a FIN packet from the remote machine independently
TCP_STATE_LAST_ACK	Socket is waiting for the last ACK packet to the FIN packet it sent out.
TCP_STATE_TWAIT	Socket is waiting on a 2 ms timeout before closing the connection.
TCP_STATE_CONNECT	Socket has established a TCP connection. You can transfer data only in this state.

See Also

tcp_check_send

Example

```
#include <rtl.h>

void send_data () {
  U8 remip[4] = {192,168,1,100};
  U8 *sendbuf;

  switch (tcp_get_state (socket_tcp)) {
    case TCP_STATE_FREE:
    case TCP_STATE_CLOSED:
      /* Connection idle, send Connect Request. */
      tcp_connect (socket_tcp, Rem_IP, 1001, 0);
      break;
    case TCP_STATE_CONNECT:
      /* We are connected, send command to remote peer. */
      if (tcp_check_send (socket_tcp)) {
        /* OK, socket is ready to send data. */
        sendbuf = tcp_get_buf (2);
        sendbuf[0] = BLINKLED;
        sendbuf[1] = p2;
        tcp_send (socket_tcp, sendbuf, SENDLEN);
      }
      break;
  }
}
```

tcp_listen

Summary

```
#include <rtl.h>

BOOL tcp_listen (
    U8  socket,         /* TCP socket to listen with. */
    U16 locport );      /* TCP port number to listen at. */
```

Description

The **tcp_listen** function opens a socket for incoming connections by causing the socket to listen at a local TCP port. The argument *socket* specifies the handle for the socket to listen with on the local machine. The argument *locport* specifies the TCP port number to listen at.

The **tcp_listen** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- TCPnet server applications (such as Telnet and HTTP server) must open a TCP socket for listening.
- Only sockets of type TCP_TYPE_SERVER or TCP_TYPE_CLIENT_SERVER can call the **tcp_listen** function.

Return Value

The **tcp_listen** function returns __TRUE if the TCP socket has started listening without errors. Otherwise, the function returns __FALSE.

See Also

tcp_abort, **tcp_close**, **tcp_connect**

Example

```
#include <rtl.h>

U8 tcp_soc;

U16 tcp_callback (U8 soc, U8 event, U8 *ptr, U16 par) {
  /* This function is called on TCP event */
   ..
  switch (event) {
    case TCP_EVT_CONREQ:
      /* Remote host is trying to connect to our TCP socket. */
      /* 'ptr' points to Remote IP, 'par' holds the remote port.
*/
       ..
      /* Return 1 to accept connection, or 0 to reject connection
*/
      return (1);
    case TCP_EVT_ABORT:
      /* Connection was aborted */
       ..
      break;
    case TCP_EVT_CONNECT:
      /* Socket is connected to remote peer. */
       ..
      break;
    case TCP_EVT_CLOSE:
      /* Connection has been closed */
       ..
      break;
    case TCP_EVT_ACK:
      /* Our sent data has been acknowledged by remote peer */
       ..
      break;
    case TCP_EVT_DATA:
      /* TCP data frame has been received, 'ptr' points to data
*/
      /* Data length is 'par' bytes */
       ..
      break;
  }
  return (0);
}

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  tcp_soc = tcp_get_socket (TCP_TYPE_SERVER, 0, 30,
tcp_callback);
  if (tcp_soc != 0) {
    /* Start listening on TCP port 80 */
    tcp_listen (tcp_soc, 80);
  }

  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

tcp_max_dsize

Summary

```
#include <rtl.h>

U16 tcp_max_dsize (
    U8 socket );    /* TCP socket to get the MSS of. */
```

Description

The **tcp_max_dsize** function determines the maximum number of data bytes that can be sent in the TCP packet (Maximum Segment Size). The argument *socket* specifies the handle of the TCP socket to get the Maximum Segment Size of.

A default Maximum Segment Size of 1460 bytes is defined at startup. However, when establishing a connection with a remote machine, TCPnet might negotiate a different (smaller) value for the Maximum Segment Size.

The **tcp_max_dsize** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **tcp_max_dsize** function returns the maximum number of data bytes that can be sent in each TCP packet.

See Also

tcp_get_buf, tcp_send

Example

```
#include <rtl.h>
#include <string.h>

void send_datalog () {
  U8 *sendbuf;
  U16 maxlen;

  maxlen = tcp_max_dsize (tcp_soc);
  sendbuf = tcp_get_buf (maxlen);
  memcpy (sendbuf, data_buf, maxlen);
  tcp_send (tcp_soc, sendbuf, maxlen);
}
```

tcp_release_socket

Summary

```
#include <rtl.h>

BOOL tcp_release_socket (
    U8 socket );     /* TCP socket to release. */
```

Description

The **tcp_release_socket** function de-allocates the memory used by the TCP socket. The argument *socket* specifies the handle of the socket to be released.

The **tcp_release_socket** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **tcp_release_socket** function when you do not need the TCP socket any longer.
- After calling **tcp_release_socket** the socket is free to use by another process.

Return Value

The **tcp_release_socket** function returns __TRUE if the TCP socket has been released successfully. Otherwise, the function returns __FALSE.

See Also

tcp_abort, tcp_close, tcp_get_socket

Example

```
#include <rtl.h>

void disconnect_tcp (U8 tcp_soc) {
   ..
  /* This TCP connection needs to close immediately */
  tcp_abort (tcp_soc);
  /* Socket will not be needed anymore */
  tcp_release_socket (tcp_soc);
}
```

tcp_reset_window

Summary

```
#include <rtl.h>

void tcp_reset_window (
    U8 socket );     /* TCP socket to reset the window size. */
```

Description

The **tcp_reset_window** function resets the TCP window size to a default value defined with TCP_DEF_WINSIZE macro. The argument *socket* specifies the handle of the socket to reset the window size of.

This function can only be used with sockets that have a TCP flow control enabled. To enable a TCP flow control for the socket, **tcp_get_socket** function has to be called with **TCP_TYPE_FLOW_CTRL** attribute set. This attribute enables using a **Sliding Window** protocol.

In **Flow Control** mode, each received data packet reduces the receiving Window Size by the number of data bytes received in the packet. Soon the window size becomes very small or 0, remote host stops sending data and waits for a window update. As soon as the received data is processed, we can call a **tcp_reset_window** function to reopen the receiver window for further incoming data.

Depending on the context, from where this function was called, it does the following actions:

- resets the window size of the *socket* and returns if called from the callback function. The window size is actually changed in the acknowledge packet generated by TCPnet when the callback function returns.
- resets the window size and sends out a **Window Update** packet if called from the other part of the user application.
- does nothing if the *socket* is not in TCP_STATE_CONNECT state and the TCP_TYPE_FLOW_CTRL attribute not set for the *socket*.

The **tcp_reset_window** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

Return Value

The **tcp_reset_window** function does not return any value.

See Also

tcp_get_socket

Example

```
#include <rtl.h>

U8 tcp_soc;
U8 buf[TCP_DEF_WINSIZE];
U32 head, tail;

void send_to_uart (void) {
  /* Send the data received from TCP to UART. */

  if (uart_busy () || head == tail) {
    /* Do nothing if UART is busy or when 'buf' is empty. */
    return;
  }

  send_uart (buf[tail++]);
  if (tail == head) {
    /* The 'buf' is empty, all bytes sent out to UART. *.
    tail = 0;
    head = 0;
    tcp_reset_window (tcp_soc);
  }
}


U16 tcp_callback (U8 soc, U8 event, U8 *ptr, U16 par) {
  /* This function is called on TCP event */
   ..
  switch (event) {
    case TCP_EVT_CONREQ:
      /* Remote host is trying to connect to our TCP socket. */
      /* 'ptr' points to Remote IP, 'par' holds the remote port.
*/
      /* Return 1 to accept connection, or 0 to reject connection
*/
      return (1);
    case TCP_EVT_ABORT:
      /* Connection was aborted */
      tcp_soc = 0;
      break;
    case TCP_EVT_CONNECT:
      /* Socket is connected to remote peer. */
      tcp_soc = soc;
      break;
    case TCP_EVT_CLOSE:
      /* Connection has been closed */
      tcp_soc = 0;
      break;
    case TCP_EVT_ACK:
      /* Our sent data has been acknowledged by remote peer */
      break;
    case TCP_EVT_DATA:
      /* TCP data frame has been received, 'ptr' points to data
*/
      /* Data length is 'par' bytes */
      memcpy (&buf[head], ptr, par);
      head += par;
      break;
  }
  return (0);
}

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  tcp_soc = tcp_get_socket (TCP_TYPE_SERVER | TCP_TYPE_FLOW_CTRL,
                            0, 30, tcp_callback);
  if (tcp_soc != 0) {
    /* Start listening on TCP port 8080 */
    tcp_listen (tcp_soc, 8080);
  }

  head = 0;
  tail = 0;
  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
    send_to_uart ();
     ..
  }
}
```

tcp_send

Summary

```
#include <rtl.h>

BOOL tcp_send (
    U8   socket,     /* TCP socket to send the data packet from. */
    U8*  buf,        /* Pointer to buffer containing the data to 
send. */
    U16  dlen );     /* Number of bytes of data to send. */
```

Description

The **tcp_send** function sends the data packet to a remote machine. The argument *socket* specifies the socket handle to use for communication on the local machine.

The argument *buf* points to the constructed TCP data packet. The argument *dlen* specifies the number of bytes in the data packet to send.

The **tcp_send** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must allocate the memory using **tcp_get_buf** before calling **tcp_send**.
- The socket must already be open and connected for communication.

Return Value

The **tcp_send** function returns __TRUE when the TCP packet has been sent successfully. Otherwise, the function returns __FALSE.

See Also

tcp_check_send, tcp_get_buf, tcp_max_dsize

Example

```
#include <rtl.h>
#include <string.h>

void send_datalog () {
  U8 *sendbuf;
  U16 maxlen;

  maxlen = tcp_max_dsize (tcp_soc);
  sendbuf = tcp_get_buf (maxlen);
  memcpy (sendbuf, data_buf, maxlen);
  tcp_send (tcp_soc, sendbuf, maxlen);
}
```

tftp_fclose

Summary

```
#include <net_config.h>

void tftp_fclose (
    FILE* file );     /* Pointer to the file to close. */
```

Description

The **tftp_fclose** function closes the file identified by the *file* stream pointer in the function argument.

The **tftp_fclose** function is in the TFTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **tftp_fclose** function does not return any value.

See Also

tftp_fopen, **tftp_fread**, **tftp_fwrite**

Example

```
void tftp_fclose (FILE *file) {
  /* Close the file, opened for reading or writing. This function
is */
  /* called, when the TFTP Session is closing.
  */
  fclose (file);
}
```

tftp_fopen

Summary

```
#include <net_config.h>

void* tftp_fopen (
    U8* fname,      /* Pointer to name of file to open. */
    U8* mode );     /* Pointer to mode of operation. */
```

Description

The **tftp_fopen** function opens a file for reading or writing. The argument *fname* specifies the name of the file to open. The *mode* defines the type of access permitted for the file. It can have one of the following values.

Mode	Description
"r"	Opens the file for reading. If the file does not exist, **fopen** fails.
"w"	Opens an empty file for writing if the file does not exist. If the file already exists, its contents are cleared.

The **tftp_fopen** function is in the TFTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **tftp_fopen** function returns a pointer to the opened file. The function returns NULL if it cannot open the file.

See Also

tftp_fclose, tftp_fread, tftp_fwrite

Example

```
void *tftp_fopen (U8 *fname, U8 *mode) {
  /* Open filename fname for reading or writing. */
  return (fopen(fname, mode));
}
```

tftp_fread

Summary

```
#include <net_config.h>

U16  tftp_fread (
    FILE* file,      /* Pointer to the file to read from.     */
    U32   fpos,      /* Position in the file, to read from.   */
    U8*   buf );     /* Pointer to buffer, to store the data. */
```

Description

The **tftp_fread** reads a block of data from the file identified by the *file* stream pointer in the function argument. The argument *fpos* is an offset relative to the beginning of the stream. The offset specifies the file position to read from. The argument *buf* is a pointer to the buffer where the function stores the read data.

The **tftp_fread** function is in the TFTP_uif.c module. The prototype is defined in net_config.h.

note

- The **tftp_fread** function must read 512 bytes. The TFTP Server stops reading and closes the TFTP session if the return value is less than 512 bytes.

Return Value

The **tftp_fread** function returns the number of bytes read from the file.

See Also

tftp_fclose, tftp_fopen, tftp_fwrite

Example

```
U16 tftp_fread (FILE *file, U32 fpos, U8 *buf)
  /* Read file data to sending buffer. Max. data length is */
  /* 512 bytes. Return number of bytes copied to buffer.   */

  if (fpos != ftell (file)) {
    /* Rewind file position pointer back for 512 bytes. */
    fseek (file, fpos, SEEK_SET);
  }
  return (fread (buf, 1, 512, file));
}
```

tftp_fwrite

Summary

```
#include <net_config.h>

U16 tftp_fwrite (
    FILE* file,     /* Pointer to the file to write to. */
    U8*   buf,      /* Pointer to the buffer containing the data.
*/
    U16   len );    /* Number of bytes to write. */
```

Description

The **tftp_fwrite** function writes a block of data to the file identified by the *file* stream pointer. The argument *buf* points to the buffer containing the data that is to be written to the file. The argument *len* specifies the number of bytes to write to the file.

The **tftp_fwrite** function is in the TFTP_uif.c module. The prototype is defined in net_config.h.

Return Value

The **tftp_fwrite** function returns the number of bytes written to the file.

See Also

tftp_fclose, **tftp_fopen**, **tftp_fread**

Example

```
U16 tftp_fwrite (FILE *file, U8 *buf, U16 len) {
  /* Write data to file. Return number of bytes actually written.
When */
  /* returned number of bytes 'len' is less than 512, TFTP
connection  */
  /* will be closed.
    */
  return (fwrite (buf, 1, len, file));
}
```

timer_tick

Summary

```
#include <rtl.h>

void timer_tick (void);
```

Description

The **timer_tick** function sets the timer tick flag, which the TCPnet system uses to measure time.

The **timer_tick** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- The TCPnet system does not use any CPU timers. Hence, you must generate the timing events by periodically calling the **timer_tick** function, for example by using a timer interrupt function.
- Since all the TCPnet timeouts are very slow, you can also generate timer events by polling for timer overflow.
- You must configure the parameter TICK_INTERVAL in net_config.c to the period of your timer events, in milliseconds. You can set this to any value between 10 and 200.
- It is better to set TICK_INTERVAL to a higher value because low value for TICK_INTERVAL results in more frequent calls to system timer update function, which reduces the performance of the TCPnet system.

Return Value

The **timer_tick** function does not return any value.

See Also

main_TcpNet

Example

```
#include <RTL.h>

static void timer_poll () {

  if (T1IR & 1) {
    T1IR = 1;
    /* Timer tick every 100 ms */
    timer_tick ();
  }
}


void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  while (1);
    /* Poll the system tick timer */
    timer_poll ();
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

tnet_cbfunc

Summary

```
#include <net_config.h>

U16 tnet_cbfunc (
    U8   code,         /* Type of message requested by the telnet
server. */
    U8* buf,           /* Location where to write the requested
message. */
    U16 buflen );     /* Number of bytes in the output buffer. */
```

Description

The **tnet_cbfunc** function provides the connection and login messages to the telnet server running on TCPnet, when requested. The telnet server sends these messages to the telnet client.

The argument *code* specifies the type of message that the telnet server requires. This is shown in the table.

Code	Message	Description
0	Initial header	To inform the user that the user's telnet client has connected to the telnet server running on TCPnet.
1	Prompt string	To inform the user that the telnet server is ready and waiting for a command.
2	Login header	Displayed only when user authentication is enabled.
3	Login username	To inform the user to enter the username.
4	Login password	To inform the user to enter the password.
5	Incorrect login	To inform the user that the login is incorrect.
6	Timeout login	To inform the user that the login has timed out.
7	Unsolicited message	To write unsolicited messages from the high-layer user application (for example a basic interpreter).

The argument *buf* points to the output buffer where the **tnet_cbfunc** must write the message. The argument *buflen* specifies the length of the output buffer in bytes.

The **tnet_cbfunc** function is part of RL-TCPnet. The prototype is defined in net_config.h. You must customize the function in telnet_uif.c.

note

- The length of the output buffer, buflen, might vary because buffer length is determined by the TCP socket Maximum Segment Size (MSS) negotiation. The buffer length is normally around 1400 bytes for local LAN. But this can be reduced to 500 bytes or even less.
- If the **tnet_cbfunc** function writes more bytes than buflen into the output buffer, then a system crash resulting from corruption of memory link pointers is highly likely.
- The telnet server does not automatically expand the carriage return (CR) character. The **tnet_cbfunc** function must write the carriage return and line feed (LF) characters into the the buffer to indicate the end of each message.

Return Value

The **tnet_cbfunc** function returns the number of bytes written to the output buffer.

See Also

tnet_process_cmd

Example

```
U16 tnet_cbfunc (U8 code, U8 *buf, U16 buflen) {
  U16 len = 0;

  switch (code) {
    case 0:
      /* Write initial header after login. */
      len = str_copy (buf, tnet_header);
      break;
    case 1:
      /* Write a prompt string. */
      len = str_copy (buf, "\r\nMcb2100> ");
      break;
    case 2:
      /* Write Login header. */
      len = str_copy (buf, CLS "\r\nKeil Embedded Telnet Server
V1.00,"
                             " please login...\r\n");
      break;
    case 3:
      /* Write 'username' prompt. */
      len = str_copy (buf, "\r\nUsername: ");
      break;
    case 4:
      /* Write 'Password' prompt. */
      len = str_copy (buf, "\r\nPassword: ");
      break;
    case 5:
      /* Write 'Login incorrect'.message. */
      len = str_copy (buf, "\r\nLogin incorrect");
      break;
    case 6:
      /* Write 'Login Timeout' message. */
      len = str_copy (buf, "\r\nLogin timed out after 60
seconds.\r\n");
      break;
    case 7:
      /* Write Unsolicited messages from the Application Layer
above. */
      len = sprintf ((S8 *)buf, "\r\nUnsolicited message nr.
%d\r\n",
                                unsol_msg++);
      break;
  }
  return (len);
}
```

tnet_ccmp

Summary

```
#include <net_config.h>

BOOL tnet_ccmp (
    U8* buf,      /* Pointer to the message from the telnet
client. */
    U8* cmd );    /* String containing the command to compare
with.  */
```

Description

The **tnet_ccmp** function compares the command part of the message in the buffer *buf* with the string *cmd*.

The **tnet_ccmp** function is in the RL-TCPnet library. The prototype is defined in net_config.h.

note

- The **tnet_ccmp** function is similar to the standard C string function **strcmp**. The difference is that **tnet_ccmp** only compares the first string, in *buf*, that is either terminated by the NULL character or followed by a space character. Hence, you must pass the message from the telnet client using *buf* and not using *cmd*.
- All the characters in the string *cmd* must be in capital letters because the string in *buf* also has only capital letters. This is due to an internal conversion before calling the **tnet_ccmp** function.

Return Value

The **tnet_ccmp** function returns __TRUE if there is a match. It returns __FALSE, if there is no match.

See Also

tnet_process_cmd, tnet_set_delay

Example

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {
  U16 len,val,ch;

  /* Simple Command line parser */
  len = strlen (cmd);

  if (tnet_ccmp (cmd, "BYE") == __TRUE) {
    /* 'BYE' command, send message and disconnect */
    len = str_copy (buf, "\r\nDisconnect...\r\n");
    /* Hi bit of return value is a disconnect flag */
    return (len | 0x8000);
  }

  if (tnet_ccmp (cmd, "ADIN") == __TRUE) {
    /* 'ADIN' command received */
    if (len >= 6) {
      sscanf (cmd+5,"%d",&ch);
      val = AD_in (ch);
      len = sprintf (buf,"\r\n ADIN %d = %d",ch,val);
      return (len);
    }
  }

  if (tnet_ccmp (cmd, "HELP") == __TRUE || tnet_ccmp (cmd, "?")
== __TRUE) {
    /* 'HELP' command, display help text */
    len = str_copy (buf,tnet_help);
    return (len);
  }
  /* Unknown command, display message */
  len = str_copy  (buf, "\r\n==> Unknown Command: ");
  len += str_copy (buf+len, cmd);
  return (len);
}
```

tnet_get_info

Summary

```
#include <net_config.h>

void tnet_get_info (
    REMOTEM* info );     /* Pointer to structure to copy     */
                         /* remote machine information into. */
```

Description

The **tnet_get_info** function provides information about the remote machine that has connected to the TCPnet telnet server. The argument *info* points to a structure into which the function writes the IP address and MAC address of the remote machine.

The **tnet_get_info** function is in the RL-TCPnet library. The prototype is defined in net_config.h.

note

- You can use the **tnet_get_info** function to restrict which machines are allowed to perform system changes.
- For PPP or SLIP links, the MAC address is set to 00-00-00-00-00-00.

Return Value

The **tnet_get_info** function does not return any value.

See Also

tnet_process_cmd

Example

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {
  REMOTEM rm;
  U16 len;

  /* Simple Command line parser */
  len = strlen (cmd);

  if (tnet_ccmp (cmd, "RINFO") == __TRUE) {
    /* Display Remote Machine IP and MAC address. */
    tnet_get_info (&rm);
    len  = sprintf (buf,     "\r\n Remote IP : %d.%d.%d.%d",

rm.IpAdr[0],rm.IpAdr[1],

rm.IpAdr[2],rm.IpAdr[3]);
    len += sprintf (buf+len, "\r\n Remote MAC:
%02X-%02X-%02X-%02X-%02X-%02X",

rm.HwAdr[0],rm.HwAdr[1],

rm.HwAdr[2],rm.HwAdr[3],

rm.HwAdr[4],rm.HwAdr[5]);
    return (len);
  }
   ..
  return (len);
}
```

tnet_msg_poll

Summary

```
#include <net_config.h>

BOOL tnet_msg_poll (
    U8 session);         /* Telnet Session handle. */
```

Description

The **tnet_msg_poll** function polls the upper-layer user application for unsolicited messages. The Telnet session handle identified with parameter *session* can be used to identify the remote host when several Telnet connections are active at the same time.

If this function returns **__TRUE** a telnet callback function is called immediatelly to get the message.

Return Value

The **tnet_msg_poll** function returns __TRUE when the upper-layer user application has a pending message to send. Otherwise, the function returns __FALSE.

See Also

tnet_cbfunc

Example

```
BOOL tnet_msg_poll (U8 session) {
  /* Poll for Unsolicited Messages from Server. Return TRUE to
the Telnet */
  /* Server when messages are available for the telnet 'session'.
       */
  if (session != 1) {
    /* Only 1st session may receive Unsolicited Messages. */
    return (__FALSE);
  }
  if (send_msg == __TRUE) {
    send_msg = __FALSE;
    /* Yes, message is available. */
    return (__TRUE);
  }
  /* No messages available. */
  return (__FALSE);
}
```

tnet_process_cmd

Summary

```
#include <net_config.h>

U16 tnet_process_cmd (
    U8*   cmd,        /* Pointer to command string from the telnet
client. */
    U8*   buf,        /* Location where to write the return message.
*/
    U16   buflen,     /* Number of bytes in the output buffer. */
    U32* pvar );      /* Pointer to a storage variable. */
```

Description

The **tnet_process_cmd** function processes and executes the command requested by the telnet client. The telnet server running on TCPnet calls the **tnet_process_cmd** function when it receives the consecutive *Carriage Return* (CR) and *Line Feed* (LF) character sequence from the telnet client (this is usually produced by the user pressing Enter on the telnet client terminal).

The argument *cmd* points to the message containing the command that is received from the telnet client. The argument *buf* points to the output buffer where the **tnet_process_cmd** must write the message to be returned to the telnet client. The argument *buflen* specifies the length of the output buffer in bytes.

The argument *pvar* is a pointer to a variable that never gets altered by the Telnet Server. You can use **pvar* as a repeat counter or simply to distinguish between different calls of the **tnet_process_cmd** function.

The **tnet_process_cmd** function is part of RL-TCPnet. The prototype is defined in net_config.h. You must customize the function in telnet_uif.c.

note

- The length of the output buffer, buflen, might vary because buffer length is determined by the TCP socket Maximum Segment Size (MSS) negotiation. The buffer length is normally around 1400 bytes for local LAN. But this can be reduced to 500 bytes or even less.
- If the **tnet_process_cmd** function writes more bytes than buflen into the output buffer, then a system crash resulting from corruption of memory link pointers is highly likely.
- The telnet server does not automatically expand the CR character. The **tnet_process_cmd** function must write the CR and LF characters into the the buffer to indicate the end of each message.
- The argument *pvar* is private to each Telnet Session. The Telnet Server clears the data in the *pvar* pointer, to 0, before the **tnet_process_cmd** function is called for the first time in each session.

Return Value

The **tnet_process_cmd** function returns the number of bytes written to the output buffer. It also encodes the values of the repeat flag and the disconnect flag into the return value.

If bit 14 (repeat flag) of the return value is set to 1, the telnet server running on TCPnet calls the **tnet_process_cmd** function again with the argument *cmd* and storage variable **pvar* of the same value. The function **tnet_process_cmd** can then enter more data into the buffer *buf*.

If bit 15 (disconnect flag) of the return value is set to 1, the telnet server disconnects the telnet session.

See Also

tnet_cbfunc, tnet_ccmp

Example

```
U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {
  U32 len,val,ch;

  /* Simple Command line parser */
  len = strlen (cmd);

  if (tnet_ccmp (cmd, "BYE") == __TRUE) {
    /* 'BYE' command, send message and disconnect */
    len = str_copy (buf, "\r\nDisconnect...\r\n");
    /* Hi bit of return value is a disconnect flag */
    return (len | 0x8000);
  }

  if (tnet_ccmp (cmd, "ADIN") == __TRUE) {
    /* 'ADIN' command received */
    if (len >= 6) {
      sscanf (cmd+5,"%d",&ch);
      val = AD_in (ch);
      len = sprintf (buf,"\r\n ADIN %d = %d",ch,val);
      return (len);
    }
  }

  if (tnet_ccmp (cmd, "HELP") == __TRUE || tnet_ccmp (cmd, "?")
== __TRUE) {
    /* 'HELP' command, display help text */
    len = str_copy (buf,tnet_help);
    return (len);
  }
  /* Unknown command, display message */
  len = str_copy  (buf, "\r\n==> Unknown Command: ");
  len += str_copy (buf+len, cmd);
  return (len);
}
```

tnet_set_delay

Summary

```
#include <net_config.h>

void tnet_set_delay (
    U16 dly );    /* Length of time to wait. */
```

Description

The **tnet_set_delay** function ensures that the telnet server does not call the **tnet_process_cmd** function until a certain time period expires. After the time interval, the telnet server can call the **tnet_process_cmd** function. The argument *dly* specifies the time period to wait, in number of timer ticks.

The **tnet_set_delay** function is in the RL-TCPnet library. The prototype is defined in net_config.h.

note

- For example, when monitoring an input channel, you can call the **tnet_set_delay** function from the **tnet_process_cmd** to continuously update the telnet client screen every 2 seconds. Pressing a key on the remote telnet client stops the monitoring process.

Return Value

The **tnet_set_delay** function does not return any value.

See Also

tnet_process_cmd

Example

```
typedef struct {
  U8 id;
  U8 nmax;
  U8 idx;
} MY_BUF;
#define MYBUF(p)        ((MY_BUF *)p)

U16 tnet_process_cmd (U8 *cmd, U8 *buf, U16 buflen, U32 *pvar) {
  U16 len;

  switch (MYBUF(pvar)->id) {
    case 0:
      /* On first call to this function, the value of '*pvar' is 
0 */
      break;

    case 1:
      /* Display measurements triggered by command 'MEAS' */
       ..
      return (len);

    case 2:
       /* Monitor analog input ADIN0. */
       len = sprintf (buf,"ADIN0 = %d",AD_in (0));
       /* Delay a call to this function for 2 seconds (20 * 
100ms). */
       tnet_set_delay (20);
       /* Request a repeated call, bit 14 is a repeat flag. */
       len |= 0x4000;
       return (len);
  }

  /* Simple Command line parser */
  if (tnet_ccmp (cmd, "ADMON") == __TRUE) {
    /* Monitor the value of ADIN0 constantly on a screen. */
    len = str_copy (buf, CLS);
    MYBUF(cmd)->id  = 2;
    return (len | 0x4000);
  }
   ..
  return (len);
}
```

tsk_lock

Summary

```
#include <rtl.h>

void tsk_lock (void);
```

Description

The **tsk_lock** function disables the RTX kernel timer interrupts and thereby disables task switching.

The **task_lock** function is in the RL-RTX library. The prototype is defined in rtl.h.

Note

- For the duration when the timer interrupts are disabled, the RTX kernel task scheduler is blocked, timeouts do not work, and the Round Robin task scheduling is also blocked. Hence, it is highly recommended that disabling of the RTX kernel timer interrupts is kept to a very short time period.

Return Value

The **tsk_lock** function does not return any value.

See Also

tsk_unlock

Example

```
#include <rtl.h>

void free_mem (void *ptr) {
  /* 'free()' is not reentrant. */
  tsk_lock ();
  free (ptr);
  tsk_unlock ();
}
```

tsk_unlock

Summary

```
#include <rtl.h>

void tsk_unlock (void);
```

Description

The **tsk_unlock** function enables the RTX kernel timer interrupts and thereby enables task switching if they had been disabled by the **tsk_lock** function.

The **tsk_unlock** function is in the RL-RTX library. The prototype is defined in rtl.h.

Return Value

The **tsk_unlock** function does not return any value.

See Also

tsk_lock

Example

```
#include <rtl.h>

void free_mem (void *ptr) {
  /* 'free()' is not reentrant. */
  tsk_lock ();
  free (ptr);
  tsk_unlock ();
}
```

udp_close

Summary

```
#include <rtl.h>

BOOL udp_close (
    U8 socket);    /* Socket handle to close. */
```

Description

The **udp_close** function closes the UDP *socket* identified in the function argument. After calling the **udp_close** function, the UDP socket cannot send or receive any data packet.

The **udp_close** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- After closing the UDP socket, you can reopen it using the **udp_open** function.
- The UDP socket still reserves memory after calling the **udp_close** function. Hence, if you do not need the socket, you must release the memory using the **udp_release_socket** function after calling **udp_close**.

Return Value

The **udp_close** function returns __TRUE if the UDP socket is successfully closed. Otherwise, the function returns __FALSE.

See Also

udp_open, udp_release_socket

Example

```
#include <rtl.h>

void disconnect_udp (U8 udp_soc) {
   ..
  /* This UDP connection is no longer needed */
  udp_close (udp_soc);
  udp_release_socket (udp_soc);
}
```

udp_get_buf

Summary

```
#include <rtl.h>

U8* udp_get_buf (
    U16 size);    /* Number of bytes to be sent. */
```

Description

The **udp_get_buf** function allocates memory for the UDP send buffer into which your application can write the outgoing data packet. The argument *size* specifies the number of data bytes that the application wants to send.

When the UDP frame has been sent, TCPnet automatically de-allocates the memory used by the send buffer.

The **udp_get_buf** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- Your application must call the **udp_get_buf** function each time it wants to send a UDP data packet.
- The total size of the allocated memory must not exceed the UDP Maximum Packet Size (1472 bytes).
- Writing more data than the allocated *size* of the data buffer overwrites the Memory Manager Block links and causes TCPnet to crash.

Return Value

The **udp_get_buf** function returns a pointer to the allocated memory. If memory allocation fails, TCPnet calls the **sys_error** function with the code ERR_MEM_ALLOC.

See Also

udp_send

Example

```
#include <rtl.h>
#include <string.h>

void send_data () {
  char udp_msg[] = {"Hello World!"};
  U8 remip[4] = {192,168,1,100};
  U8 *sendbuf;
  U16 len;

  len = strlen (udp_msg);
  sendbuf = udp_get_buf (len);
  str_copy (sendbuf, udp_msg);
  /* Send 'Hello World!' to remote peer */
  udp_send (udp_soc, remip, 1000, sendbuf, len);
}
```

udp_get_socket

Summary

```
#include <rtl.h>

U8 udp_get_socket (
    U8   tos,           /* Type Of Service. */
    U8   opt,           /* Option to calculate or verify the
checksum. */
    U16 (*listener)(   /* Function to call when TCPnet receives a
data packet. */
        U8  socket,     /* Socket handle of the local machine. */
        U8* remip,      /* Pointer to IP address of remote
machine. */
        U16 port,       /* Port number of remote machine. */
        U8* buf,        /* Pointer to buffer containing the
received data. */
        U16 len ));     /* Number of bytes in the received data
packet. */
```

Description

The **udp_get_socket** function allocates a free UDP socket. The function initializes all the state variables of the UDP socket to the default state.

The argument *tos* specifies the IP Type Of Service. The most common value for *tos* is 0. The argument *opt* specifies the checksum option as shown in the table.

Option Value	Description
UDP_OPT_SEND_CS	Calculate the UDP checksum for the packets to be sent.
UDP_OPT_CHK_CS	Check the UDP checksum on the received packets.

You can also set *opt* to UDP_OPT_SEND_CS|UDP_OPT_CHK_CS to enable both options. This is recommended for a more secure UDP connection. You can also disable both options by setting *opt* to 0 for fast system reaction time.

The argument ***listener*** is the event listening function of the UDP socket. TCPnet calls the ***listener*** function whenever a UDP data packet is received. The arguments to the ***listener*** function are:

- *socket*: UDP socket handle of the local machine.
- *remip*: pointer to the IP address of the remote machine.
- *port*: UDP port number on the remote machine.
- *buf*: pointer to buffer containing the received data.
- *len*: number of bytes in the received packet.

The **udp_get_socket** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must call the **udp_get_socket** function before any other function calls to the UDP socket.
- Ethernet packets are also protected by the ethernet CRC.
- Packets can be modified when passing through routers, proxy servers, gateways, etc.
- You must define the ***listener*** function to use with the UDP socket.

Return Value

The **udp_get_socket** function returns a handle to the allocated UDP socket. If a socket cannot be allocated, the function returns 0.

See Also

udp_close, udp_open, udp_release_socket

Example

```
#include <rtl.h>

U8 udp_soc;

U16 udp_callback (U8 socket, U8 *remip, U16 remport, U8 *buf, U16
len) {
  /* This function is called when UDP data is received */

  /* Process received data from 'buf' */
   ..
  return (0);
}

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  udp_soc = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS,
udp_callback);
  if (udp_soc != 0) {
    /* Open UDP port 1000 for communication */
    udp_open (udp_soc, 1000);
  }

  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

udp_mcast_ttl

Summary

```
#include <rtl.h>

BOOL udp_mcast_ttl (
    U8   socket,      /* UDP socket to set Time to Live. */
    U8* ttl);         /* TTL value - number of routers to cross. */
```

Description

The **udp_mcast_ttl** function sets the Time to Live value for multicast packets of an UDP socket identified with a parameter *socket*. A TTL value controls the number of routers the packet can cross before being expired. For multicast message the default TTL value is 1. If this function is not called a default TTL value of 1 is assumed for all transmitted Multicast UDP datagrams.

The argument *ttl* specifies the TTL value for the UDP socket.

The **udp_mcast_ttl** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- This TTL value is not used for **unicast** UDP datagrams.

Return Value

The **udp_mcast_ttl** function returns __TRUE when the TTL value successfully changed. Otherwise, the function returns __FALSE.

See Also

udp_get_socket, udp_open, udp_send

Example

```
 ..
udp_soc = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS,
udp_callback);
if (udp_soc != 0) {
  /* Open UDP port 1000 for communication */
  udp_open (udp_soc, 1000);
  /* Let UDP Multicast datagrams cross a router. */
  udp_mcast_ttl (udp_soc, 2);
}
```

udp_open

Summary

```
#include <rtl.h>

BOOL udp_open (
    U8  socket,       /* Socket handle to use for communication. */
    U16 locport);     /* Local port to use for communication. */
```

Description

The **udp_open** function opens the UDP socket identified by the argument *socket* for communication. The argument *locport* specifies the local port that is to be used to send and receive data packets.

The **udp_open** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- If you specify 0 for *locport*, TCPnet automatically allocates the first free UDP port for communication.

Return Value

The **udp_open** function returns __TRUE if the socket is opened successfully. Otherwise, the function returns __FALSE.

See Also

udp_close, **udp_get_socket**

Example

```
#include <rtl.h>

U8 udp_soc;

U16 udp_callback (U8 socket, U8 *remip, U16 remport, U8 *buf, U16
len) {
  /* This function is called when UDP data is received */

  /* Process received data from 'buf' */
   ..
  return (0);
}

void main (void) {

  init ();
  /* Initialize the TcpNet */
  init_TcpNet ();
  udp_soc = udp_get_socket (0, UDP_OPT_SEND_CS | UDP_OPT_CHK_CS,
udp_callback);
  if (udp_soc != 0) {
    /* Open UDP port 1000 for communication */
    udp_open (udp_soc, 1000);
  }

  while (1);
    /* Run main TcpNet 'thread' */
    main_TcpNet ();
     ..
  }
}
```

udp_release_socket

Summary

```
#include <rtl.h>

BOOL udp_release_socket (
    U8 socket);    /* Socket handle to release. */
```

Description

The **udp_release_socket** function releases the the *socket* identified in the function argument and de-allocates its memory.

The **udp_release_socket** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- Once the socket is released, TCPnet can allocate the socket to another process.
- You must call the **udp_release_socket** function when you do not need the UDP socket any longer.

Return Value

The **udp_release_socket** function returns __TRUE if the UDP socket is successfully released. Otherwise, the function returns __FALSE.

See Also

udp_close, **udp_get_socket**

Example

```
#include <rtl.h>

void disconnect_udp (U8 udp_soc) {
   ..
  /* This UDP connection is no longer needed */
  udp_close (udp_soc);
  udp_release_socket (udp_soc);
}
```

udp_send

Summary

```
#include <rtl.h>

BOOL udp_send (
    U8   socket,      /* UDP socket to send the data packet from. */
    U8*  remip,       /* Pointer to the IP address of the remote 
machine. */
    U16  remport,     /* Port number of remote machine to send the 
data to. */
    U8*  buf,          /* Pointer to buffer containing the data to 
send. */
    U16  dlen );      /* Number of bytes of data to send. */
```

Description

The **udp_send** function sends the data packet to a remote machine. The argument *socket* specifies the socket handle to use for communication on the local machine.

The argument *remip* points to a buffer containing the four octets that make up the ip address of the remote machine. The argument *remport* specifies the port on the remote machine to send the data packet to.

The argument *buf* points to the constructed UDP data packet. The argument *dlen* specifies the number of bytes in the data packet to send.

The **udp_send** function is in the RL-TCPnet library. The prototype is defined in rtl.h.

note

- You must allocate the memory using **udp_get_buf** before calling **udp_send** .
- The socket must already be open for communication before you can send data.
- The same UDP socket, using the same local port, can communicate with several remote machines using UDP ports. The user application must handle proper multiplexing of outgoing packets and demultiplexing of received packets.

Return Value

The **udp_send** function returns __TRUE when the UDP packet has been sent successfully. Otherwise, the function returns __FALSE.

See Also

udp_get_buf

Example

```
#include <rtl.h>
#include <string.h>

void send_data () {
  char udp_msg[] = {"Hello World!"};
  U8 remip[4] = {192,168,1,100};
  U8 *sendbuf;
  U16 len;

  len = strlen (udp_msg);
  sendbuf = udp_get_buf (len);
  str_copy (sendbuf, udp_msg);
  /* Send 'Hello World!' to remote peer */
  udp_send (udp_soc, remip, 1000, sendbuf, len);
}
```

ungetc

Summary

```
#include <stdio.h>

int ungetc (
    int  c,             /* character to un-get */
    FILE *stream);     /* file stream to unget from */
```

Description

The **ungetc** function stores the character *c* back into the input *stream* and clears the **EOF** indicator. Subsequent calls to **fgetc** and other stream input functions return *c*.

The **ungetc** function is in the RL-FlashFS library. The prototype is defined in stdio.h.

Note

- The **ungetc** function may be invoked only once between calls to functions that read from the stream. Subsequent calls to **ungetc** fail with an **EOF** return value.

Return Value

The <ungetc function returns the character *c* if successful. A return value of **EOF** indicates an error.

See Also

fgetc, fputc

Example

```
#include <rtl.h>
#include <stdio.h>

void main (void) {
  int c;
  FILE *fin;

  fin = fopen ("Test.txt","r");
  if (fin == NULL) {
    printf ("File not found!\n");
  }
  else {
    // Skip leading spaces
    while ((c = fgetc (fin)) == ' ');

    // Unget the first non-space
    ungetc (c, fin);
    ungetc (c, fin);  // This call fails with EOF

    // The file pointer is now positioned
    // to a non-space character
    fclose (fin);
  }
}
```

USB_Core

Summary

```
#include <usbuser.h>

void USB_Core  (
  void);
```

Description

The **USB_Core** function handles the USB core events sent by the host (such as the set interface event). The **USB_Core** function handles the USB core events that have been enabled in **usbcfg.h** or using the Configuration Wizard. See Configuration Parameters on page 1-34 for a list of USB core events and how to enable them.

The **USB_Core** function is part of the USB Core Device layer of the RL-USB Software Stack. You can Modify this function to provide your own special handling code for the USB core events.

USB_Core is a continuously running task.

Return Value

None.

See Also

USB_Device

Example

```
#include <usbuser.h>

void USB_Core (void) __task
{
…
#if USB_INTERFACE_EVENT
  if (evt & USB_EVT_SET_IF)  {            // Set Interface event
    // write your Set Interface event handling code here
  }
#endif
…
}
```

USB_Device

Summary

```
#include <usbuser.h>

void USB_Device  (
  void);
```

Description

The **USB_Device** function handles the USB device events sent by the host (such as the suspend event). The **USB_Device** function only handles the device events that you enable in **usbcfg.h** or using the Configuration Wizard. See Configuration Parameters on page 1-34 for a list of USB device events and how to enable them.

The **USB_Device** function is part of the USB Core Driver layer of the RL-USB Software Stack.

You can modify this function to provide your own special handling code for the USB device events.

USB_Device is a continuously running task and must not be invoked from the application.

Return Value

None.

See Also

USB_ISR

Example

```
#include <usbuser.h>

void USB_Device (void) __task  {
…
#if USB_SOF_EVENT
  if (evt & USB_EVT_SOF)  {    // Start of Frame event
    // write your Start-Of-Frame event handling code here
  }
#endif
…
}
```

USB_EndPoint*Number*

Summary

```
#include <usbuser.h>

void USB_EndPointNumber  (                    /* Number   [1 ..
15] */
  void);
```

Description

The **USB_EndPoint*Number*** function handles data transfers to and from an specific endpoint denoted by *Number*.

The function receives the requests from the main USB interrupt service routine **USB_ISR**. RL-USB enables the function only if the appropriate endpoint is enabled in **usbcfg.h**.

Modify or add code that suits the application to the **USB_EndPoint*Number*** function. When enabled, each **USB_EndPoint*Number*** function is a continuously running task.

The **USB_EndPoint*Number*** function is part of the USB Core Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

USB_ISR, USB_ReadEP, USB_WriteEP

Example

```
#include <usbuser.h>

void USB_EndPoint7 (void) __task  {

  for (;;)  {
    os_evt_wait_or (USB_EVT_IN, 0xFFFF);       /* Wait for IN
packet to arrive from host */
    GetInReport ();
    USB_WriteEP (0x81, &InReport, sizeof (InReport));
  }
}
```

USB_EndPoint0

Summary

```
#include <usbcore.h>

void USB_EndPoint0 (void);
```

Description

The **USB_EndPoint0** function handles all the requests sent to Endpoint0 (mainly done by the interrupt routine **USB_ISR**). This includes the standard and class specific setup requests and any data transfer using Endpoint0.

The **USB_EndPoint0** function is part of the USB Core Driver layer of the RL-USB software stack. There is no requirement to modify this function.

Return Value

None.

See Also

USB_EndPoint*Number*, USB_ISR

Example

It is not required to invoke **USB_EndPoint0** because it is a continuously running task.

USB_IRQHandler

Summary

```
#include <usb.h>

void USB_IRQHandler (
  void);
```

Description

The **USB_IRQHandler** function is just a generic name for USB Host OHCI controller Interrupt Handler Routine. As different platforms have different predefined IRQ Handler routine names this function might not be called USB_IRQHandler. This function must call USB Host library function USBH_OHCI_IRQHandler() as this function handles USB Host OHCI controller specific interrupts.

The **USB_IRQHandler** function is part of the **RL-USB-Host** Software Stack.

You can modify this function to suit different microcontroller interrupt system.

note

- This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

None.

Example

```
#include <usb.h>

void USB_IRQHandler (void) {
  USBH_OHCI_IRQHandler();
}
```

USB_ISR

Summary

```
#include <usbhw.h>

void USB_ISR  (
  void);
```

Description

The **USB_ISR** function receives all the USB requests from the controller hardware and sends the appropriate USB event to the corresponding task for processing.

It is not required to modify this function.

The **USB_ISR** function is part of the USB Device Controller Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

USB_Core, USB_Device, USB_EndPoint*Number*

Example

It is not required to invoke **USB_ISR**, since it is a continuously active interrupt service routine.

USB_ReadEP

Summary

```
#include <usbhw.h>

U32 USB_ReadEP (
  U32 EPNum,                        /* Endpoint number and direction
*/
  U8   *pData);                     /* Pointer to buffer to write to
*/
```

Description

The **USB_ReadEP** function reads data from the USB controller’s endpoint buffer to the local software buffer.

When new data are available, the **USB_EndPoint*Number*** function can call **USB_ReadEP** to obtain the data. The *EPNum* parameter contains the endpoint number (0-15) in the first 4 bits, and the direction in bit 7. Usually, the direction bit is 0, denoting an OUT endpoint. *pData* is the pointer to the buffer to store the data received from the endpoint.

It is not required to modify this function.

The **USB_ReadEP** function is part of the USB Device Controller Driver layer of the RL-USB Software Stack.

Return Value

The **USB_ReadEP** function returns the number of bytes read from the endpoint buffer.

See Also

USB_EndPoint*Number*, **USB_WriteEP**

Example

```
#include <usbhw.h>

void MSC_BulkOut (void)  {

  BulkLen = USB_ReadEP (MSC_EP_OUT, BulkBuf);
  switch (BulkStage)  {
    case MSC_BS_CBW:
      MSC_GetCBW ();
      break;
   …
  }
}
```

USB_TaskInit

Summary

```
#include <usbuser.h>

void USB_TaskInit  (
  void);
```

Description

The **USB_TaskInit** function creates all the USB event handling and endpoint tasks for endpoints that have been enabled in **usbcfg.h**. See Configuration Parameters on page 1-34 for how to enable the endpoints.

The **USB_TaskInit** function gives a fixed priority for the tasks. You can modify the priorities to suit your application’s requirements.

The **USB_TaskInit** function is part of the USB Core Driver layer of the RL-USB Software Stack. There is no requirement to modify this function.

Return Value

None.

See Also

USB_Device, USB_EndPoint*Number*

Example

```
#include <usbuser.h>

void USB_TaskInit (void)  {


…
#if (USB_EP_EVENT & (1 << 4))
  // Create task for endpoint 4 with a priority of 2
  USB_EPTask [4] = os_tsk_create (USB_EndPoint4, 2);
#endif
…
}
```

USB_WriteEP

Summary

```
#include <usbhw.h>

U32 USB_WriteEP(
  U32 EPNum,                    /* Endpoint number and direction
*/
  U8 *pData,                    /* Pointer to data buffer
*/
  U32 cnt);                     /* Number of bytes to write
*/
```

Description

The **USB_WriteEP** function writes data into the endpoint buffer. When the host computer requests new data, the **USB_EndPoint*Number*** function can call **USB_WriteEP** to send the data. The argument *EPNum* contains the endpoint number (0-15) in the first 4 bits, and the direction in bit 7. Usually, the direction bit is one (1) denoting an IN endpoint.

It is not required to modify this function.

The **USB_WriteEP** function is part of the USB Device Controller Driver layer of the RL-USB Software Stack.

Return Value

The **USB_WriteEP** function returns the number of bytes written to the endpoint buffer.

See Also

USB_EndPoint*Number*, **USB_ReadEP**

Example

```
#include <usbhw.h>

void USB_EndPoint1 (void) __task  {

  for (;;)  {
    os_evt_wait_or (USB_EVT_IN, 0xFFFF);    /* Wait for
USB_EVT_IN event */
    GetInReport ();
    USB_WriteEP (0x81, &InReport, sizeof (InReport) );
  }
}
```

usbd_adc_init

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_adc_init (
  void);
```

Description

The function **usbd_adc_init** initializes the USB Device for audio support. This function is called automatically and needs no invocation in the code.

The function **usbd_adc_init** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_init

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {
  ..
                                         // usbd_adc_init() is
invoked automatically by USB Device Core
  usbd_init ();                          // USB Device
Initialization
  usbd_connect (__TRUE);                 // USB Device Connect
  ..
}
```

usbd_cdc_init

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_init (
  void);
```

Description

The function **usbd_cdc_init** initializes the USB Device for CDC support. This function is called automatically and needs no invocation in the code.

The function **usbd_cdc_init** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {
  ..
                                         // usbd_cdc_init() is
invoked automatically by USB Device Core
  usbd_init ();                          // USB Device
Initialization
  usbd_connect (__TRUE);                 // USB Device Connect
  ..
}
```

usbd_cdc_ser_availchar

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_ser_availchar  (
  S32* availChar);              // available character pending at
port
```

Description

The function **usbd_cdc_ser_availchar** checks whether a character, *availChar*, is pending at the serial port.

The function **usbd_cdc_ser_availchar** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_init, **usbd_cdc_ser_closeport**, **usbd_cdc_ser_initport**, **usbd_cdc_ser_linestate**, **usbd_cdc_ser_openport**, **usbd_cdc_ser_read**, **usbd_cdc_ser_write**, **usbd_connect**, **usbd_init**, **usbd_vcom_chkserstate**, **usbd_vcom_serial2usb**, **usbd_vcom_usb2serial**

usbd_cdc_ser_closeport

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_ser_closeport (
  void);
```

Description

The function **usbd_cdc_ser_closeport** closes the serial communication port.

The function **usbd_cdc_ser_closeport** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

usbd_cdc_ser_initport

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_ser_initport  (
  U32 baudrate,                 // baud rate
  U32 databits,                 // data packet size
  U32 parity,                   // parity control bit
  U32 stopbits);                // stop bits
```

Description

The function **usbd_cdc_ser_initport** initializes the serial communication port with a baud rate of *baudrate*, data packet size *data*, parity control bit *parity*, and the number of stop bits *stopbits*.

The function **usbd_cdc_ser_initport** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

usbd_cdc_ser_linestate

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_ser_linestate  (
  U16* lineState);              // serial line status
```

Description

The function **usbd_cdc_ser_linestate** checks the line state of the serial port. The argument *lineState* is a pointer to the variable that stores the status. The variable is set with each read or write attempt to the COM port.

The function **usbd_cdc_ser_linestate** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_init, **usbd_cdc_ser_availchar**, **usbd_cdc_ser_closeport**, **usbd_cdc_ser_initport**, **usbd_cdc_ser_openport**, **usbd_cdc_ser_read**, **usbd_cdc_ser_write**, **usbd_connect**, **usbd_init**, **usbd_vcom_chkserstate**, **usbd_vcom_serial2usb**, **usbd_vcom_usb2serial**

usbd_cdc_ser_openport

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_cdc_ser_openport (
  void);
```

Description

The function **usbd_cdc_ser_openport** opens the serial communication port.

The function **usbd_cdc_ser_openport** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

usbd_cdc_ser_read

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

S32 usbd_cdc_ser_read  (
        S8*  buffer,               // character buffer
  const S32* length);              // buffer length, in bytes
```

Description

The function **usbd_cdc_ser_read** reads data, *buffer*, with the length *length* from the serial port. The argument *buffer* is a pointer to the data block that should be read. The argument *length* is a pointer that specifies the length of the data block.

The function **usbd_cdc_ser_read** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

Number of bytes read from the serial port.

See Also

usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_openport, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

usbd_cdc_ser_write

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

S32 usbd_cdc_ser_write  (
  const S8*  buffer,              // data buffer
        S32* length);             // buffer length, in bytes
```

Description

The function **usbd_cdc_ser_write** writes data with the length to the serial port. The argument *buffer* is a pointer to the data buffer. The argument *length* is a pointer specifiying the amount of data, in bytes, that should be written.

The function **usbd_cdc_ser_write** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

Number of bytes written to the serial port.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb, usbd_vcom_usb2serial

usbd_connect

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_connect (
  BOOL Conn);                          // Enable or disable the USB 
controller
```

Description

The function **usbd_connect** connects or disconnect the USB device controller to the bus. Invoke **usbd_connect** with *Conn* set to 1 (*__TRUE*) to ensure that the host can recognize the device.

Valid values for *Conn* are:

- 0 or (__FALSE) to disconnect the USB Device.
- 1 or (__TRUE) to connect the USB Device.

The function **usbd_connect** is part of the USB Device Core Driver layer of the RL-USB Device Software Stack.

Return Value

None.

See Also

usbd_adc_init, usbd_cdc_init, usbd_hid_init, usbd_init, usbd_msc_init, usbd_reset_core

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void)  {
  ..

  usbd_init ();                         // Initialize USB Device 
hardware
  usbd_connect (__TRUE);                // Enable the USB Device 
controller
  ..
}
```

usbd_hid_getinreport

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_hid_getinreport (
  U8* buf);                    // pointer to the buffer to store the
data
```

Description

The function **usbd_hid_getinreport** prepares data that will be returned to the USB Host when it asks for it. The argument *buf* is a pointer to the buffer that stores the report data.

The function **usbd_hid_getinreport** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_hid_init, usbd_hid_setoutreport, usbd_init

usbd_hid_init

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_hid_init (
  void);
```

Description

The function **usbd_hid_init** initializes the USB Device for HID support. This function is called automatically and needs no invocation in the code.

The function **usbd_hid_init** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_hid_getinreport, usbd_hid_setoutreport, usbd_init

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {
  ..
                                          // usbd_hid_init() is
invoked automatically by USB Device Core
  usbd_init ();                           // USB Device
Initialization
  usbd_connect (__TRUE);                  // USB Device Connect
  ..
}
```

usbd_hid_setoutreport

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_hid_setoutreport (
  U8* buf);                      // pointer to the buffer to store the
data
```

Description

The function **usbd_hid_setoutreport** processes the data received from the USB Host. The argument *buf* is a pointer to the buffer that contains the report data.

The function **usbd_hid_setoutreport** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_hid_getinreport, usbd_hid_init, usbd_init

usbd_init

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_init (
  void);
```

Description

The function **usbd_init** initializes the USB Device Controller Core and Hardware Driver (such as the USB clock). It starts all the tasks and sets up the main USB interrupt service routine. In any application, the **usbd_init** function must be called before invoking any other USB function. The function does not initialize any non-USB hardware features.

The **usbd_init** function is part of the USB Device Core Driver layer of the RL-USB Software Stack.

Return Value

None.

See Also

usbd_adc_init, **usbd_cdc_init**, **usbd_connect**, **usbd_hid_init**, **usbd_msc_init**, **usbd_reset_core**

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void)  {
  ..

  usbd_init();                             // Initialize USB
Device hardware
  usbd_connect(__TRUE);                    // Enable the USB
Device controller
  ..
}
```

usbd_msc_init

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_msc_init (
  void);
```

Description

The function **usbd_msc_init** initializes the USB Device for MSC support. This function is called automatically and needs no invocation in the code.

The function **usbd_msc_init** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modify this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_init, usbd_msc_read_sect, usbd_msc_write_sect

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {
  ..
                                         // usbd_msc_init() is
invoked automatically by USB Device Core
  usbd_init ();                          // USB Device
Initialization
  usbd_connect (__TRUE);                 // USB Device Connect
  ..
}
```

usbd_msc_read_sect

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_msc_read_sect (
  U32 block,                    // starting data block to be read
  U8* buf,                      // pointer to the buffer to store the
data
  U32 num_of_blocks);          // number of blocks to be read
);
```

Description

The function **usbd_msc_read_sect** reads the data that should be returned to the USB Host that requested it. The argument *block* specifies the starting block from where the data should be read. The argument *buf* is a pointer to the buffer where the read data should be stored. The argument *num_of_blocks* specifies the number of blocks that should be read.

The function **usbd_msc_read_sect** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modifying this function to the application needs.

Return Value

None.

See Also

usbd_connect, **usbd_init**, **usbd_msc_init**, **usbd_msc_write_sect**

usbd_msc_write_sect

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_msc_write_sect (
  U32 block,                   // starting block to where write data
  U8* buf,                     // pointer to the buffer that stores
the data
  U32 num_of_blocks);          // number of blocks to be written
);
```

Description

The function **usbd_msc_write_sect** writes data received from the USB Host. The argument *block* specifies the starting block to where data should be written. The argument *buf* is a pointer to the buffer that holds the data that should be written. The argument *num_of_blocks* specifies the number of blocks that should be written.

The function **usbd_msc_write_sect** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

Modifying this function to the application needs.

Return Value

None.

See Also

usbd_connect, usbd_init, usbd_msc_init, usbd_msc_read_sect

usbd_reset_core

Summary

```
#include <RTL.h>
#include <rl_usb.h>
#include <usb.h>

void usbd_reset_core (
  void);
```

Description

The function **usbd_reset_core** resets the USB Device Core. Invoke this function if connection errors are reported.

The function **usbd_reset_core** is part of the USB Device Core Driver layer of the RL-USB Device Software Stack.

Return Value

None;

See Also

usbd_connect, usbd_init

usbd_vcom_chkserstate

Summary

```
#include <RTL.h>
#include <rl_usb.h>

void usbd_vcom_chkserstate (
  void);
```

Description

The function **usbd_vcom_chkserstate** checks the state of the serial communication port.

The function **usbd_vcom_chkserstate** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

It is not required to modify this function.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_serial2usb, usbd_vcom_usb2serial

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {

  usbd_init();                          // USB Device 
Initialization
  usbd_connect(__TRUE);                 // USB Device Connect

  while (1) {                           // Loop forever
    ..
    usbd_vcom_chkserstate();            // Check serial 
connection
    ..
  }
}
```

usbd_vcom_serial2usb

Summary

```
#include <RTL.h>
#include <rl_usb.h>

void usbd_vcom_serial2usb (
  void);
```

Description

The function **usbd_vcom_serial2usb** sends data from the serial communication port to the USB CDC Device.

The function **usbd_vcom_serial2usb** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

It is not required to modify this function.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_usb2serial

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {

  usbd_init ();                          // USB Device
Initialization
  usbd_connect (__TRUE);                 // USB Device Connect

  while (1) {                            // Loop forever
    ..
    usbd_vcom_serial2usb ();             // Send data from serial
COM port to USB CDC Device
    ..
  }
}
```

usbd_vcom_usb2serial

Summary

```
#include <RTL.h>
#include <rl_usb.h>

void usbd_vcom_usb2serial (
  void);
```

Description

The function **usbd_vcom_usb2serial** sends data from the USB CDC Device to a serial communication port.

The function **usbd_vcom_serial2usb** is part of the USB Device Function Driver layer of the RL-USB Device Software Stack.

It is not required to modify this function.

Return Value

None.

See Also

usbd_cdc_init, usbd_cdc_ser_availchar, usbd_cdc_ser_closeport, usbd_cdc_ser_initport, usbd_cdc_ser_linestate, usbd_cdc_ser_openport, usbd_cdc_ser_read, usbd_cdc_ser_write, usbd_connect, usbd_init, usbd_vcom_chkserstate, usbd_vcom_serial2usb

Example

```
#include <RTL.h>
#include <rl_usb.h>

int main (void) {

  usbd_init ();                             // USB Device
Initialization
  usbd_connect (__TRUE);                    // USB Device Connect

  while (1) {                               // Loop forever
    ..
    usbd_vcom_usb2serial ();                // Send data from USB
CDC Device to serial COM port
    ..
  }
}
```

usbh_connected

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_connected (
  void);
```

Description

The **usbh_connected** function returns the connect status of USB host root port. The **usbh_connected** function is used for checking if there is a device present on the USB host bus.

The **usbh_connected** function is part of the of the **RL-USB-Host** software stack.

Return Value

The **usbh_connected** function returns one of the following manifest constants.

- **USBH_OK**
  Success.
- **USBH_NO_CONNECTED_ERROR**
  Indicates that no device is connected on USB host bus.

See Also

usbh_engine, usbh_init, usbh_uninit

Example

```
#include <rl_usb.h>

int main (void)  {
  ..
  if (usbh_connected() == USBH_OK)  {   // If device is connected
    ..
  }
  ..
}
```

usbh_engine

Summary

```
#include <rl_usb.h>

void usbh_engine (
  void);
```

Description

The **usbh_engine** function handles USB host events. It automatically executes enumeration when new device is connected and it uninitializes and releases resources when device is disconnected. The **usbh_engine** function should be called periodically to enable proper response to USB host events.

The **usbh_engine** function is part of the of the **RL-USB-Host** software stack.

Return Value

None.

See Also

usbh_connected, usbh_init, usbh_uninit

Example

```
#include <rl_usb.h>

int main (void)  {
  ..
  usbh_init();                          // Initialize USB host
stack and hardware
  ..
  while (1)  {
    ..
    usbh_engine();                      // Handle USB host events
    ..
  }
  ..
}
```

usbh_hid_kbd_getkey

Summary

```
#include <rl_usb.h>

int usbh_hid_kbd_getkey (
  void);
```

Description

The function **usbh_hid_kbd_getkey** retrieves the keyboard signal.

The **usbh_hid_kbd_getkey** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

Value of retrieved keyboard signal.

See Also

usbh_engine, usbh_hid_mouse_getdata, usbh_init

Example

```
#include <rl_usb.h>

int getline (char *line, int n)  {
  char c;
  ..
  c = (char )(usbh_hid_kbd_getkey ());      /* Read key from HID
keyboard     */
  ..
)
```

usbh_hid_mouse_getdata

Summary

```
#include <rl_usb.h>

BOOL usbh_hid_mouse_getdata (
  U8* btn,
  S8* x,
  S8* y,
  S8* wheel);
```

Description

The function **usbh_hid_mouse_getdata** retrieves the signals sent by an USB mouse. The argument *btn* is a pointer indicating the mouse button pressed. The arguments *x* and *y* are pointers indicating relative location change of the mouse pointer. The argument *wheel* is a pointer indicating the mouse wheel change.

The **usbh_hid_mouse_getdata** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

A boolean value indicating whether the mouse has been used.

See Also

usbh_engine, usbh_hid_kbd_getkey, usbh_init

Example

```
#include <rl_usb.h>

void getmouse (void)  {
  ..
  if (usbh_hid_mouse_getdata (U8 *btn, S8 *x, S8 *y, S8 *wheel))
{    /* mouse moved  */
    ..
  }
)
```

usbh_init

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_init (
  void);
```

Description

The **usbh_init** function initializes the USB Host Stack and USB Host Controller Hardware. It prepares the USB Host Controller to detect connection or disconnection of a device on the bus and does enumeration of connected device. In any application, the **usbh_init** function must be called before calling any other USB Host function. The function does not initialize any non-USB Host hardware features.

The **usbh_init** function is part of the of the **RL-USB-Host** Software Stack.

Return Value

The **usbh_init** function returns one of the following manifest constants.

- **USBH_OK**
  Success.
- **USBH_MEM_POOL_INIT_ERROR**
  Indicates that memory pool used for USB Host communication was not initialized properly.
- **USBH_HW_ERROR**
  Indicates that the initialization of hardware failed.
- **USBH_EP_NOT_AVAILABLE_ERROR**
  Indicates that the there was on available resource for creating endpoint that will be used as Control Endpoint 0.

See Also

usbh_connected, usbh_engine, usbh_uninit

Example

```
#include <rl_usb.h>

int main (void)  {
  ..
  usbh_init();                          // Initialize USB host
stack and hardware
  ..
}
```

usbh_msc_read

Summary

```
#include <rl_usb.h>

BOOL usbh_msc_read (
  U32  blk_adr,
  U8*  ptr_data,
  U16  blk_num);
```

Description

The function **usbh_msc_read** reads data from a mass storage device. The argument *blk_adr* is the address of starting block to be read. The arguments *ptr_data* is a pointer indicating the location where data will be read. The argument *blk_num* is a value indicating the number of blocks to be read.

The **usbh_msc_read** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

A boolean value indicating whether the read process was successful.

See Also

usbh_engine, usbh_init, usbh_msc_read_config, usbh_msc_status, usbh_msc_write

usbh_msc_read_config

Summary

```
#include <rl_usb.h>

BOOL usbh_msc_read_config (
  U32*   tot_blk_num,
  U32*   blk_sz);
```

Description

The function **usbh_msc_read_config** reads the configuration parameters of a mass storage device. The argument *tot_blk_num* is a pointer indicating the total number of blocks on the device. The argument *blk_sz* is a pointer indicating the block size.

The **usbh_msc_read_config** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

A boolean value indicating whether the USB Device configuration could be read.

See Also

usbh_engine, usbh_init, usbh_msc_read, usbh_msc_status, usbh_msc_write

usbh_msc_status

Summary

```
#include <rl_usb.h>

BOOL usbh_msc_status (
  void);
```

Description

The function **usbh_msc_status** checks whether a mass storage device is connected.

The **usbh_msc_status** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

Returns a boolean value indicating that a mass storage device has been connected.

See Also

usbh_engine, usbh_init, usbh_msc_read, usbh_msc_read_config, usbh_msc_write

Example

```
#include <rl_usb.h>

int main (void) {
  ..
  init_msd ();                               // Initialize mass
storage device
  while (1) {
    usbh_engine();
    if (!usbh_msc_status()) {                // If device is not
connected
      usbh_engine();
    }
    ..
  }
}
```

usbh_msc_write

Summary

```
#include <rl_usb.h>

BOOL usbh_msc_write (
  U32  blk_adr,
  U8*  ptr_data,
  U16  blk_num);
```

Description

The function **usbh_msc_write** writes data to a mass storage device. The argument *blk_adr* is the address of staring block to be written. The argument *ptr_data* is a pointer indicating the location of data to be written. The argument *blk_num* is a value indicating the number of blocks to be written.

The **usbh_msc_write** function is part of the of the **RL-USB Host Class Driver** software layer.

Return Value

A boolean value indicating whether the write process was successful.

See Also

usbh_engine, **usbh_init**, **usbh_msc_read**, **usbh_msc_read_config**, **usbh_msc_status**

usbh_ohci_hw_delay

Summary

```
#include <rl_usb.h>

void usbh_ohci_hw_delay (
    U32 delay);          /* Delay. */
```

Description

The **usbh_ohci_hw_delay** function provides a functionality to the USB host for OHCI controller to delay execution for a specified time. This function is used by USB Host OHCI controller driver for hardware specific request that require specific timing, for example for hardware initialization and port resetting.

The **usbh_ohci_hw_delay** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different CPU clock settings. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

note

- Delay base for **usbh_ohci_hw_delay** function is 100 us.

Return Value

None.

See Also

usbh_ohci_hw_init, **usbh_ohci_hw_irq_dis**, **usbh_ohci_hw_irq_en**, **usbh_ohci_hw_power**, **usbh_ohci_hw_reg_rd**, **usbh_ohci_hw_reg_wr**, **usbh_ohci_hw_uninit**

Example

```
void usbh_ohci_hw_delay (U32 delay) {
  delay <<= 10;
  while (delay--) {
    __nop(); __nop(); __nop(); __nop(); __nop(); __nop();
__nop(); __nop();
  }
}
```

usbh_ohci_hw_init

Summary

```
#include <rl_usb.h>


USBH_ERROR usbh_ohci_hw_init (
  void);
```

Description

The **usbh_ohci_hw_init** function is used for initialization of hardware (clocks, pins) for the USB host OHCI controller.

The **usbh_ohci_hw_init** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller hardware. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_init** function returns one of the following manifest constants.

- **USBH_OK**
  Success.

- **USBH_HW_ERROR**
  Indicates that the initialization of hardware failed.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_irq_dis, usbh_ohci_hw_irq_en, usbh_ohci_hw_power, usbh_ohci_hw_reg_rd, usbh_ohci_hw_reg_wr, usbh_ohci_hw_uninit

Example

```
USBH_ERROR usbh_ohci_hw_init (void) {
  U32 tout;

  LPC_SC->PCONP       |=  (1UL << 31);
  LPC_USB->OTGClkCtrl |=  0x19;

  for (tout = 100; ; tout--) {
    if ((LPC_USB->OTGClkSt & 0x19) == 0x19)
      break;
    if (!tout)
      return (USBH_HW_ERROR);
  }

  LPC_USB->OTGStCtrl  |=  0x03;

  LPC_PINCON->PINSEL1 &= ~((3 << 26) | (3 << 28));
  LPC_PINCON->PINSEL1 |=  ((1 << 26) |
                           (1 << 28));

  LPC_PINCON->PINSEL3 &= ~((3 <<  4) | (3 <<  6) | (3 << 12) | (3
<< 22));
  LPC_PINCON->PINSEL3 |=  ((1 <<  4) |
                           (2 <<  6) |
                           (2 << 12) |
                           (2 << 22));


  NVIC_SetPriority (USB_IRQn, 0);
  return (USBH_OK);
}
```

usbh_ohci_hw_irq_dis

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_ohci_hw_irq_dis (
  void);
```

Description

The **usbh_ohci_hw_irq_dis** function is used for disabling the interrupt of the USB host OHCI controller.

The **usbh_ohci_hw_irq_dis** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller interrupt system. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_irq_dis** function returns one of the following manifest constants.

- **USBH_OK**
  Success.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_init, usbh_ohci_hw_irq_en, usbh_ohci_hw_power, usbh_ohci_hw_reg_rd, usbh_ohci_hw_reg_wr, usbh_ohci_hw_uninit

Example

```
USBH_ERROR usbh_ohci_hw_irq_dis (void) {
  NVIC_DisableIRQ (USB_IRQn);

  return (USBH_OK);
}
```

usbh_ohci_hw_irq_en

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_ohci_hw_irq_en (
  void);
```

Description

The **usbh_ohci_hw_irq_en** function is used for enabling the interrupt of the USB host OHCI controller.

The **usbh_ohci_hw_irq_en** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller interrupt system. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_irq_en** function returns one of the following manifest constants.

- **USBH_OK**
  Success.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_init, usbh_ohci_hw_irq_dis, usbh_ohci_hw_power, usbh_ohci_hw_reg_rd, usbh_ohci_hw_reg_wr, usbh_ohci_hw_uninit

Example

```
USBH_ERROR usbh_ohci_hw_irq_en (void) {
  NVIC_EnableIRQ  (USB_IRQn);

  return (USBH_OK);
}
```

usbh_ohci_hw_power

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_ohci_hw_power (
    U32 on);                /* Requested power state. */
```

Description

The **usbh_ohci_hw_power** function is used for supplying or removing power on the USB host OHCI controller port.

The **usbh_ohci_hw_power** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller hardware. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_power** function returns one of the following manifest constants.

- **USBH_OK**
  Success.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_init, usbh_ohci_hw_irq_dis, usbh_ohci_hw_irq_en, usbh_ohci_hw_reg_rd, usbh_ohci_hw_reg_wr, usbh_ohci_hw_uninit

Example

```
USBH_ERROR usbh_ohci_hw_power (U32 on) {
  if (on) {
    usbh_ohci_hw_reg_wr(oHcRhStatus,
usbh_ohci_hw_reg_rd(oHcRhStatus) | USBH_OHCI_HcRhStatus_LPSC);
  } else {
    usbh_ohci_hw_reg_wr(oHcRhStatus,
usbh_ohci_hw_reg_rd(oHcRhStatus) | USBH_OHCI_HcRhStatus_LPS );
  }

  return (USBH_OK);
}
```

usbh_ohci_hw_reg_rd

Summary

```
#include <rl_usb.h>

U32 usbh_ohci_hw_reg_rd (
    U32 reg_ofs);        /* Register offset. */
```

Description

The **usbh_ohci_hw_reg_rd** function is used for reading a value from the USB host OHCI register.

The **usbh_ohci_hw_reg_rd** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller register interface. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_reg_rd** function returns 32-bit value read from OHCI register.

See Also

usbh_ohci_hw_delay, **usbh_ohci_hw_init**, **usbh_ohci_hw_irq_dis**, **usbh_ohci_hw_irq_en**, **usbh_ohci_hw_power**, **usbh_ohci_hw_reg_wr**, **usbh_ohci_hw_uninit**

Example

```
U32 usbh_ohci_hw_reg_rd (U32 reg_ofs) {
  return (*((U32 *)(USBH_OHCI_ADDR + reg_ofs)));
}
```

usbh_ohci_hw_reg_wr

Summary

```
#include <rl_usb.h>

void usbh_ohci_hw_reg_wr (
    U32 reg_ofs,         /* Register offset. */
    U32 val);             /* Value to write to register. */
```

Description

The **usbh_ohci_hw_reg_wr** function is used for writing a value to the USB host OHCI register.

The **usbh_ohci_hw_reg_wr** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller register interface. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

None.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_init, usbh_ohci_hw_irq_dis, usbh_ohci_hw_irq_en, usbh_ohci_hw_power, usbh_ohci_hw_reg_rd, usbh_ohci_hw_uninit

Example

```
void usbh_ohci_hw_reg_wr (U32 reg_ofs, U32 val) {
  *((U32 *)(USBH_OHCI_ADDR + reg_ofs)) = val;
}
```

usbh_ohci_hw_uninit

Summary

```
#include <rl_usb.h>

USBH_ERROR usbh_ohci_hw_uninit (
  void);
```

Description

The **usbh_ohci_hw_uninit** function is used for uninitialization of hardware (clocks, pins) for the USB host OHCI controller.

The **usbh_ohci_hw_uninit** function is part of the **RL-USB-Host** software stack.

You can modify this function to suit different OHCI controller hardware. This function will need to be changed if OHCI controller is used on a chip that does not yet have a low level driver provided.

Return Value

The **usbh_ohci_hw_init** function returns one of the following manifest constants.

- **USBH_OK**
  Success.
- **USBH_HW_ERROR**
  Indicates that the initialization of hardware failed.

See Also

usbh_ohci_hw_delay, usbh_ohci_hw_init, usbh_ohci_hw_irq_dis, usbh_ohci_hw_irq_en, usbh_ohci_hw_power, usbh_ohci_hw_reg_rd, usbh_ohci_hw_reg_wr

Example

```
USBH_ERROR usbh_ohci_hw_uninit (void) {
  U32 tout;

  LPC_USB->OTGStCtrl   &= ~0x03;

  LPC_PINCON->PINSEL3 &= ~((3 <<  4) | (3 <<  6) | (3 << 12) | (3
<< 22));
  LPC_PINCON->PINSEL1 &= ~((3 << 26) | (3 << 28));

  LPC_USB->OTGClkCtrl &= ~0x19;

  for (tout = 100; ; tout--) {
    if ((LPC_USB->OTGClkSt & 0x19) == 0)
      break;
    if (!tout)
      return (USBH_HW_ERROR);
  }

  LPC_SC->PCONP       &= ~(1UL << 31);

  return (USBH_OK);
}
```

usbh_uninit

Summary

```
#include <usb.h>

USBH_ERROR usbh_uninit (
  void);
```

Description

The **usbh_uninit** function uninitializes the USB Host Stack and USB Host Controller Hardware. It can be used if during the application run time USB Host Stack needs to be disabled for whatever reason (for example lowering power consumption). After **usbh_uninit** function is called only **usbh_init** should be called for reinitialization of USB Host Stack and USB Host Controller Hardware.

The **usbh_uninit** function is part of the of the **RL-USB-Host** software stack.

Return Value

The **usbh_uninit** function returns one of the following manifest constants.

- **USBH_OK**
  Success.
- **USBH_EP_INVALID_ERROR**
  Indicates that Control Endpoint 0 was invalid.
- **USBH_HW_ERROR**
  Indicates that the uninitialization of hardware failed.

See Also

usbh_connected, usbh_engine, usbh_init

Example

```
#include <usb.h>

int main (void)  {
  ..
  usbh_init();                        // Initialize USB host
stack and hardware
  ..
  usbh_uninit();                      // Uninitialize USB host
stack and hardware
  ..
}
```

Library Files

The Real-Time Library includes the following library files, which are located in the **\Keil\ARM\RV31\LIB** folder.

Library File	Description
RTX_ARM_L.LIB	Real-Time eXecutive kernel library (RL-RTX) for ARM7™ and ARM9™ devices - Little Endian.
RTX_ARM_B.LIB	Real-Time eXecutive kernel library (RL-RTX) for ARM7™ and ARM9™ devices - Big Endian.
RTX_CM1.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M0 and Cortex™-M1 devices - Little Endian.
RTX_CM1_B.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M0 and Cortex™-M1 devices - Big Endian.
RTX_CM3.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M3 devices - Little Endian.
RTX_CM3_B.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M3 devices - Big Endian.
RTX_CM4.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M4 devices - Little Endian.
RTX_CM4_B.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-M4 devices - Big Endian.
RTX_CR4.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-R4 devices - Little Endian.
RTX_CR4_B.LIB	Real-Time eXecutive kernel library (RL-RTX) for Cortex™-R4 devices - Big Endian.
FS_ARM_L.LIB	Flash File System library (RL-FlashFS) for ARM7™ and ARM9™ devices - Little Endian.
FS_CM3.LIB	Flash File System library (RL-FlashFS) for Cortex™-M3 devices - Little Endian.
TCP_ARM_L.LIB	TCP/IP library without debug messages (RL-TCPnet) for ARM7™ and ARM9™ devices - Little Endian.
TCP_CM1.LIB	TCP/IP library without debug messages (RL-TCPnet) for Cortex™-M1 devices - Little Endian.
TCP_CM3.LIB	TCP/IP library without debug messages (RL-TCPnet) for Cortex™-M3 devices - Little Endian.
TCPD_ARM_L.LIB	TCP/IP library containing debug messages (RL-TCPnet) for ARM7™ and ARM9™ devices - Little Endian.
TCPD_CM1.LIB	TCP/IP library containing debug messages (RL-TCPnet) for Cortex™-M0 and Cortex™-M1 devices - Little Endian.
TCPD_CM3.LIB	TCP/IP library containing debug messages (RL-TCPnet) for Cortex™-M3 devices - Little Endian.
USB_ARM_L.LIB	USB library containing host and device mode (RL-USB) for ARM7™ and ARM9™ devices - Little Endian.
USB_CM3.LIB	USB library containing host and device mode (RL-USB) for Cortex™-M3 devices - Little Endian.

Note

- Depending on the target architecture, one of either file RTX_ARM_L.LIB, RTX_ARM_B.LIB, RTX_CM1.LIB, RTX_CM1_B.LIB, RTX_CM3.LIB, RTX_CM3_B.LIB, RTX_CM4.LIB, RTX_CM4_B.LIB, RTX_CR4.LIB, or RTX_CR4_B.LIB is included automatically into the link process when you select the RTX kernel (in the µVision IDE) as the operating system for the project. To use any of the other RL-ARM™ library files, the user must explicitly add them to the link process of the application.
- There is no object library for RL-CAN driver. Include the RL-CAN source files manually in the project.

Appendix

This appendix provides additional details about the RL-ARM™ Real-Time Library:

- µVision Debug Dialogs
- Glossary

A. µVision Debug Dialogs

A **kernel-aware** dialog displays all aspects of the **RTX Kernel** and the tasks in your program. It can also display **files** when the **Flash File System** is used. It may be used also with your target hardware, if you are debugging your application over **ULINK**® USB-JTAG adapter.

When you select the **RTX Kernel** from the **Peripherals** menu, µVision® loads Kernel-Aware debug DLL - **ARTXARM.DLL**. This file is located in **\KEIL\ARM\BIN\** folder.

The following pages are available:

- **Active Tasks** - shows currently active tasks and their status.
- **System** - shows the system information of the application.
- **File System** - shows the files stored in file system and some file information.

Note

- The **ARTXARM.DLL** must know the **exact format** of the kernel and Flash File System control variables. If the format of a variable is changed in a new **RL-ARM** release, then this **DLL** is also updated. The ARTXARM debug pages may display **erroneous** results if you are using an incompatible **ARTXARM.DLL** driver.

B. Glossary

Active task

A task which is created and not yet destroyed.

Context switch

A task takes over the CPU from another task, after an event for this task occurred, and provided, this task has a higher priority than the currently running task.

Cooperative multitasking

Tasks of same priority keep in possession of the CPU resources until they relinquish it to other tasks.

Event

A signal or message which triggers some kind of processing.

Interrupt service routine

A function into which the program vectors into on occurrence of an hardware or software interrupt.

ISR

Abbreviation for **interrupt service routine**

Multi-tasking

A set of tasks which share one or more CPUs for concurrent processing.

Memory pool

A list of memory blocks having the same size which can get allocated and de allocated dynamically by application functions.

Preemption

The process which moves a currently running task into the ready state, when a task with higher priority gets ready to run.

Priority

The importance of a task. Important or time-critical task can get control over the CPU resources even if another task of lower priority is currently executing.

Polling

A task which runs in an infinite loop without waiting for events.

Round robin

A technique to force running tasks to the ready state and thus allowing ready tasks of the same priority to get control over the CPU resources.

Task

A function which does some processing in concurrence with other functions.

Socket

A socket is one endpoint of a two-way communication link between two programs running on the network. It is bound to a port number so that the TCP/UDP layer can identify the application that data is destined to be sent.

Port number

A port number identifies both a computer and also a "channel" within that computer where network communication will take place.