Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of "Quality Parts,Customers Priority,Honest Operation,and Considerate Service",our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

KEIL™
Tools by ARM

Getting Started

Building Applications with RL-ARM

For ARM Processor-Based Microcontrollers

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than for the purchaser's personal use, without written permission.

Copyright © 1997-2009 ARM Ltd and ARM Germany GmbH.
All rights reserved.

Keil, the Keil Software Logo, µVision, MDK-ARM, RL-ARM, ULINK, and Device Database are trademarks or registered trademarks of ARM Ltd, and ARM Inc.

Microsoft® and Windows™ are trademarks or registered trademarks of Microsoft Corporation.

NOTE
This manual assumes that you are familiar with Microsoft® Windows™ and the hardware and instruction set of the ARM7™ and ARM9™ processor families or the Cortex™-M series processors. In addition, basic knowledge of µVision®4 is anticipated.

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.

Preface

This manual is an introduction to the **Real-Time Library** (RL-ARM™), which is a group of tightly coupled libraries designed to solve the real-time and communication challenges of embedded systems based on ARM processor-based microcontroller devices.

Using This Book

This book comes with a number of practical exercises that demonstrate the key operating principles of the RL-ARM. To use the exercises you will need to have both the Keil™ Microcontroller Development Kit (MDK-ARM™) installed and the Real-Time Library (RL-ARM). If you are new to the MDK-ARM, there is a separate *Getting Started* guide, which will introduce you to the key features. The online documentation for the MDK-ARM, including the *Getting Started* guide, is located at www.keil.com/support/man_arm.htm.

Alongside the standard RL-ARM examples, this book includes a number of additional examples. These examples present the key principles outlined in this book using the minimal amount of code. Each example is designed to be built with the evaluation version of the MDK-ARM. If this is not possible, the example is prebuilt so that it can be downloaded and run on a suitable evaluation board.

This book is useful for students, beginners, advanced and experienced developers alike.

However, it is assumed that you have a basic knowledge of how to use microcontrollers and that you are familiar with the instruction set of your preferred microcontroller. In addition, it is helpful to have basic knowledge on how to use the µVision Debugger & IDE.

Chapter Overview

“Chapter 1. **Introduction**”, provides a product overview, remarks referring to the installation requirements, and shows how to get support from the Keil technical support team.

“Chapter 2. **Developing with an RTOS**”, describes the advantages of the RTX, explains the RTX kernel, and addresses RTOS features, such as tasks, semaphores, mutexes, time management, and priority schemes.

“Chapter 3. **RL-Flash Introduction**”, describes the features of the embedded file system, how to set it up, configuration options, standard routines used to maintain the file system, and how to adapt flash algorithms.

“Chapter 4. **RL-TCPnet Introduction**”, describes the network model, TCP key features, communication protocols, and how to configure an ARM processor-based microcontroller to function with HTTP, Telnet, FTP, SMTP, or DNS applications.

“Chapter 5. **RL-USB Introduction**”, describes the USB key features, the physical and logical network, pipes and endpoints, the device communication descriptors, and the supported interfaces and their classes.

“Chapter 6. **RL-CAN Introduction**”, describes the CAN key concepts, the message frame, and the programming API implemented.

Document Conventions

Examples	Description
README.TXT[1]	Bold capital text is used to highlight the names of executable programs, data files, source files, environment variables, and commands that you can enter at the command prompt. This text usually represents commands that you must type in literally. For example: **ARMCC.EXE** **DIR** **LX51.EXE**
`Courier`	Text in this typeface is used to represent information that is displayed on the screen or is printed out on the printer This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents required information that you must provide. For example, *projectfile* in a syntax string means that you must supply the actual project file name Occasionally, italics are used to emphasize words in the text.
Elements that repeat…	Ellipses (…) are used to indicate an item that may be repeated

Omitted code
.
.
.
	Vertical ellipses are used in source code listings to indicate that a fragment of the program has been omitted. For example:
	void main (void) {
	.
	.
	.
	while (1);

«Optional Items»	Double brackets indicate optional items in command lines and input fields. For example: **C51 TEST.C PRINT** «filename»
{ opt1 | opt2 }	Text contained within braces, separated by a vertical bar represents a selection of items. The braces enclose all of the choices and the vertical bars separate the choices. Exactly one item in the list must be selected.
Keys	Text in this sans serif typeface represents actual keys on the keyboard. For example, “Press **F1** for help”.
Underlined text	Text that is underlined highlights web pages. In some cases, it marks email addresses.

[1] *It is not required to enter commands using all capital letters.*

Content

Preface 3

Document Conventions 5

Content 6

Chapter 1. Introduction 10
RL-ARM Overview 10
RTX RTOS 11
Flash File System 11
TCP/IP 12
USB 12
CAN 13
Installation 14
Product Folder Structure 14
Last-Minute Changes 15
Requesting Assistance 15

Chapter 2. Developing With an RTOS 16
Getting Started 16
Setting-Up a Project 17
RTX Kernel 19
Tasks 19
Starting RTX 21
Creating Tasks 22
Task Management 24
Multiple Instances 24
Time Management 24
Time Delay 25
Periodic Task Execution 26
Virtual Timer 26
Idle Demon 27
Inter-Task Communication 28
Events 28
RTOS Interrupt Handling 29
Task Priority Scheme 31
Semaphores 32
Using Semaphores 34
Signaling 34
Multiplex 34

Rendezvous ... 35
Barrier Turnstile ... 36
Semaphore Caveats ... 38
Mutex ... 38
Mutex Caveats ... 39
Mailbox ... 39
Task Lock and Unlock ... 43
Configuration ... 43
Task Definitions ... 44
System Timer Configuration ... 45
Round Robin Task Switching ... 45
Scheduling Options ... 45
Pre-emptive Scheduling ... 46
Round Robin Scheduling ... 46
Round Robin Pre-emptive Scheduling ... 47
Co-operative Multitasking ... 47
Priority Inversion ... 47

Chapter 3. RL-Flash Introduction ... 49
Getting Started ... 49
Setting-Up the File System ... 50
File I/O Routines ... 52
Volume Maintenance Routines ... 54
Flash Drive Configuration ... 56
Adapting Flash Algorithms for RL-Flash ... 58
MultiMedia Cards ... 60
Serial Flash ... 62

Chapter 4. RL-TCPnet Introduction ... 63
TCP/IP – Key Concepts ... 63
Network Model ... 63
Ethernet and IEEE 802.3 ... 65
TCP/IP Datagrams ... 65
Internet Protocol ... 65
Address Resolution Protocol ... 66
Subnet Mask ... 67
Dynamic Host Control Protocol DHCP ... 68
Internet Control Message Protocol ... 68
Transmission Control Protocol ... 69
User Datagram Protocol ... 70
Sockets ... 70
First Project - ICMP PING ... 71

Debug Support ... 74
Using RL-TCPnet with RTX ... 74
RL-TCPnet Applications ... 76
Trivial File Transfer ... 76
Adding the TFTP Service ... 76
HTTP Server ... 77
Web Server Content ... 78
Adding Web Pages ... 78
Adding HTML as C Code ... 79
Adding HTML with RL-Flash ... 81
The Common Gateway Interface ... 82
Dynamic HTML ... 82
Data Input Using Web Forms ... 84
Using the POST Method ... 84
Using the GET Method ... 87
Using JavaScript ... 88
AJAX Support ... 90
Simple Mail Transfer Client ... 94
Adding SMTP Support ... 94
Sending a Fixed Email Message ... 95
Dynamic Message ... 96
Telnet Server ... 98
Telnet Helper Functions ... 100
DNS Client ... 101
Socket Library ... 102
User Datagram Protocol (UDP) Communication ... 103
Transmission Control Protocol (TCP) Communication ... 105
Deployment ... 108
Serial Drivers ... 109

Chapter 5. RL-USB Introduction ... 111
The USB Protocol – Key Concepts ... 111
USB Physical Network ... 111
Logical Network ... 112
USB Pipes And Endpoints ... 113
Interrupt Pipe ... 115
Isochronous Pipe ... 115
Bulk Pipe ... 115
Bandwidth Allocation ... 116
Device Configuration ... 117
Device Descriptor ... 118
Configuration Descriptor ... 119

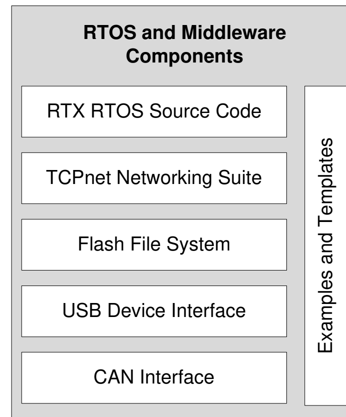
Interface Descriptor 120
Endpoint Descriptor 121
RL-USB 122
RL-USB Driver Overview 122
First USB Project 124
Configuration 124
Event Handlers 125
USB Descriptors 126
Class Support 127
Human Interface Device 128
HID Report Descriptors 128
HID Client 133
Enlarging the IN & OUT Endpoint Packet Sizes 134
Mass Storage 136
Audio Class 138
Composite Device 139
Compliance Testing 140

Chapter 6. RL-CAN Introduction 141
The CAN Protocol – Key Concepts 141
CAN Node Design 142
CAN Message Frames 143
CAN Bus Arbitration 145
RL-CAN Driver 146
First Project 146
CAN Driver API 147
Basic Transmit and Receive 148
Remote Request 149
Object Buffers 151

Glossary 152

Chapter 1. Introduction

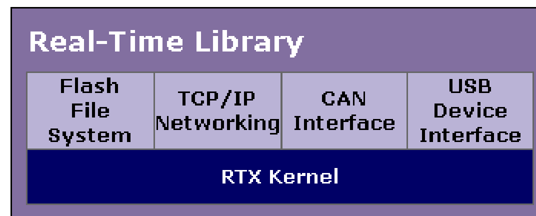
The last few years have seen an explosive growth in both the number and complexity of ARM processor-based microcontrollers. This diverse base of devices now offers the developer a suitable microcontroller for almost all applications. However, this rise in sophisticated hardware also calls for more and more complex software. With ever-shorter project deadlines, it is becoming just about impossible to develop the software to drive these devices without the use of third-party middleware.



The Keil Real-Time Library (RL-ARM) is a collection of easy-to-use middleware components that are designed to work across many different microcontrollers. This allows you to learn the software once and then use it multiple times. The RL-ARM middleware integrates into the Keil Microcontroller Development Kit (MDK-ARM).

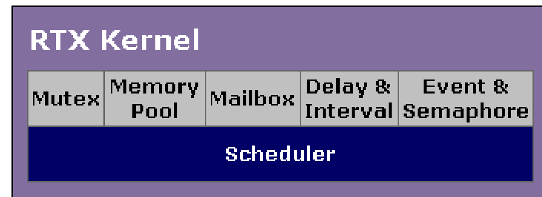
These two development tools allow you to rapidly develop sophisticated software applications across a vast range of ARM processor-based microcontrollers. In this book, we will look at each of the RL-ARM middleware components and see how to use all the key features in typical applications.

RL-ARM Overview



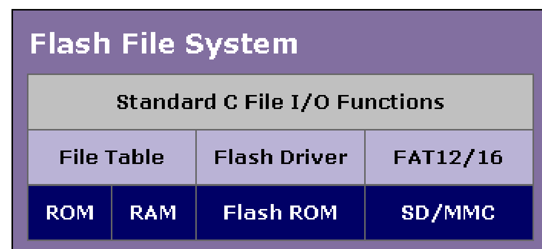
The RL-ARM library consists of five main components; a Flash-based file system, a TCP/IP networking suite, drivers for USB and CAN, and the RTX Kernel. Each of the middleware components is designed to be used with the Keil RTX real-time operating system. However, with the exception of the CAN driver, each component may be used without RTX.

RTX RTOS



Traditionally developers of small, embedded applications have had to write virtually all the code that runs on the microcontroller. Typically, this is in the form of interrupt handlers with a main background-scheduling loop. While there is nothing intrinsically wrong with this, it does rather miss the last few decades of advancement in program structure and design. Now, for the first time, with the introduction of 32-bit ARM processor-based microcontrollers we have low-cost, high-performance devices with increasingly large amounts of internal SRAM and Flash memory. This makes it possible to use more advanced software development techniques. Introducing a Real-Time Operating System (RTOS) or real-time executive into your project development is an important step in the right direction. With an RTOS, all the functional blocks of your design are developed as tasks, which are then scheduled by RTX. This forces a detailed design analysis and consideration of the final program structure at the beginning of the development. Each of the program tasks can be developed, debugged, and tested in isolation before integration into the full system. Each RTOS task is then easier to maintain, document, and reuse. However, using an RTOS is only half the story. Increasingly, customers want products that are more complex in shorter and shorter time. While microcontrollers with suitable peripherals are available, the challenge is to develop applications without spending months writing the low-level driver code.

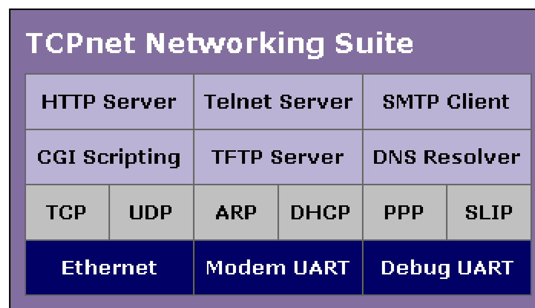
Flash File System



The RL-Flash file system allows you to place a PC-compatible file system in any region of a microcontroller's memory. This includes the on-chip and external RAM and Flash memory, as well as SPI based Flash memory and SD/MMC memory cards.

The RL-Flash file system comes with all the driver support necessary, including low-level Flash drivers, SPI drivers, and MultiMedia Card interface drivers. This gets the file system up-and-running with minimal fuss and allows you to concentrate on developing your application software. In the past, the use of a full file system in a small, embedded microcontroller has been something of a luxury. However, once you start developing embedded firmware with access to a small file system, you will begin to wonder how you ever managed without it!

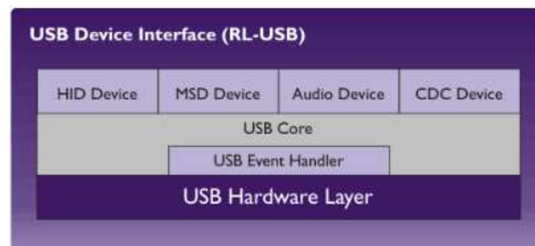
TCP/IP



The RL-TCPnet library is a full networking suite written for small ARM processor-based microcontrollers specifically. It consists of one generic library with dedicated Ethernet drivers for supported microcontrollers and a single configuration file. SLIP and PPP protocols are also supported to allow UART-based communication either directly from a PC or remotely via a modem.

The RL-TCPnet library supports raw TCP and UDP communication, which allows you to design custom networking protocols. Additional application layer support can be added to enable common services, including SMTP clients to send email notification, plus DNS and DHCP clients for automatic configuration. RL-TCPnet can also enable a microcontroller to be a server for the TELNET, HTTP, and File Transfer (FTP) protocols.

USB

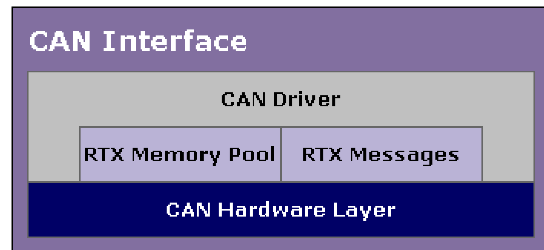


The USB protocol is complex and wide-ranging. To implement a USB-based peripheral, you need a good understanding of the USB peripheral, the USB protocol, and the USB host operating system.

Typically, the host will be a PC. This means that you need to have a deep knowledge of the Windows operating system and its device drivers. Getting all of these elements working together would be a development project in its own. Like the TCP/IP library, the RL-USB driver is a common software stack designed to work across all supported microcontrollers. Although you can use the RL-USB driver to communicate with a custom Windows device driver, it has been designed to support common USB classes. Each USB class has its own native driver within the Windows operating system. This means that you do not need to develop or maintain your own driver.

The class support provided with RL-USB includes Human Interface Device (HID), Mass Storage Class (MSC), Communication Device Class (CDC), and Audio Class. The HID Class allows you to exchange custom control and configuration data with your device. The Mass Storage Class allows the Windows operating system to access the data stored within the RL-Flash file system in the same manner as a USB pen drive. The Communication Device Class can be used to realize a virtual COM Port. Finally, the Audio Class allows you to exchange streaming audio data between the device and a PC. Together these four classes provide versatile support for most USB design requirements.

CAN



The RL-CAN driver is the one component of the RL-ARM library that is tightly coupled to the RTX. The CAN driver consists of just six functions that allow you to initialize a given CAN peripheral, define, transmit and receive CAN message objects, and exchange data with other nodes on the CAN network.

The RL-CAN driver has a consistent programming API for all supported CAN peripherals, allowing easy migration of code or integration of several different microcontrollers into the one project. The CAN driver also uses RTX message queues to buffer, transmit and receive messages, ensuring ordered handling of the CAN network data.

Installation

The RL-ARM is a collection of middleware components designed to integrate with the Keil Microcontroller Development Kit (MDK-ARM). To use this book you will need to have both the MDK-ARM and RL-ARM installed on your PC. MDK-ARM may be installed from either CD-ROM, or may be downloaded from the web. Currently, RL-ARM may only be downloaded from the web.

Keil products are available on CD-ROM and via download from www.keil.com. Updates to the related products are regularly available at www.keil.com/update. Demo versions of various products are obtainable at www.keil.com/demo. Additional information is provided under www.keil.com/arm.

Please check the minimum hardware and software requirements that must be satisfied to ensure that your Keil development tools are installed and will function properly. Before attempting installation, verify that you have:

- A standard PC running Microsoft Windows XP, or Windows Vista,
- 1GB RAM and 500 MB of available hard-disk space is recommended,
- 1024x768 or higher screen resolution; a mouse or other pointing device,
- A CD-ROM drive.

Product Folder Structure

The **SETUP** program copies the development tools into subfolders. The base folder defaults to **C:\KEIL**. When the RL-ARM is installed, it integrates into the MDK-ARM installation. The table below outlines the key RL-ARM files:

File Type	Path
MDK-ARM Toolset	**C:\KEIL\ARM**
Include and Header Files	**C:\KEIL\ARM\RV*xx*\INC**
Libraries	**C:\KEIL\ARM\RV*xx*\LIB**
Source Code	**C:\KEIL\ARM\RL**
Standard Examples	**C:\KEIL\ARM\Boards*manufacturer**board***
Flash Programming	**C:\KEIL\ARM\FLASH**
On-line Help Files and Release Notes	**C:\KEIL\ARM\HLP**

Last-Minute Changes

As with any high-tech product, last minute changes might not be included into the printed manuals. These last-minute changes and enhancements to the software and manuals are listed in the **Release Notes** shipped with the product.

Requesting Assistance

At Keil, we are committed to providing you with the best-embedded development tools, documentation, and support. If you have suggestions and comments regarding any of our products, or you have discovered a problem with the software, please report them to us, and where applicable make sure to:

1. Read the section in this manual that pertains to the task you are attempting,
2. Check the update section of the Keil web site to make sure you have the latest software and utility version,
3. Isolate software problems by reducing your code to as few lines as possible.

If you are still having difficulties, please report them to our technical support group. Make sure to include your license code and product version number displayed through the **Help – About** Menu of µVision. In addition, we offer the following support and information channels, accessible at ww.keil.com/support.

1. The Support Knowledgebase is updated daily and includes the latest questions and answers from the support department,
2. The Application Notes can help you in mastering complex issues, like interrupts and memory utilization,
3. Check the on-line Discussion Forum,
4. Request assistance through Contact Technical Support (web-based E-Mail),
5. Finally, you can reach the support department directly via support.intl@keil.com or support.us@keil.com.

Chapter 2. Developing With an RTOS

In the course of this chapter we will consider the idea of using RTX, the Keil small footprint RTOS, on an ARM processor-based microcontroller. If you are used to writing procedural-based C code on microcontrollers, you may doubt the need for such an operating system. If you are not familiar with using an RTOS in real-time embedded systems, you should read this chapter before dismissing the idea. The use of an RTOS represents a more sophisticated design approach, inherently fostering structured code development, which is enforced by the RTOS Application Programming Interface (API).

The RTOS structure allows you to take an object-orientated design approach while still programming in C. The RTOS also provides you with multithreaded support on a small microcontroller. These two features create a shift in design philosophy, moving us away from thinking about procedural C code and flow charts. Instead, we consider the fundamental program tasks and the flow of data between them. The use of an RTOS also has several additional benefits, which may not be immediately obvious. Since an RTOS-based project is composed of well-defined tasks, using an RTOS helps to improve project management, code reuse, and software testing.

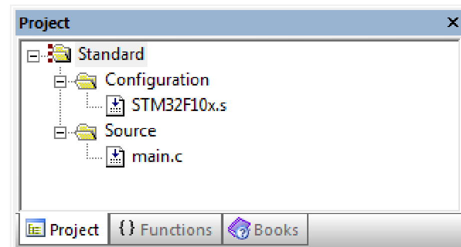
The tradeoff for this is that an RTOS has additional memory requirements and increased interrupt latency. Typically, RTX requires between 500 Bytes and 5KBytes of RAM and 5KBytes of code, but remember that some of the RTOS code would be replicated in your program anyway. We now have a generation of small, low-cost microcontrollers that have enough on-chip memory and processing power to support the use of an RTOS. Developing using this approach is therefore much more accessible.

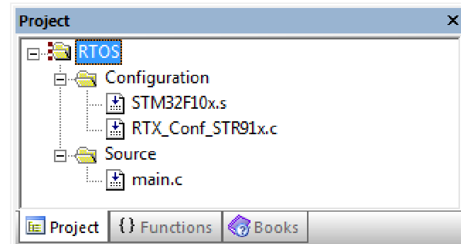
Getting Started

This chapter first looks at setting up an introductory RTOS project for ARM7, ARM9, and Cortex-M based microcontrollers. Next, we will go through each of the RTOS primitives and explain how they influence the design of our application code. Finally, when we have a clear understanding of the RTOS features, we will take a closer look at the RTOS configuration file.

Setting-Up a Project

The first exercise in the examples accompanying this book provides a PDF document giving a detailed step-by-step guide for setting up an RTX project. Here we will look at the main differences between a standard C program and an RTOS-based program. First, our µVision project is defined in the default way. This means that we start a new project and select a microcontroller from the µVision Device Database®. This will add the startup code and configure the compiler, linker, simulation model, debugger, and Flash programming algorithms. Next, we add an empty C module and save it as **main.c** to start a C-based application. This will give us a project structure similar to that shown on the right. A minimal application program consists of an Assembler file for the startup code and a C module.





The RTX configuration is held in the file **RTX_Config.c** that must be added to your project. As its name implies, **RTX_Config.c** holds the configuration settings for RTX. This file is specific to the ARM processor-based microcontroller you are using. Different versions of the file are located in **C:\KEIL\ARM\STARTUP**.

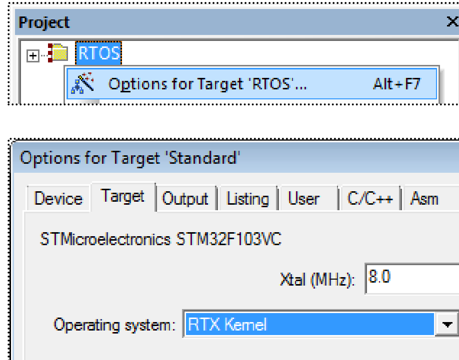
If you are using an ARM7 or ARM9-based microcontroller, you can select the correct version for the microcontroller family you are using and RTX will work "out-of-the-box". For Cortex-M-based microcontrollers there is one generic configuration file. We will examine this file in more detail later, after we have looked more closely at RTX and understood what needs to be configured.

To enable our C code to access the RTX API, we need to add an include file to all our application files that use RTX functions. To do this you must add the following include file in *main.*

```
#include <RTL.h>
```

We must let the µVision IDE utility know that we are using RTX so that it can link in the correct library. This is done by selecting "RTX Kernel" in the **Options for Target** menu, obtained by right clicking on "RTOS".

The RTX Kernel library is added to the project by selecting the operating system in the dialog Options for Target.



When using RTX with an ARM7 or ARM9 based microcontroller, calls to the RTOS are made by Software Interrupt instructions (SWI). In the default startup code, the SWI interrupt vector jumps to a tight loop, which traps SWI calls. To configure the startup code to work with RTX we must modify the SWI vector code to call RTX.

A part of RTX runs in the privileged supervisor mode and is called with software interrupts (SWI). We must therefore disable the SWI trap in the startup code. With Cortex-based microcontroller, the interrupt structure is different and does not require you to change the startup code, so you can ignore this step.

You must disable the default SWI handler and import the SWI_Handler used by the RTOS, when used with ARM7 or ARM9.

```
                IMPORT   SWI_Handler

Undef_Handler   B        Undef_Handler
;SWI_Handler    B        SWI_Handler           ; Part of RTL
PAbt_Handler    B        PAbt_Handler
DAbt_Handler    B        DAbt_Handler
IRQ_Handler     B        IRQ_Handler
FIQ_Handler     B        FIQ_Handler
```

In the vector table, the default SWI_Handler must be commented out and the SWI_Handler label must be declared as an import. Now, when RTX generates a software interrupt instruction, the program will jump to the SWI_Handler in the RTX library. These few steps are all that are required to configure a project to use RTX.

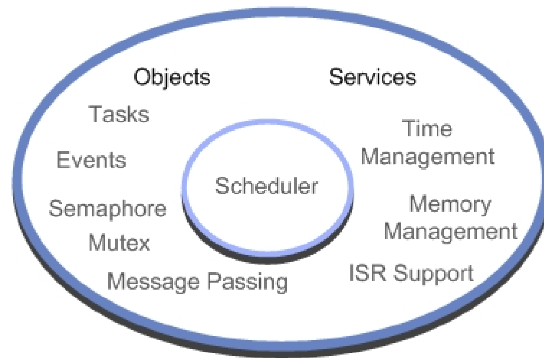
Exercise: First Project

The first RTOS exercise guides you through setting up and debugging an RTX-based project.

RTX Kernel

RTX consists of a scheduler that supports round-robin, pre-emptive, and co-operative multitasking of program tasks, as well as time and memory management services. Inter-task communication is supported by additional RTOS objects, including event triggering, semaphores, Mutex, and a mailbox system. As we will see, interrupt handling can also be accomplished by prioritized tasks, which are scheduled by the RTX kernel.

The RTX kernel contains a scheduler that runs program code as tasks. Communication between tasks is accomplished by RTOS objects such as events, semaphores, Mutexes, and mailboxes. Additional RTOS services include time and memory management and interrupt support.



Tasks

The building blocks of a typical C program are functions that we call to perform a specific procedure and which then return to the calling function. In an RTOS, the basic unit of execution is a "Task". A task is very similar to a C procedure, but has some fundamental differences.

Procedure	Task

`unsigned int procedure (void) {`
`  …`
`  …`
`  return (val);`
`}`
	`__task void task (void) {`
	`  for (;;) {`
	`    …`
	`  }`
	`}`

We always expect to return from C functions, however, once started an RTOS task must contain an endless loop, so that it never terminates and thus runs forever. You can think of a task as a mini self-contained program that runs within the RTOS. While each task runs in an endless loop, the task itself may be started by other tasks and stopped by itself or other tasks. A task is declared as a C function, however RTX provides an additional keyword *__task* that should be added to the function prototype as shown above. This keyword tells the compiler

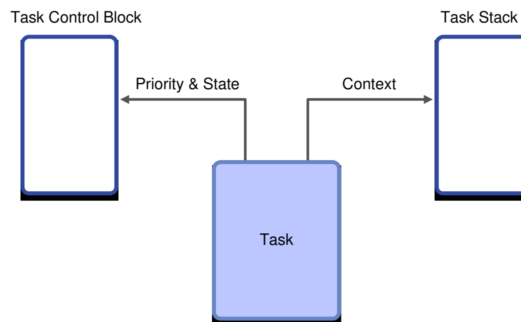
not to add the function entry and exit code. This code would normally manage the native stack. Since the RTX scheduler handles this function, we can safely remove this code. This saves both code and data memory and increases the overall performance of the final application.

An RTOS-based program is made up of a number of tasks, which are controlled by the RTOS scheduler. This scheduler is essentially a timer interrupt that allots a certain amount of execution time to each task. So task1 may run for 100ms then be de-scheduled to allow task2 to run for a similar period; task 2 will give way to task3, and finally control passes back to task1. By allocating these slices of runtime to each task in a round-robin fashion, we get the appearance of all three tasks running in parallel to each other.

Conceptually we can think of each task as performing a specific functional unit of our program, with all tasks running simultaneously. This leads us to a more object-orientated design, where each functional block can be coded and tested in isolation and then integrated into a fully running program. This not only imposes a structure on the design of our final application but also aids debugging, as a particular bug can be easily isolated to a specific task. It also aids code reuse in later projects. When a task is created, it is allocated its own task ID. This is a variable, which acts as a handle for each task and is used when we want to manage the activity of the task.

```
OS_TID id1, id2, id3;
```

In order to make the task-switching process happen, we have the code overhead of the RTOS and we have to dedicate a CPU hardware timer to provide the RTOS time reference. For ARM7 and ARM9 this must be a timer provided by the microcontroller peripherals. In a Cortex-M microcontroller, RTX will use the SysTick timer within the Cortex-M processor. Each time we switch running tasks the RTOS saves the state of all the task variables to a task stack and stores the runtime information about a task in a Task Control Block. The "context switch time", that is, the time to save the current task state and load up and start the next task, is a crucial value and will depend on both the RTOS kernel and the design of the underlying hardware.



Each task has its own stack for saving its data during a context switch. The Task Control Block is used by the kernel to manage the active tasks.

The Task Control Block contains information about the status of a task. Part of this information is its run state. A task can be in one of four basic states, RUNNING, READY, WAITING, or INACTIVE. In a given system only one task can be running, that is, the CPU is executing its instructions while all the other tasks are suspended in one of the other states. RTX has various methods of inter-task communication: events, semaphores, and messages. Here, a task may be suspended to wait to be signaled by another task before it resumes its READY state, at which point it can be placed into RUNNING state by the RTX scheduler.

At any moment a single task may be running. Tasks may also be waiting on an OS event. When this occurs, the tasks return to the READY state and are scheduled by the kernel.

Task	Description
RUNNING	The currently running TASK
READY	TASKS ready to run
WAIT DELAY	TASKS halted with a time DELAY
WAIT INT	TASKS scheduled to run periodically
WAIT OR	TASKS waiting an event flag to be set
WAIT AND	TASKS waiting for a group event flag to be set
WAIT SEM	TASKS waiting for a SEMAPHORE
WAIT MUT	TASKS waiting for a SEMAPHORE MUTEX
WAIT MBX	TASKS waiting for a MAILBOX MESSAGE
INACTIVE	A TASK not started or detected

Starting RTX

To build a simple RTX-based program, we declare each task as a standard C function and a TASK ID variable for each Task.

```
__task  void task1 (void);
__task  void task2 (void);
OS_TID  tskID1,  tskID2;
```

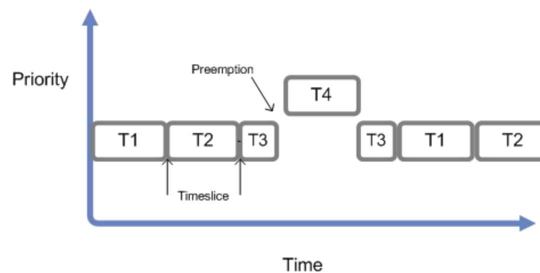
After reset, the microcontroller enters the application through the *main()* function, where it executes any initializing C code before calling the first RTX function to start the operating system running.

```
void main (void)  {

  IODIR1 = 0x00FF0000;        // Do any C code you want
  os_sys_init (task1);        // Start the RTX call the first task
}
```

The *os_sys_init ()* function launches RTX, but only starts the first task running. After the operating system has been initialized, control will be passed to this task. When the first task is created it is assigned a default priority. If there are a number of tasks ready to run and they all have the same priority, they will be allotted run time in a round-robin fashion. However, if a task with a higher priority becomes ready to run, the RTX scheduler will de-schedule the currently running task and start the high priority task running. This is called pre-emptive priority-based scheduling. When assigning priorities you have to be careful, because the high priority task will continue to run until it enters a WAITING state or until a task of equal or higher priority is ready to run.

Tasks of equal priority will be scheduled in a round-robin fashion. High priority tasks will pre-empt low priority tasks and enter the RUNNING state "on demand".



Two additional calls are available to start RTX; *os_sys_init_prio(task1)* will start the RTOS and create the task with a user-defined priority. The second OS call is *os_sys_init_user(task1, &stack, Stack_Size)*. This starts the RTOS and defines a user stack.

Creating Tasks

Once RTX has been started, the first task created is used to start additional tasks required for the application. While the first task may continue to run, it is good programming style for this task to create the necessary additional tasks and then delete itself.

```
__task void task1 (void)  {
tskID2 = os_tsk_create (task2,0x10);      // Create the second task
                                          // and assign its priority.
tskID3 = os_tsk_create (task3,0x10);      // Create additional tasks
                                          // and assign priorities.
os_tsk_delete_self ();                    // End and self-delete this task
}
```

The first task can create further active tasks with the *os_tsk_create()* function. This launches the task and assigns its task ID number and priority. In the example above we have two running tasks, *task2* and *task3*, of the same priority, which will both be allocated an equal share of CPU runtime. While the *os_tsk_create()* call is suitable for creating most tasks, there are some additional task creation calls for special cases.

It is possible to create a task and pass a parameter to the task on startup. Since tasks can be created at any time while RTX is running, a task can be created in response to a system event and a particular parameter can be initialized on startup.

```
tskID3 = os_tsk_create_ex (Task3, priority, parameter);
```

When each task is created, it is also assigned its own stack for storing data during the context switch. This task stack is a fixed block of RAM, which holds all the task variables. The task stacks are defined when the application is built, so the overall RAM requirement is well defined. Ideally, we need to keep this as small as possible to minimize the amount of RAM used by the application. However, some tasks may have a large buffer, requiring a much larger stack space than other tasks in the system. For these tasks, we can declare a larger task stack, rather than increase the default stack size.

```
static U64 stk4 [400/8];
```

A task can now be declared with a custom stack size by using the *os_tsk_create_user()* call and the dedicated stack.

```
tskID4 = os_tsk_create_user (Task4, priority, &stk4, sizeof (stk4));
```

Finally, there is a combination of both of the above task-creating calls where we can create a task with a large stack space and pass a parameter on startup.

```
static U64 stk5 [400/8];

tskID5 = os_tsk_create_user_ex (Tsk5, prio, &stk5, sizeof (stk5), param);
```

Exercise: Tasks

This exercise presents the minimal code to start the RTOS and create two running tasks.

Task Management

Once the tasks are running, there are a small number of RTX system calls, which are used to manage the running tasks. It is possible to elevate or lower a task's priority either from another function or from within its own code.

```
OS_RESULT os_tsk_prio (tskID2, priority);
OS_RESULT os_tsk_prio_self (priority);
```

As well as creating tasks, it is also possible for a task to delete itself or another active task from the RTOS. Again we use the task ID rather than the function name of the task.

```
OS_RESULT = os_tsk_delete (tskID1);
            os_tsk_delete_self ();
```

Finally, there is a special case of task switching where the running task passes control to the next ready task of the same priority. This is used to implement a third form of scheduling called co-operative task switching.

```
os_tsk_pass ();                    // switch to next ready to run task
```

Multiple Instances

One of the interesting possibilities of an RTOS is that you can create multiple running instances of the same base task code. For example, you could write a task to control a UART and then create two running instances of the same task code. Here each instance of UART_Task would manage a different UART.

```
tskID3_0 = os_tsk_create_ex (UART_Task, priority, UART1);
```

Exercise: Multiple instances

This exercise creates multiple instances of one base task and passes a parameter on startup to control the functionality of each instance.

Time Management

As well as running your application code as tasks, RTX also provides some timing services, which can be accessed through RTX function calls.

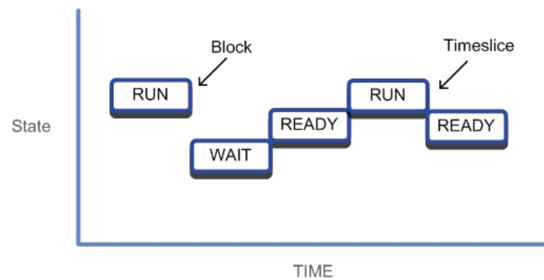
Time Delay

The most basic of these timing services is a simple timer delay function. This is an easy way of providing timing delays within your application. Although the RTX kernel size is quoted as 5K bytes, features such as delay loops and simple scheduling loops are often part of a non-RTOS application and would consume code bytes anyway, so the overhead of the RTOS can be less than it initially appears.

```
void os_dly_wait (unsigned short delay_time)
```

This call will place the calling task into the WAIT_DELAY state for the specified number of system timer ticks. The scheduler will pass execution to the next task in the READY state.

During their lifetime, tasks move through many states. Here, a running task is blocked by an *os_dly_wait()* call so it enters a WAIT state. When the delay expires, it moves to the READY state. The scheduler will place it in the RUN state. If its time slice expires, it will move back to the READY state.



When the timer expires, the task will leave the WAIT_DELAY state and move to the READY state. The task will resume running when the scheduler moves it to the RUNNING state. If the task then continues executing without any further blocking OS calls, it will be de-scheduled at the end of its time slice and be placed in the READY state, assuming another task of the same priority is ready to run.

Exercise: Time Management

This exercise replaces the user delay loops with the OS delay function.