Getting Started with

RapidiTTy MCU 2.3

TTE Systems

Rapid development of reliable embedded systems

http://www.tte-systems.com

Version

Getting Started with RapidiTTy MCU v2.3 (November 2009)

Copyright

This document is copyright © TTE Systems Limited 2007-2009. All rights reserved.

Trademarks

Altera® is a registered trademark of Altera Corporation.

ARM™ and Cortex™ are registered trademarks of ARM Limited.

Cygwin™ is a registered trademark of Red Hat, Inc.

Eclipse™ and Built on Eclipse™ are trademarks of the Eclipse Foundation, Inc.

GNU™ is a registered trademark of the Free Software Foundation.

Linux™ is a registered trademark of Linus Torvalds.

MIPS® is a registered trademark of MIPS Technologies Inc.

NXP™ is a trademark of NXP Semiconductors

RapidiTTy® and TTE® are registered trademarks of TTE Systems Ltd.

TTE Builder™ and TTE Debug™ are trademarks of TTE Systems Ltd.

Sun®, Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc.

Windows® is a registered trademark of Microsoft Corporation.

Xilinx®, ISE™, Spartan™, Virtex™ and WebPACK™ are trademarks or registered trademarks of Xilinx, Inc.

All other trademarks are acknowledged.

Table of Contents

1 Introduction....5
1.1 Overview of this guide....5
2 Installation and hardware requirements....7
2.1 JTAG debugging hardware....7
2.1.1 JTAG Wiggler driver installation....8
2.1.2 Olimex ARM-USB-OCD driver installation....8
2.1.3 Other USB JTAG devices....8
2.2 Alternative hardware....8
3 Working with projects....9
3.1 Workspaces....9
3.2 Project creation....9
3.3 Project-wide actions and settings....11
3.3.1 Altering configuration settings....12
3.3.2 Building and testing....14
4 Choosing a scheduler....17
4.1 Empty project....17
4.2 Fixed priority co-operative....18
4.3 Fixed priority hybrid....18
4.4 Fixed priority pre-emptive....19
4.5 Dynamic priority co-operative....19
4.6 Dynamic priority pre-emptive....19
5 Working with components....21
5.1 Component basics....21
5.1.1 Adding a new component....22
5.2 Connection basics....24
5.2.1 Making a connection....24
5.2.2 Connection internals....25
5.3 User components....26
5.3.1 Importing new tasks and ports....27
5.3.2 Pointer types....29
5.4 Drivers....30
6 Debugging a project....33
6.1 Basic debugging....33
6.1.1 Examining variables....34
6.1.2 Setting and examining breakpoints....34
6.1.3 Examining registers....35
6.2 Handling assertions....36
6.3 Targets without debugging support....37
7 Acquiring timing data....39
7.1 Creating a demonstration project....40
7.2 Acquiring timing statistics....40
7.3 Viewing timing statistics....41
7.3.1 An overview of the entire system....42
7.3.2 Details for individual tasks....43
7.4 Making changes....44
8 Observing memory usage with analytics....47
8.1 Memory meters....47

8.2 Detailed analytics reporting....48
9 Where do we go from here?....51
9.1 RapidiTTy MCU....51
9.2 Other products in the RapidiTTy family....51
9.2.1 RapidiTTy FPGA....51
9.2.2 RapidiTTy x86....52
10 Migrating and importing projects....53
10.1 Importing an existing project....53
10.2 Migrating from RapidiTTy Builder....54
10.2.1 The new build system....55
10.2.2 Generated startup and linker scripts....55
10.2.3 Relative paths in preprocessor includes....56
11 Troubleshooting....57
11.1 Problems uploading and debugging....57
11.2 Common causes of runtime problems....57
11.3 Problems with timing analysis....58
11.4 Finding potential stack overflows....58

1 Introduction

RapidiTTy MCU™ is a tightly integrated tool-suite which supports the rapid development of code for "Commercial off-the-Shelf" (CotS) microcontrollers with ARM7® and Cortex-M3™ cores. In the current version of the tool (2.3), full support is provided for the LPC-2xxx and STM32F10xxx families of devices. Preliminary support is also provided for the LPC-17xx family and for the TTE32-SM3 (running on the Altera DE2-70 FPGA development board).

RapidiTTy MCU provides the following key benefits for developers who wish to create, test and maintain reliable and resource efficient embedded systems:

- Component-based design with the TTE Builder™ engine and an extensive library of source-code – system prototypes can be created with just a few clicks of a mouse.

- The powerful TTE Statistics toolbox, which provides important timing information like task execution, period and jitter, all based on real measurements from you own code running on your target hardware.

- Automated static analysis tools provide detailed estimates of the worst-case stack usage for the system, helping to prevent hard-to-find stack overflow problems before they become an issue.

- Full support for an integrated and coherent range of thin, resource-efficient operating systems (all royalty free, all fully integrated with the toolset).

Like all RapidiTTy toolsets, RapidiTTy MCU is based on Time-Triggered (TT)[1] technology. Use of TT technology helps to ensure that even new developers can produce reliable embedded systems, and helps to maximise the efficiency of an experienced development team.

1.1 Overview of this guide

In this guide, we will explore the key benefits of RapidiTTy MCU by illustrating how this tool can be used to develop a simple example system.

Before beginning the tutorial, RapidiTTy MCU must first be installed and configured to work with our hardware. We describe how to do this in the next section.

1 More information about time-triggered systems can be found online at the TTE Systems website: http://www.tte-systems.com/books/pttes.

2 Installation and hardware requirements

RapidiTTy MCU includes an automated installer that should take care of all the necessary software installation tasks. In addition, we must connect and configure a development board for the target embedded hardware platform.

In this tutorial, we will target the popular NXP LPC-2378 microcontroller. More specifically, we will be using the LPC-2378-STK development board from Olimex.[2] In order for RapidiTTy MCU to be able to program this board, it must be connected to the desktop computer with a JTAG link.

2.1 JTAG debugging hardware

There are various choices when it comes to connecting an embedded development board to a computer via JTAG. RapidiTTy MCU Supports the use of a JTAG Wiggler device, and the Olimex ARM-USB-OCD device; examples of both of these devices are shown in Figure 2-1. In addition, USB debug devices from Amontec – JTAGkey and JTAGkey-Tiny – are also supported.

Figure 2-1: A JTAG Wiggler device (left) and an ARM-USB-OCD device (right).

As we can see in Figure 2-1, the JTAG Wiggler connects to a desktop computer via a parallel port, whereas the Olimex ARM-USB-OCD is a USB-based device. Beyond this distinction, there is little practical difference between the two in how they are used, but some additional work is needed to install their required drivers.

2 Available online from the Olimex website: http://www.olimex.com.

2.1.1 JTAG Wiggler driver installation

Drivers for the JTAG Wiggler are included with RapidiTTy MCU – simply use the following procedure to install them:

1. Locate the RapidiTTy MCU installation directory.
2. Open the "`OpenOCD\drivers\parport`" subdirectory.
3. Run the "`install_giveio.bat`" executable (double click on it).

A command prompt window will open and close again when the operation is complete. Unfortunately, this can happen too quickly to determine if it has worked, but it should be successful provided you have administrator access to the computer.

2.1.2 Olimex ARM-USB-OCD driver installation

The Olimex ARM-USB-OCD is a USB device, so it will prompt for a driver immediately after it has been connected for the first time. The driver can be located in the RapidiTTy MCU installation directory, under "`OpenOCD\driver\arm_usb_ocd`". Windows may prompt for two additional drivers – if so, simply point it to the same directory each time to complete the installation.

2.1.3 Other USB JTAG devices

Other USB devices, such as the Amontec JTAGkey and JTAGkey-Tiny (as shown in Figure 8-1), can be installed using the same method as the Olimex ARM-USB-OCD device. The drivers for all supported USB devices can be found as subdirectories of the "`OpenOCD\driver`" folder.

2.2 Alternative hardware

If we are targeting a device that RapidiTTy MCU provides only preliminary support for, then an alternative interface may be required. Currently, the use of the TTE32-SM3 on an Altera DE2-70 board is supported through a standard USB A-to-B cable. Debugging is fully supported for this target.

In order to upload code onto an NXP LPC-17xx device, we can use the Embedded Systems Academy (ESA) Flash Magic tool[3] to erase the flash memory and upload a hex file over a standard RS-232 serial connection. See Section 6.3 for details.

3 ESA Flash Magic can be downloaded from http://www.flashmagictool.com.

3 Working with projects

3.1 Workspaces

We can start by opening RapidiTTy MCU. The first time we do this, we are presented with a dialog asking us to choose a "workspace", as shown in Figure 3-1.

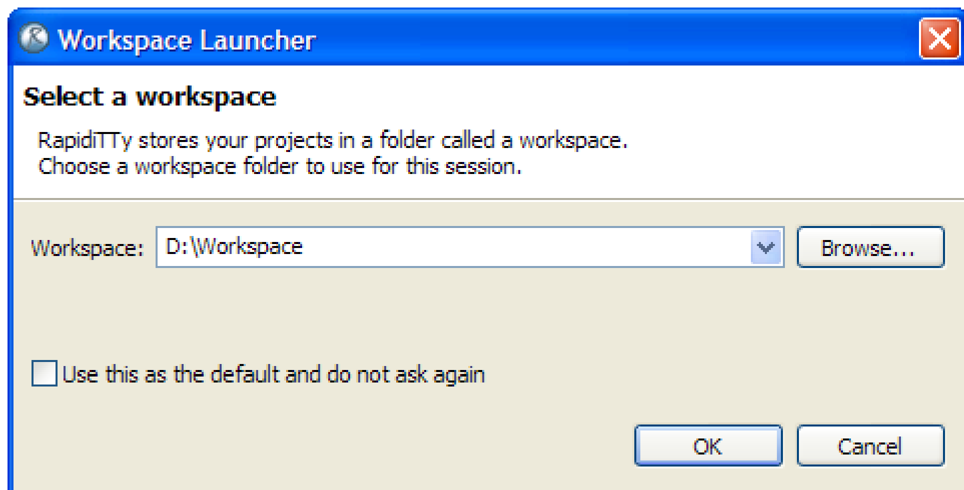

Figure 3-1: The workspace selection dialog.

The workspace is simply the directory under which all current projects are stored. For this tutorial, we will be using "`D:\Workspace`", but any value may be used as long as you know where the projects are being stored for future reference.

3.2 Project creation

We can create a new RapidiTTy MCU project by selecting "`New → RapidiTTy Project`" from the `File` menu (or by clicking the new project button on the left of the toolbar). This results in the new project dialog, shown in Figure 3-2.

> For this tutorial we will be starting with new projects. If you were using RapidiTTy Builder and wish to upgrade to RapidiTTy MCU, or if you have a project from an older version of RapidiTTy MCU, see Section 10 on page 53 for full details of the migration process.

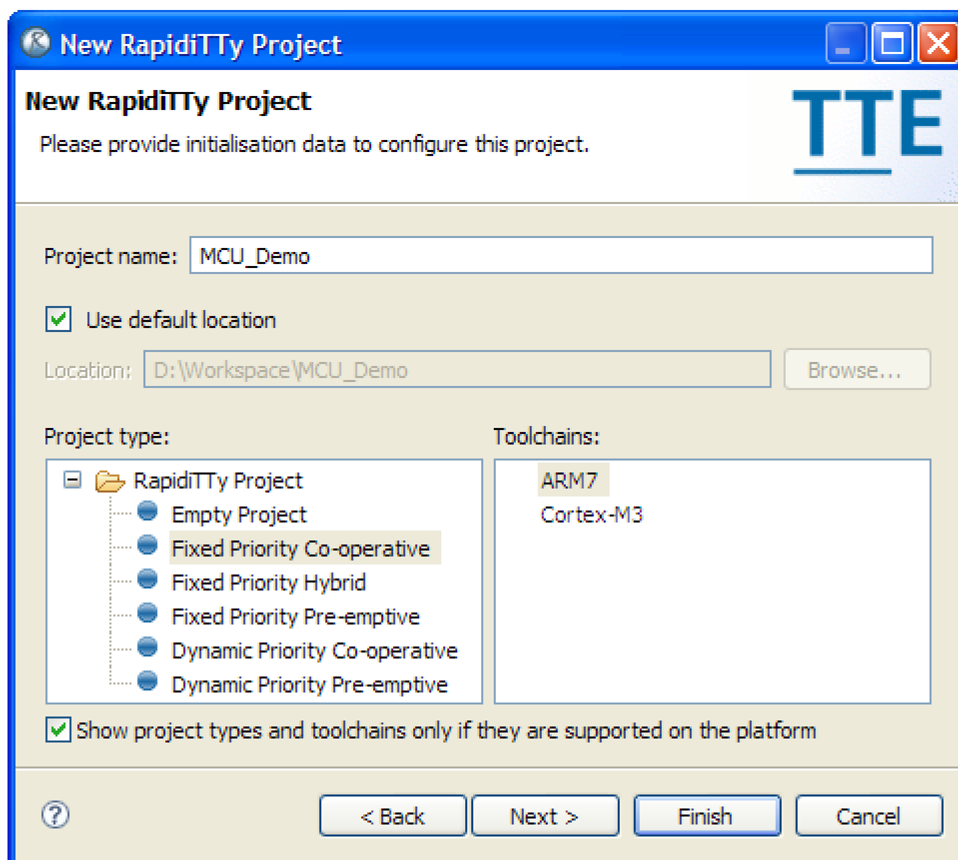

Figure 3-2: The new project dialog.

In this case, we have chosen to name the project "MCU_Demo". This name will also be the default name of the compiled binary that is created, should it be required outside of RapidiTTy MCU.

Once we have selected an appropriate name for the project we must decide on the "project type" that we will use, which essentially selects the source-code template that is generated for the initial project. In Figure 3-2, we have selected the "Fixed Priority Co-operative" scheduler, which can be seen as a very simple operating system that will execute individual tasks periodically, one after another.

> The other scheduler options can provide a basic operating system for a wide variety of different situations. See Section 4 on page 17 for more information.

The following page allows us to define or alter the desired build configurations for the project. For this demonstration, as with a great many projects, we can simply make do with the default "debug" and "release" configurations.

Once we press "finish", the project will be created and RapidiTTy MCU will compile the generated source-code with all the selected build configurations. Once this

process is complete, we can examine the source-code in the project explorer, as shown in Figure 3-3.

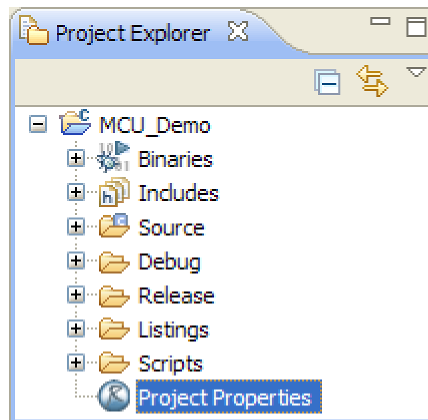

Figure 3-3: The project explorer for a newly created project.

Figure 3-3 shows some of the folders and files generated at project creation. One of the most important of these is the Project Properties file, which is selected in the screenshot. This allows us to carry out actions and alter settings that will affect the project as a whole.

3.3 Project-wide actions and settings

The project properties view can be opened at any time by double-clicking on the “Project Properties” file for the project, in the project explorer view. It is used to alter many of the available project-wide settings. From here, we can alter settings for the microcontroller target (including selecting the actual device that we are using), rebuild and start debugging the project, setup the automatic generation of startup code and linker scripts, and configure pre-built software components to be included in the project.

> Whenever changes are made in this view, it must be “saved” before the changes will be applied to the remainder of the project. Changes made to the configuration may alter the relevant generated source files – this saving mechanism allows us to experiment with the configuration options in a temporary way, without affecting the code.

The overall layout of the view is shown in Figure 3-4.

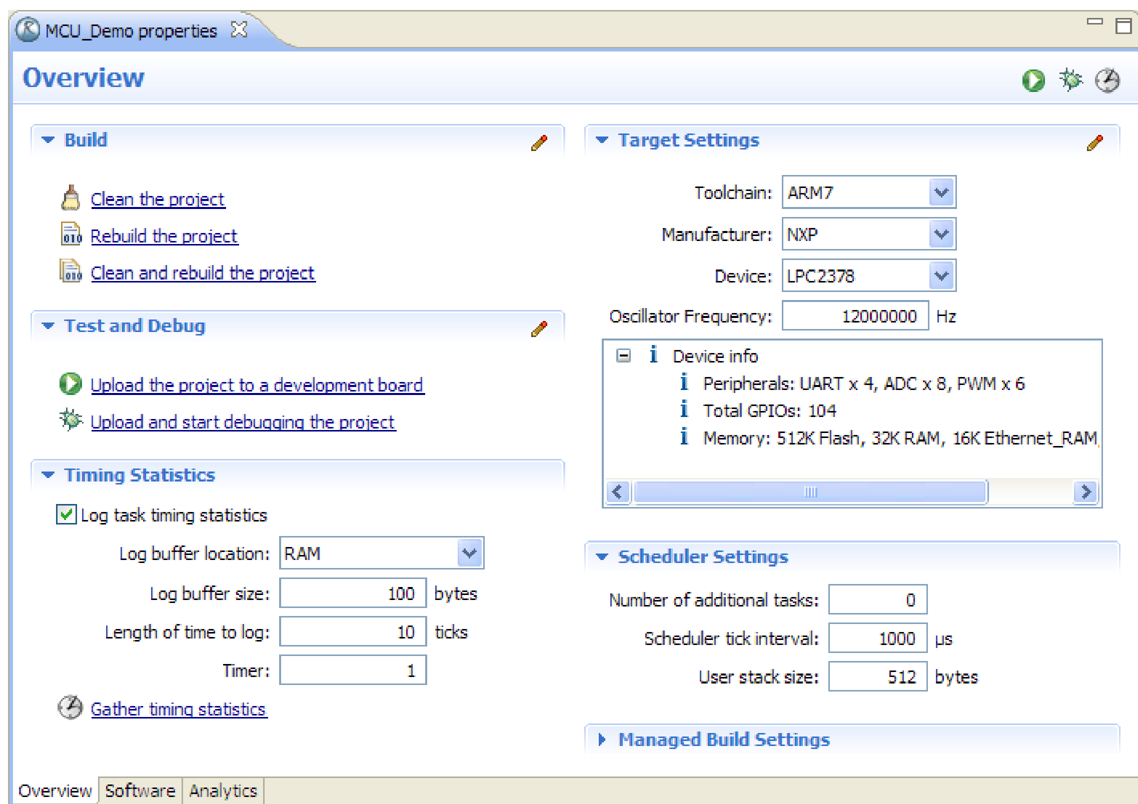

Figure 3-4: The initial project properties view.

In the bottom-left of Figure 3-4, we can see tabs for the “Overview” and “Software” pages. The overview page contains details of the currently selected target and the scheduler configuration, as well as providing easy access to frequently used actions, such as building and debugging.

> Many of the sections in the project properties view have additional configuration options available. These can be accessed by pressing the “pencil” icon in the top-right corner of the section, shown in Figure 3-4.

3.3.1 Altering configuration settings

The project properties view allows access to functionality that affects the project as a whole, such as the choice of microcontroller manufacturer and device. These options are controlled from the “Target Settings” section (shown in Figure 3-5), which also allows us to set the oscillator frequency that we are using. Simply save the project properties file to apply the changes to the rest of the project.

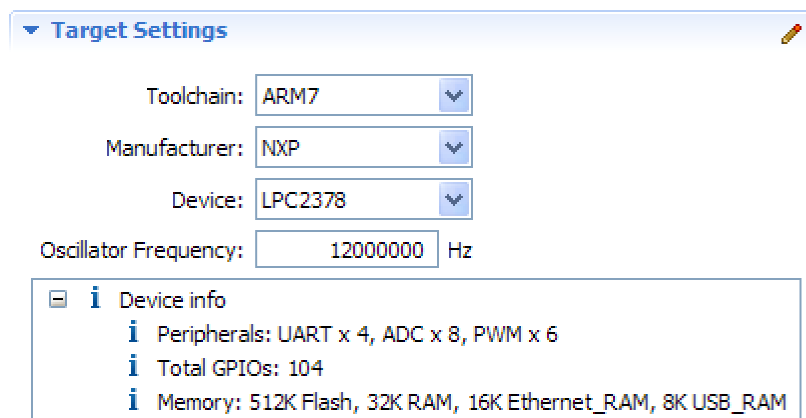

Figure 3-5: The target settings section.

Clicking on the "pencil" icon at the top-right of the target settings section will bring up the tool settings dialog. This allows us to alter the specific compiler, assembler and linker options that will be used in building the project, as shown in Figure 3-6.

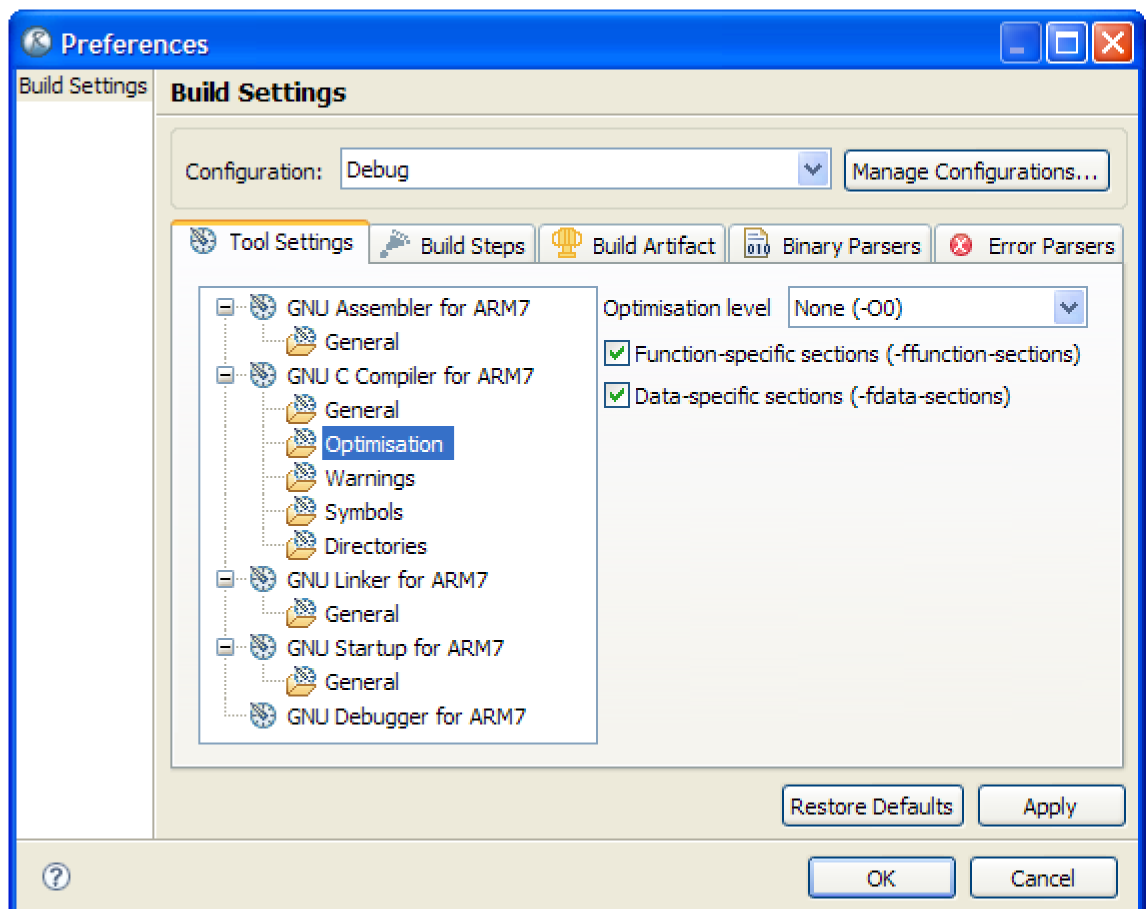

Figure 3-6: The build settings dialog.

Figure 3-5 shows that the target settings section displays information about the currently selected target. If a device is selected that does not have the peripherals

needed by the software components in our system, an error will be displayed in this area.

> The project properties file cannot be saved while there are errors. These must first be addressed by switching to the software page and re-configuring any components for which errors are present. If components rely on peripherals that are not available in the current device, they must be removed (or a different device selected) before the project properties can be saved.

3.3.2 Building and testing

To the left of the target settings section is the "Build" section, as shown in Figure 3-7. Here we can clean all generated binaries and objects files, rebuild any source files that were altered since the last build, or clean and then rebuild the entire project.

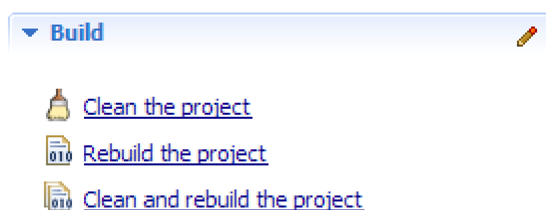

Figure 3-7: The build section.

Once the source-code has been compiled and an executable has been created, we must have a way of testing and debugging the application. This is provided by the "Test and Debug" section, as shown in Figure 3-8.

Figure 3-8: The test and debug section.

The first time we upload the code, RapidiTTy MCU will show the Run dialog and ask us to enter the desired "debug device". This is simply the choice of JTAG device, as described in Section 2.1. Figure 3-9 shows the selection being made.

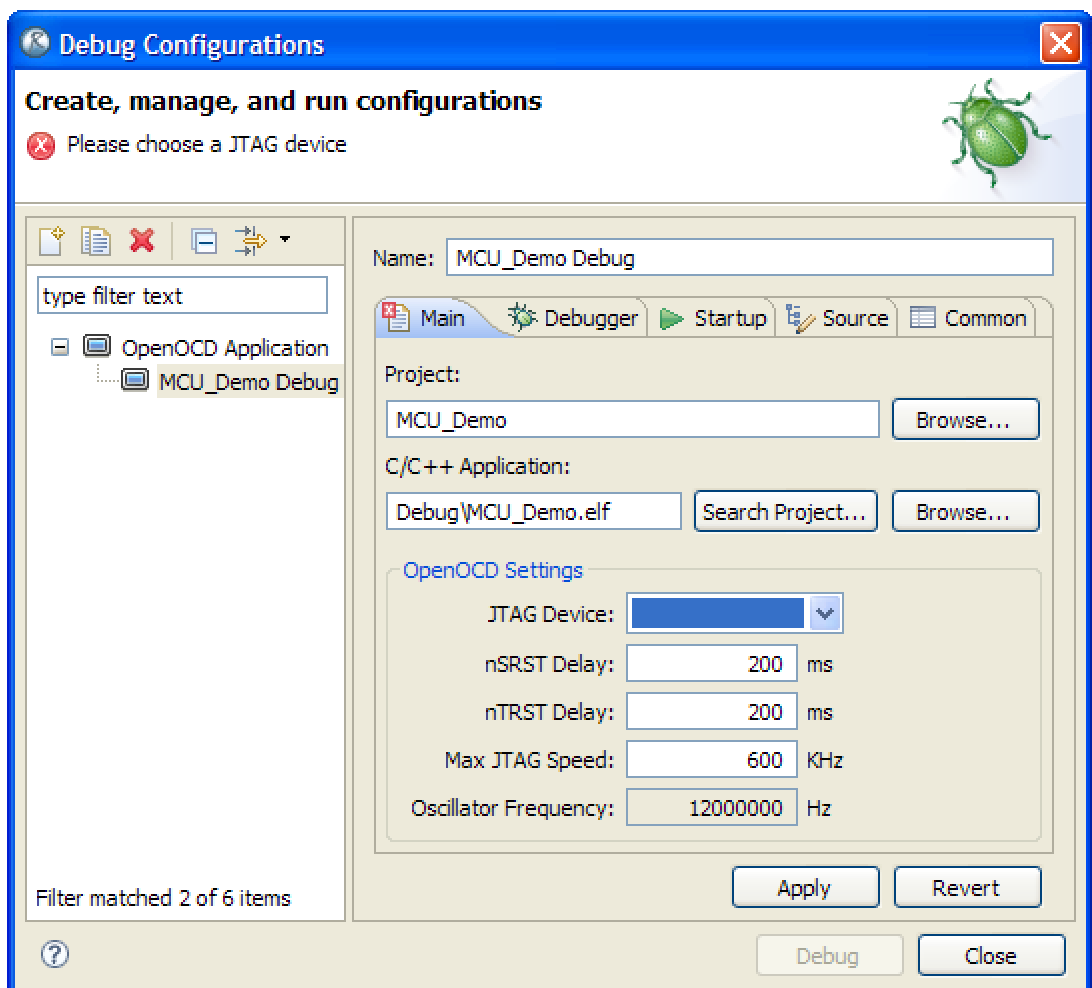

Figure 3-9: Selecting a JTAG device from the debug dialog.

The dialog in Figure 3-9 can be brought up at any time by clicking the "pencil" icon located at the top-right of the test and debug section, as shown in Figure 3-8.

> If an attempt to upload or debug should fail, it is often the result of incorrect settings in the launch configuration dialog (shown in Figure 3-9). The most common solution for such problems (excepting those caused by hardware failure or misconfiguration), is to try altering the JTAG speed setting. See also the troubleshooting advice in Section 11.

> The oscillator frequency cannot be altered in this dialog – it is here for information only. Instead it should be changed from the target settings section of the project properties view.

Below the test and debug section is the timing statistics section, which is detailed in Section 7 on page 39.

4 Choosing a scheduler

As discussed in Section 3.2 above, there are a number of schedulers to choose from. One of these must be chosen at project creation, as shown in Figure 4-1.

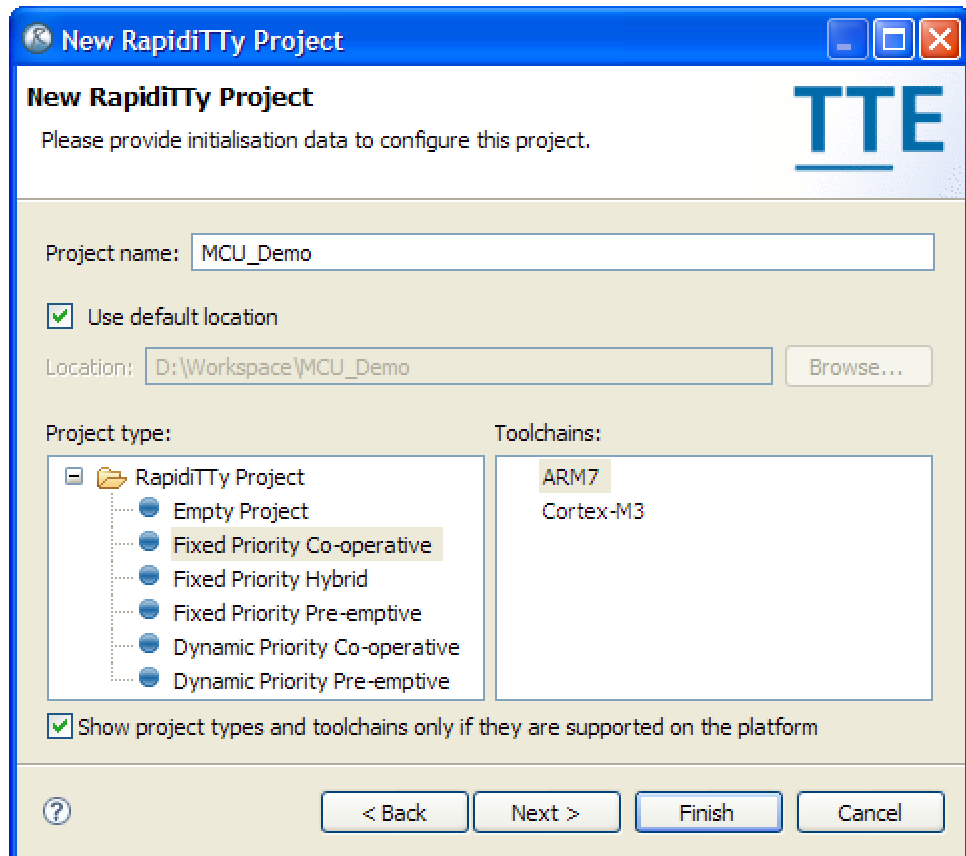

Figure 4-1: Choosing a scheduler at project creation.

This section details each project type and the scheduler it employs. The remainder of this guide assumes that a fixed priority co-operative scheduler was chosen.

> The choice of scheduler can be changed at any time after the project has been created, from the project properties page.

4.1 *Empty project*

The empty project option provides a minimal skeleton consisting of just enough code to compile and debug an embedded application and nothing else. It includes the generation of basic startup and linker scripts, as well as a main source file con-

taining the application entry point (the `main` function) and functions to either handle or trap any interrupts that might occur.

> Empty projects are ideal for simple testing or systems that do not need a scheduler – they still have access to the drivers (see Section 5.4).

4.2 *Fixed priority co-operative*

The Fixed Priority Co-operative (FPC) scheduler will execute tasks after a given delay and periodically at a given interval, based on multiples of the system tick. Each task will run to completion before the task with the next highest priority (that is due to run) begins. Task priorities are specified explicitly and cannot change at runtime.

A fully co-operative approach provides certain advantages – the lack of context switching means that the scheduler's operation is very simple and easy to fully comprehend. Similarly, there is no need for critical sections or locking and all tasks may share a single stack (which saves a considerable amount of memory).

> FPC schedulers are useful for any set of tasks where the total combined (worst-case) execution time of all tasks due to execute in a given tick is less than the scheduler's tick period.

4.3 *Fixed priority hybrid*

The Fixed Priority Hybrid (FPH) scheduler will execute most tasks in exactly the same way as the FPC scheduler. There is one exception: a single task may be designated as pre-emptive and allowed to interrupt any other task that is still executing when it is due to run.

The introduction of a single pre-emptive (time-triggered) task will only moderately increase the complexity of a scheduler, as the task may be executed directly from the scheduler's ISR and so benefit from the microcontroller's built-in context switch (as well as utilising the separate ISR stack).

> FPH schedulers are useful when we have one high-priority task with a short worst-case execution time that must execute at a very high frequency (compared to the other tasks in the system).

4.4 Fixed priority pre-emptive

The Fixed Priority Pre-emptive scheduler will allow any task to interrupt any other task with a lower priority. The FPP scheduler requires a priority for each task.

> FPP schedulers are useful when a co-operative or hybrid solution would be inappropriate. Note also that each task needs a separate stack area.

4.5 Dynamic priority co-operative

The Dynamic Priority Co-operative scheduler operates in the same basic manner as the FPC scheduler, except that the task's priorities are determined dynamically based on the Earliest Deadline First (EDF) approach.

Each task in the DPC scheduler must be provided with a deadline, which represents the time (within each period) that the task should be completed by.

> DPC schedulers are useful when the execution time of tasks is highly variable or cannot be known prior to execution.

4.6 Dynamic priority pre-emptive

The Dynamic Priority Pre-emptive (DPP) scheduler is based on the FPP scheduler, with the difference that it determines task's priorities dynamically based on the EDF approach (described in Section 4.5 above).

> DPP schedulers are useful when a co-operative or hybrid solution would be inappropriate. Note also that each task needs a separate stack area.

5 Working with components

In order to assist in quickly creating a working prototype system, RapidiTTy MCU provides two levels of support: components and drivers. Components are made up of high-level device independent source-code that can be combined in numerous different ways in order to build a system. Components are connected together through "ports", which could be either a global variable or a function.

5.1 Component basics

Components can be added, configured and removed from the "Software" page of the project properties view (reached using the tabs shown at the bottom-left of Figure 3-4 and Figure 5-1).

An overview of the software properties page, populated with a few typical components and connections, is shown in Figure 5-1.

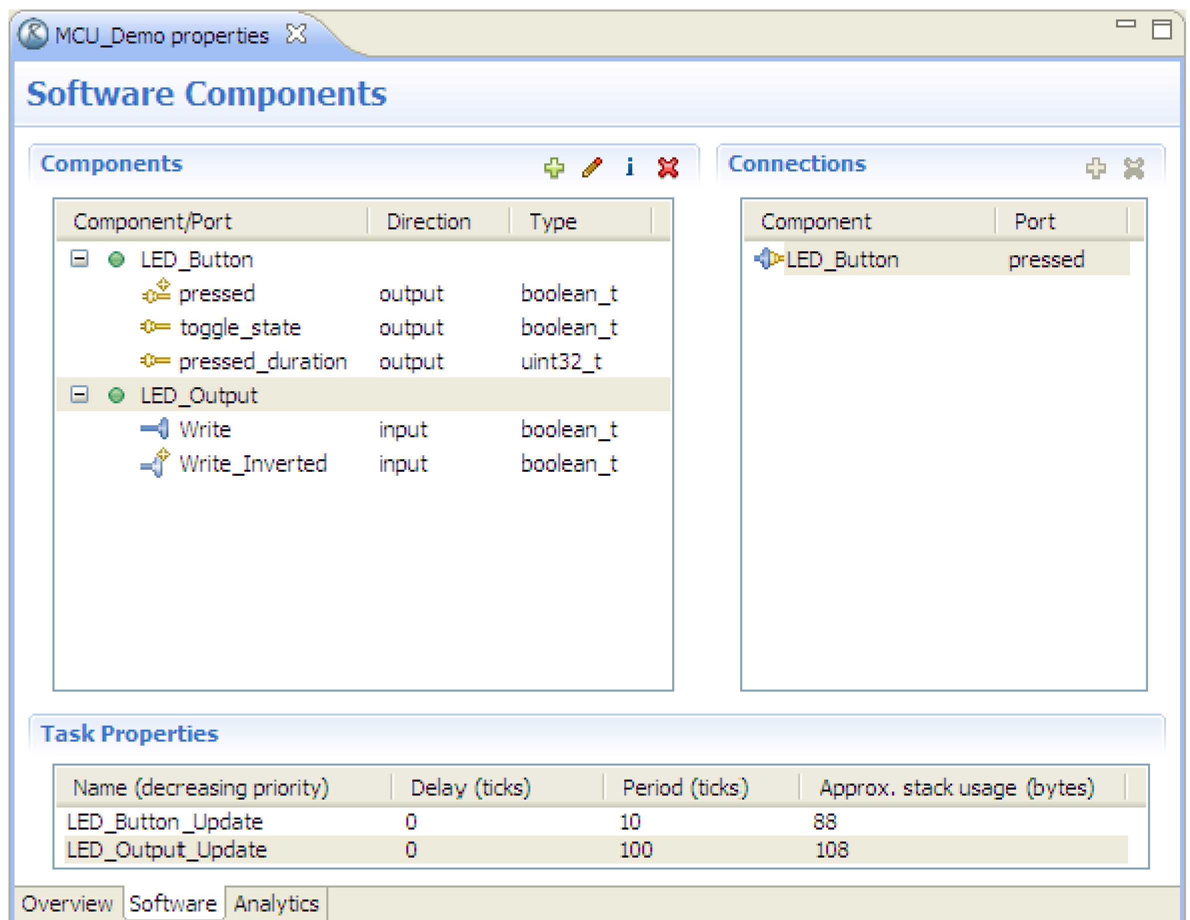

Figure 5-1: The software properties page.

The software properties page, shown in Figure 5-1, contains three main sections. The first of these, the "Components" section (in the top-left) lists all the components in use in the current project, along with their available ports. At the top of the section are buttons for adding new components, configuring existing components, displaying information about the currently selected component and deleting any selected components.

The "Connections" section (in the top-right) is linked to the components section. When a port is selected in the components section, the connections section displays any other ports that it is connected to. At the top of the section are buttons to add and remove connections.

Finally, the "Timing" section shows the tasks associated with the components in the system. From here we can alter the period at which the task will be executed by the scheduler and also the delay before the task will begin. The timing section is also linked to the components section, so selecting a component will highlight the associated tasks. Tasks have an implicit priority that is represented here by their order in the table – this can be adjusted by dragging and dropping tasks to reorder them. We can also see the stack usage of the task – see Section 8.2 for details.

5.1.1 Adding a new component

A new project will start with no components, connections, or tasks. Clicking the "plus" button in the components section will open a dialog that allows us to choose which component we wish to add. After making a selection, a separate dialog will prompt for configuration options – shown in Figure 5-2 for the switch component.

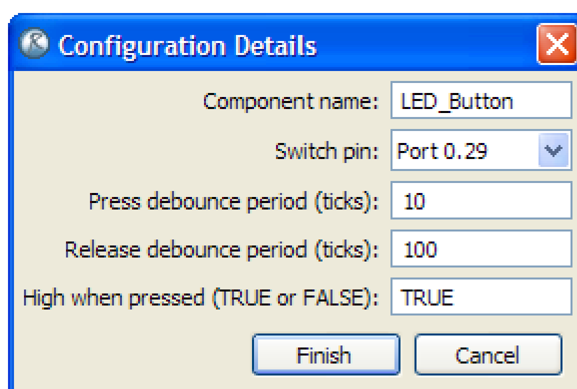

Figure 5-2: Configurations details for the switch component.

There are three types of options that may be required: pins, peripherals, and preferences. In Figure 5-2 we can see that the switch component requires one pin and three preferences to be specified. Pins and peripherals are hardware resources and, as such, are afforded special treatment – RapidiTTy MCU will detect the case

where multiple components are attempting to use the same resource and will provide a warning.

This can be seen as we attempt to add a “GPIO_Output” component to the system. The switch we have already added is using Port 0.29, which is may also be selected for the gpio output component. This can be seen in Figure 5-3.

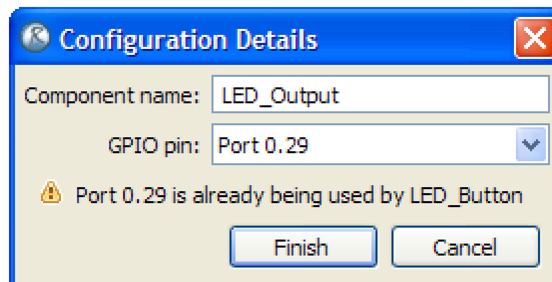

Figure 5-3: Configuring the GPIO_Output component, with a warning.

The specific development board we used for this example was an Olimex LPC-2378-STK, which has a button on Port 0.29 and a LED on Port 0.21. Changing the configuration of LED_Output to reflect this removes the warning and, after saving, leaves us with two complete components and their associated tasks.

The generated source-code will be placed in the project, as shown in Figure 5-4.

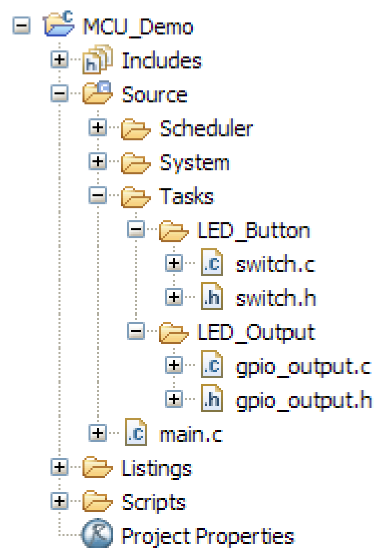

Figure 5-4: The location of the generated source-code in the project.

5.2 Connection basics

The code can be compiled and the resulting executable can be uploaded to the development board, tested and debugged. However, in their current state the components don't actually do anything – to fix this we need to connect them together.

5.2.1 Making a connection

Components are connected together through their ports. Ports are specific to a task, which is part of the component. Each of the components that we have added so far has a task, which will have its own ports. These are shown in Figure 5-5.

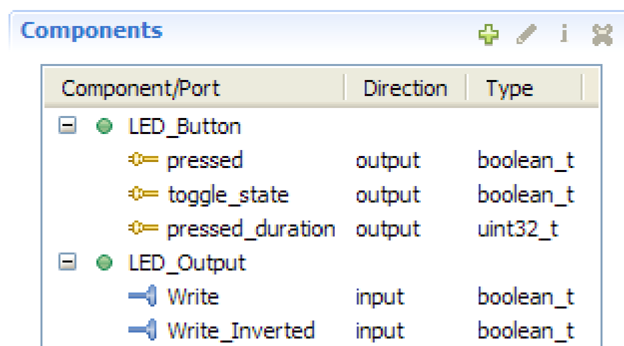

Figure 5-5: The components section, showing the available ports.

Each of the five ports shown in Figure 5-5 has both a type and a direction associated with it. A gpio output component (such as LED_Output) simply takes a value provided to it and outputs it to a port pin, which is why its ports are all inputs and all have the type "`boolean_t`".

To connect the "Write_Inverted" port, we either select it and click on the "plus" button in the connections section, or simply double-click on the port itself. This results in the port connection dialog, as seen in Figure 5-6.

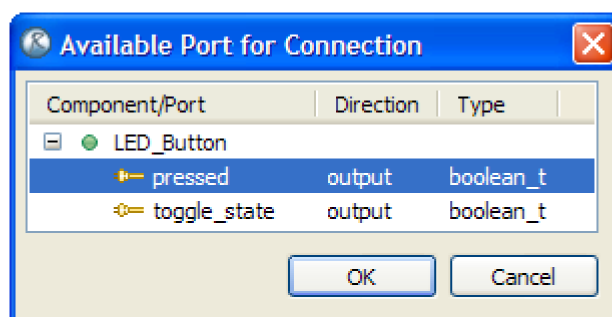

Figure 5-6: The port connection dialog.

Figure 5-6 shows the port connection dialog, which simply lists all the components and ports that would make a valid connection. The "pressed_duration" port is not available, because its type ("uint32_t") is incompatible with the type of the port we are connecting to ("boolean_t"). Also, the "Write" port is not listed because it has an incompatible direction.

Note that a port can only be connected to another port that has the *same* type, but the *opposite* direction. Connections are controlled by the output port, so data only really flows between components in a single direction.

5.2.2 Connection internals

Every component has one or more tasks, which will in turn have ports associated with them. A task corresponds to a function that is executed by the scheduler; if a task has one or more output ports associated with it, then it will pass values for those ports to its output function.

The output function is defined in the components header file. It will transfer the latest values of the component's output ports to any connected input ports. For example, the LED_Button component, once connected to the "Write_Inverted" port of LED_Output, will have the output function shown in Figure 5-7.

```
static inline void LED_Button_Update_Output(
        const boolean_t pressed,
        const boolean_t toggle_state,
        const uint32_t pressed_duration)
{
    LED_Output_Write_Inverted(pressed);
}
```

Figure 5-7: The output function for the Update task of LED_Button.

Notice that the parameters of the output function match the output ports for the task and component that it belongs to. All output ports are actually variable ports – only input ports may be functions.

The LED_Output_Write_Inverted function is defined in the source-file of the LED_Output component and, as you can probably guess, takes a "boolean_t" parameter. This is because it is a function-based port with a type of "boolean_t".

If "Write_Inverted" was a variable, it would be defined as a global and the output function would use a simple assignment instead of a call.

> In order to adapt to changes in the connections, component header files may be regenerated whenever the project properties are saved. For this reason, the header files should not be modified – any required additions may be made to the source-files instead.

5.3 User components

It is possible to create our own components by selecting the "User Component" option in the component selection dialog. This results in the initial user component configuration dialog, shown in Figure 5-8.

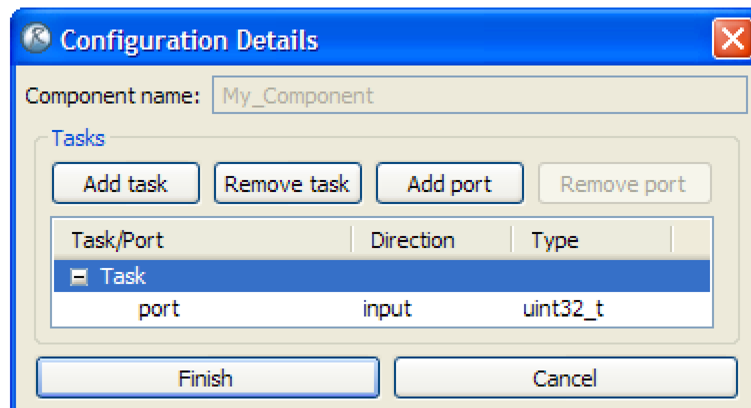

Figure 5-8: Creating a user component with one task and one port.

Figure 5-8 shows a new user component being created. So far, one task (with one port) has been added. Tasks can be added with the "Add task" button, which results in the dialog shown in Figure 5-9.

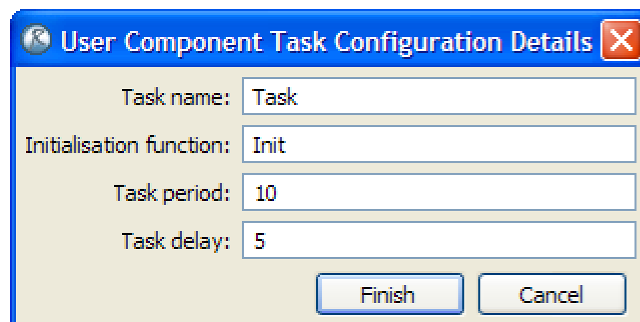

Figure 5-9: Creating a new task in a user component.

If a task is selected in the user component configuration dialog (in Figure 5-8), then we can add a port to it. This is shown in Figure 5-10.

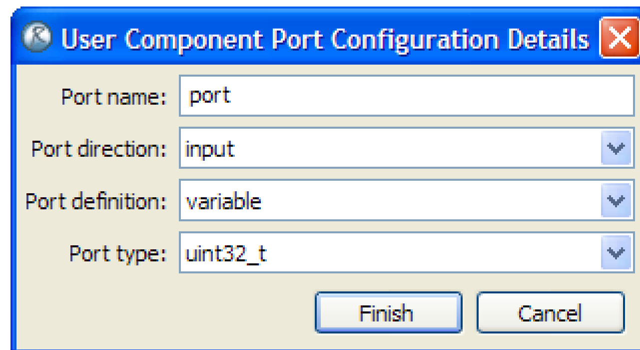

Figure 5-10: Adding a new port to a user component.

Once a user component has been created, the source-code will be generated and then cannot be changed – as any modifications would be lost. However, a user component can be updated (purely in the software view) to match the source-code by removing and *importing* both tasks and ports.

5.3.1 Importing new tasks and ports

New tasks and ports can be created by directly modifying the source code for the component (that is, the "`.c`" file). Once such modifications are made (and saved), they can be imported back into the system from the user component configuration dialog.

However, in order for new tasks and ports to become available for importing, they must follow certain rules. For example, tasks are always represented by functions that take no parameters and return nothing; but they are also always prefixed with the name of the component (e.g. "`void My_Component_Task(void)`").

Ports are more complicated – output ports are always variables that match a specific type (typical choices are shown in Table 1, but availability varies by architecture). Outputs can be variables local to the main task function, fully global or local to the task's translation unit (i.e. declared globally but with the static specifier).

Allowed Type	Description
`char_t`	Single character
`uint8_t`	8-bit unsigned integer
`uint16_t`	16-bit unsigned integer
`uint32_t`	32-bit unsigned integer
`uint64_t`	64-bit unsigned integer
`char_t*`	Pointer to zero terminated string of characters
`array_t*`	Pointer to array (see array driver for details)
`queue_t*`	Pointer to queue (see queue driver for details)
`float`	32-bit floating point value
`double`	64-bit floating point value
`boolean_t`	Boolean value (`TRUE` or `FALSE`)
`adc_data_t`	Raw ADC data (see ADC driver for details)

Table 1: Typical allowed types for ports on ARM7.

Output ports can be any variable that is local or declared static, but for a global variable to be allowed it must be prefixed with the name of the component. Such global variables are also valid as input ports, as are functions that are prefixed with the component's name. Some examples of valid ports are shown in Figure 5-11.

```
// Name: outport1, type: uint8_t, dir: output
static uint8_t outport1;

// Name: EitherPort, type: uint32_t, dir: either
typedef uint32_t my_type_t;
my_type_t My_Component_EitherPort;

// Name: InPort, type: char_t*, dir: input
void My_Component_InPort(char_t* str) {}

void My_Component_Task(void)
{
	// Name: outport2, type: adc_data_t, dir: output
	adc_data_t outport2;

	// Any output ports should be added as parameters to this
	// function call (in the order that they were imported):
	My_Component_Task_Output();
}
```

Figure 5-11: Examples of valid ports in task source-code.

In Figure 5-11, note that the type for a function-based port is that of its parameter; only one parameter is allowed for function-based ports, and they must not return a value.

If we now return to the user component configuration dialog (by double-clicking on the component, or selecting it and pressing the edit button), ports can be imported to a specific task by selecting the task and pressing “Import port”. Full details of each port will be taken from the source-file and automatically filled in for us.

Once an output port has been added to a task, it will also be added as a parameter to the output function in the task's header file. These parameters are in the order that the ports have been added to the task (the order listed in the components section), and must be added to the function call in the task's function manually (as RapidiTTy MCU will not modify the source-file once it has been created – only the header is regenerated when it must update connections or properties).

5.3.2 Pointer types

Of the types shown in Table 1 above, three are based on pointers. The first is simple and represents the standard C99 string type. Arrays and queues are more complex data structures that are accessed through their associated drivers – see the driver documentation for more details. Essentially they avoid the need for dynamic memory allocation by having the user pass in a byte buffer which they then use for their internal storage.

Each function in the array and queue drivers takes a pointer to the associated data structure and performs some operation on it. If we create a user component with an array or queue input port, we will have access to this pointer and so can simply use the driver functions on it directly.

Creating a user component with an array or queue as an output port is slightly trickier. First, we must have the actual queue object within the component's source file and also the buffer that the queue will operate on. We initialise the queue in the component's initialisation routine and send the *address* of the object to the task's output. This can be seen in the UART component, shown in Figure 5-12.

```
// Create the queue object and its internal buffer.
static queue_t receive_queue;
static uint8_t receive_buffer[UART_RECEIVE_BUFFER_SIZE];

// Initialise the queue - called from the initialisation function.
receive_queue = QUEUE_Init(receive_buffer,
                           sizeof(receive_buffer),
                           BYTE_ORDER_NATIVE);

// Send the queue to the output function - called from the task.
UART_Receive_Output(&receive_queue);
```

Figure 5-12: The use of a queue as an output port for the UART component.

5.4 Drivers

Components are written to be device independent, so that they may work equally with different target platforms. Drivers are the hardware abstraction libraries that make this possible.

For example, the initialisation function for the LED output component makes use of the GPIO driver, as shown in Figure 5-13.

```
/**
 * Initialises the GPIO pin for output.
 */
void LED_Output_Init(void)
{
        GPIO_Init(LED_OUTPUT_PIN, NULL);
        GPIO_Set_Direction(LED_OUTPUT_PIN, GPIO_OUTPUT);

        pin_high = FALSE;
        GPIO_Clear(LED_OUTPUT_PIN);
}
```

Figure 5-13: The LED_Output component making use of the GPIO driver.

Drivers are located on the system include path, so the GPIO driver is included with the "`#include <tte/gpio.h>`" preprocessor directive.

When switching between toolchains and manufacturers, the system include directories are altered so that the correct drivers are always used for the current target. This can be seen in Figure 5-14.

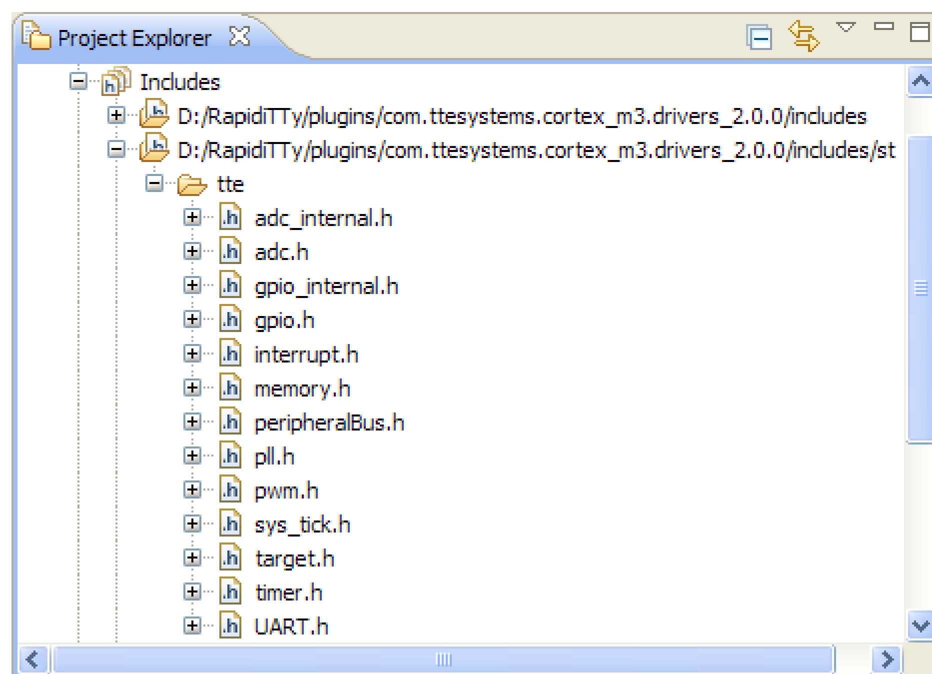

Figure 5-14: Driver include files for the STM32 targets.

As shown in Figure 5-14, the driver files can be found through the project explorer view, by looking under the "Includes" tree.

6 Debugging a project

Now that we have a simple project, we will need a way to test and debug it. This can be done from project properties, with the method shown in Section 3.3.2 on page 14.

If we choose to debug the project, RapidiTTy MCU will ask us if we want to switch to the debug perspective. This is a separate collection of views specifically aimed at debugging, so it is usually best to accept the switch.

> We can return to the default RapidiTTy perspective, or switch to any other available perspective, using the buttons in the top-right of the workbench (the main RapidiTTy MCU window).

6.1 Basic debugging

Once in the debug perspective, we are presented with a number of helpful views. The first, and perhaps the most important, is the debug view (shown in Figure 6-1).

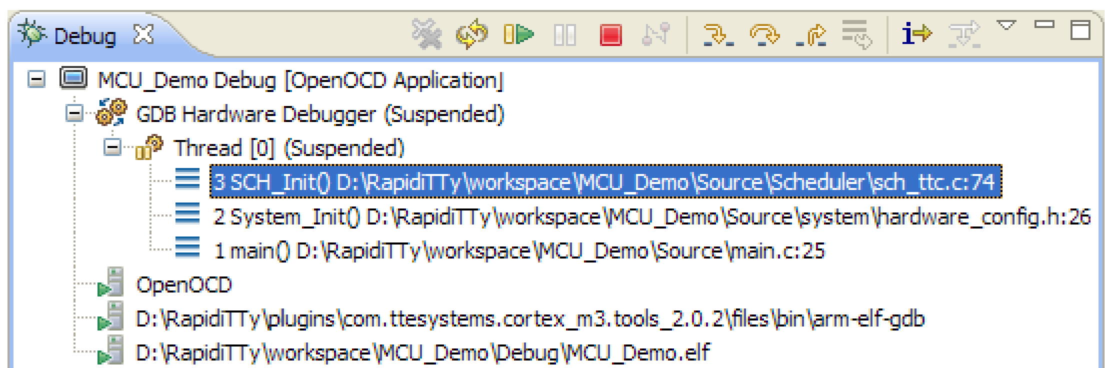

Figure 6-1: The debug view, showing the current stack trace.

There are two important things to note in Figure 6-1: the stack trace and the debugging toolbar. The top of the stack trace is highlighted; this shows the function that is currently being executed, along with the list of functions that were called to get here. In this example, the `main` function called `System_Init`, which in turn called the `SCH_Init` function.

> Clicking on another function in the stack trace will take us directly to it. All views in the debug perspective will update to show the details of any function we select.

The debug toolbar (at the top of Figure 6-1), contains a number of essential actions for debugging. Some of the most important actions are shown in Figure 6-2. These are: continue, pause, terminate the current session, disconnect from the target hardware, step into a function, step to the next line, and step out of a function, respectively.

Figure 6-2: The debug toolbar.

> The restart action, located next to continue on the toolbar, is not supported in RapidiTTy MCU. Using it will most likely disrupt the current session, and we would have to restart debugging manually from the project properties overview page.

6.1.1 Examining variables

As we use the step operations on the debug toolbar to move through our code, we will probably want to see more detail about what is going on. For this, we need to be able to examine the content of variables, which can be accomplished using the variables view shown in Figure 6-3.

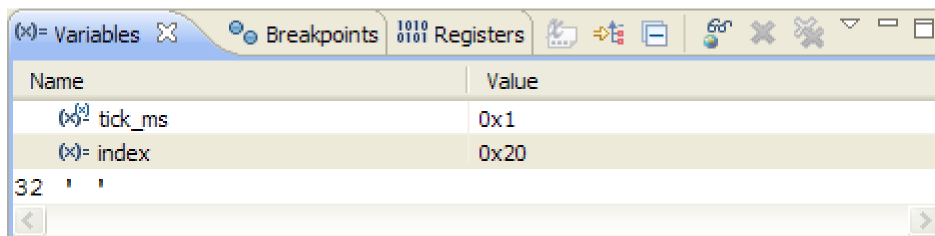

Figure 6-3: The variables view, showing different interpretations of a variable.

> By default, the variables view only shows entries that are local to the current function. This can be changed by clicking on the 'add global variables' button, in the centre of the toolbar. A dialog will then appear asking us to specify which global variables we want to add to the view.

6.1.2 Setting and examining breakpoints

If stepping is too slow, we can get to a specific line of code using a breakpoint. This is activated by double-clicking in the left margin in the editor view, next to the line we want to break at. If execution is resumed, using the continue button on the tool-

bar of the debug view, then the application will continue to execute until a breakpoint is reached.

The breakpoints view, with one active breakpoint, is shown in Figure 6-4.

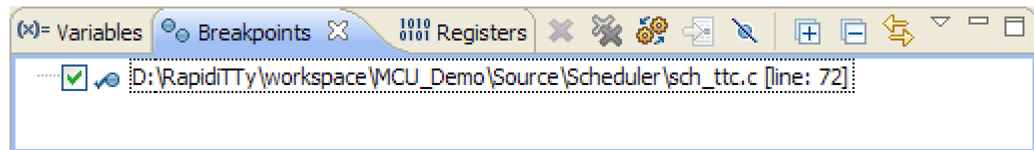

Figure 6-4: The breakpoints view, showing all active or inactive breakpoints.

Most embedded architectures impose a limit on the number of hardware breakpoints available; exceeding this limit will result in an error being reported and the breakpoint will not work. Sometimes this limit is as low as two!

6.1.3 Examining registers

Sometimes we need to take a closer look at what is going on in the hardware. In much the same way as examining variables, we can examine the contents of registers using the registers view. This is shown in Figure 6-5.

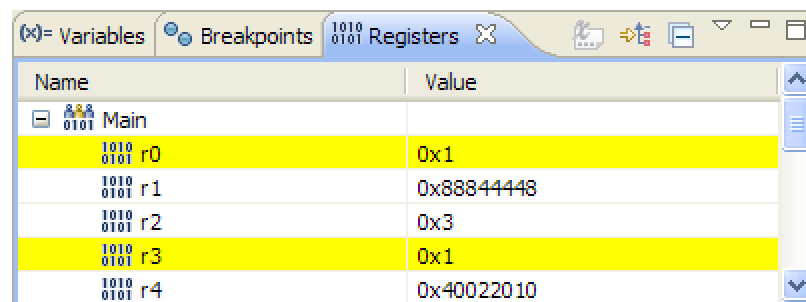

Figure 6-5: The registers view, with two showing changed values.

In Figure 6-5, note that some entries are highlighted in yellow, indicating that they have changed since the last step operation or refresh. This is the same as the variables view.

6.2 Handling assertions

Both the drivers and components make use of *assertions*. Debug assertions are only included in debug builds, whereas release assertions are present in both debug and release. Static assertions are a special case that can trigger errors at compile-time without actually influencing the generated code.

> Release assertions are used to detect errors that may well occur at runtime, or may potentially be missed during testing. Debug assertions are only used for potential problems that should always be found during testing. Static assertions are used in the same way as debug assertions.

RapidiTTy MCU handles a debug assertion by entering an infinite loop. If the debugger is running and everything seems to have stopped, we can pause the execution and see if a debug assertion has been raised. The stack trace in the debug view can be used to discover exactly which assertion was raised, as shown in Figure 6-6.

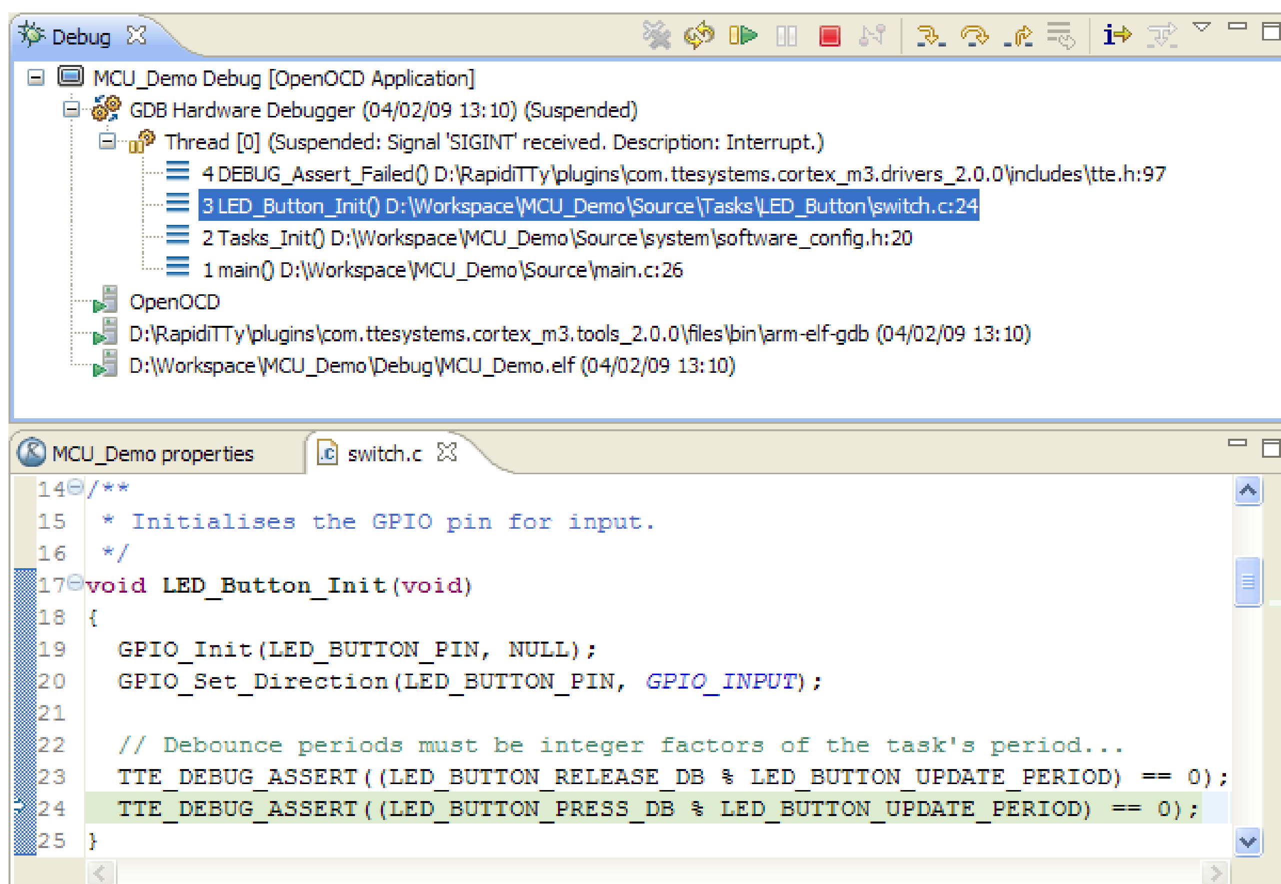

Figure 6-6: Using the stack trace to locate a debug assertion.

Note that the expression used in the assertion from Figure 6-6 is based on constant values that are known at compile-time. For this reason, it has since been replaced with a static assertion.

There are few debug assertions used in the RapidiTTy MCU components and drivers. This is because they must be found during testing. As such, they are only used where there is no alternative, or where a test may be too costly to include in release code.

6.3 Targets without debugging support

RapidiTTy MCU includes preliminary support for some targets that cannot yet be debugged using the existing JTAG debugging framework. These targets cannot be used with the debugger, but compiled code may be uploaded to the device over a standard RS-232 serial connection using a third party tool: Flash Magic[4].

Once the Flash Magic tool has started, we can simply fill out the information in each step – specifying the device, oscillator frequency, serial port, erase options and hex file to be used. We recommend that the erase options in step two are set to the "Erase all Flash+Code Rd Prot" setting.

The hex file for your project can be found in the project's folder (in the workspace), under a subdirectory named after the current build target (usually "Debug" or "Release"). Using the example project discussed previously, the hex file would be: "`D:\Workspace\MCU_Demo\Debug\MCU_Demo.hex`".

In order for Flash Magic to connect to the development board, there must be a serial cable connecting the specified port on the computer to the correct ISP port on the device. There may also be specific jumper settings that are required to use the ISP functionality – please see the documentation for the development board for details.

4 Flash Magic can be downloaded from http://www.flashmagictool.com.

7 Acquiring timing data

RapidiTTy MCU ships with the TTE Statistics toolbox, which can acquire detailed timing information about a project without requiring expensive dedicated hardware. As discussed in Section 3.3, the statistics gathering capabilities can be accessed from the timing statistics section of project properties (shown in Figure 7-1).

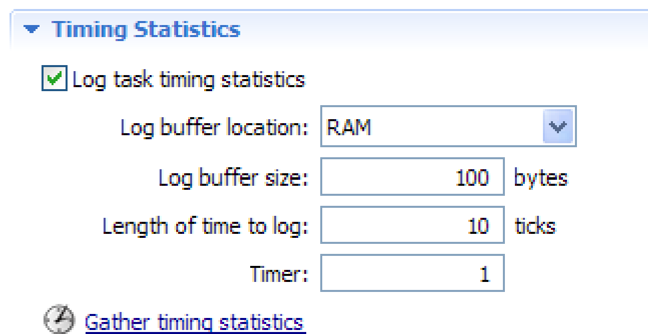

Figure 7-1: The timing statistics section of project properties.

The work of gathering timing data is carried out by the schedulers, which log the data to a buffer on the microcontroller. The location and size of this buffer is configurable through the options shown in Figure 7-1. When the buffer is full RapidiTTy MCU will read in the data, after which the scheduler can start filling the buffer again. For this reason, the size of the buffer only affects the time taken to gather data and not necessarily the data itself.

> While data is transferred, the microcontroller is stopped. If used in a real-time environment, this may cause problems. In this case, we can either use a test-bed capable of being paused, or a buffer of sufficiently large size to store all the required data.

The remaining options shown in Figure 7-1 are used to configure the length of time we will acquire data for (in scheduler ticks), and the number of the timer to use for the logging.

> The timer used for logging data will be configured as free-running and should not be used or altered for any other purpose. In particular, you must ensure that the scheduler and data logger are using different timers – most of the schedulers will use timer zero by default.

7.1 Creating a demonstration project

In order to demonstrate timing analysis, we will be using a new project based on the FPH (Fixed Priority Hybrid) scheduler discussed in Section 4.3 on page 18. The project will have two components (called "Long" and "Short"), each with one task (called "Task"). This will provide us with a system containing two tasks, as shown in Figure 7-2, with full names "Long_Task" and "Short_Task".

Task Properties

Name (decreasing priority)	Delay (ticks)	Period (ticks)	Pre-emptive
Short_Task	0	1	false
Long_Task	0	2	false

Figure 7-2: Task properties for the timing demonstration.

After creating the project and setting up the tasks as shown in Figure 7-2, we could simply run timing analysis and see the results. However, the tasks do not currently do anything and so would be barely visible in the results. To fix this, we can add empty loops to each task, to create a crude delay and simulate some real processing time. Figure 7-3 shows an example of this for the `Long_Task` function.

```
void Long_Task(void)
{
	for (uint32_t i = 0; i < 800; i++)
	{
		// Deliberately empty!
	}
}
```

Figure 7-3: The `Long_Task` function, with a crude delay.

The same delay shown in Figure 7-3 can be used for the `Short_Task` function, but the delay should be shorter – a loop counting up to `100` should be short enough for this demonstration. With those tasks set up, we are ready to start timing analysis.

7.2 Acquiring timing statistics

With a project created and setup, we can now simply click on the "Gather timing statistics" link shown in Figure 7-1 to start the data logging. RapidiTTy MCU will then compile the source-code, upload the executable and begin acquiring data. Each time a full buffer of data is transferred, the upload dialog will be updated with the current progress, as shown in Figure 7-4.

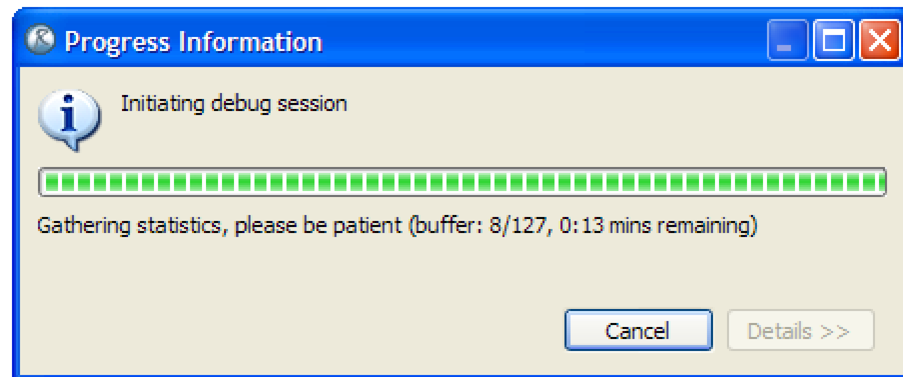

Figure 7-4: Gathering timing data, with an estimated 13 seconds remaining.

The session shown in Figure 7-4 is gathering one second of timing data (`1000` ticks with a period of `1000` µs per tick) with a buffer size of `100` bytes. This is shown as resulting in approximately `127` buffers of data (or `12700` bytes) – clearly the small buffer is delaying the data acquisition.

The NXP LPC-2378 device that we are using has an area of RAM that is reserved for use by the Ethernet peripheral. As we are not using this peripheral in our system, we can happily use the entire `16384` byte region for logging the timing data, simply by setting the appropriate options in the timing statistics section (shown in Figure 7-1).

> Logging timing data is not instantaneous and the overhead of logging can sometimes interfere with the results. Consequently, we recommend either gathering data in the release mode of the application, or turning on optimisation just for the scheduler's source-file (right-click on the file, select "properties" and change the optimisation setting).

> Because statistics are gathered from the system itself, if there is a major flaw in the application (stack overflow, constant task overruns, etc.) then the acquisition process may not complete. See the troubleshooting advice in Section 11 for more information.

7.3 Viewing timing statistics

Once the statistics have been acquired, the statistics editor will open automatically. This editor has three tabs showing an overview of the task timing for the whole system, graphical representations of the data for each individual task and the actual data itself.

7.3.1 An overview of the entire system

The first thing we notice on the overview tab is the task timing section. This shows all the tasks executing on the microcontroller in a form of simplified Gantt chart, as shown in Figure 7-5.

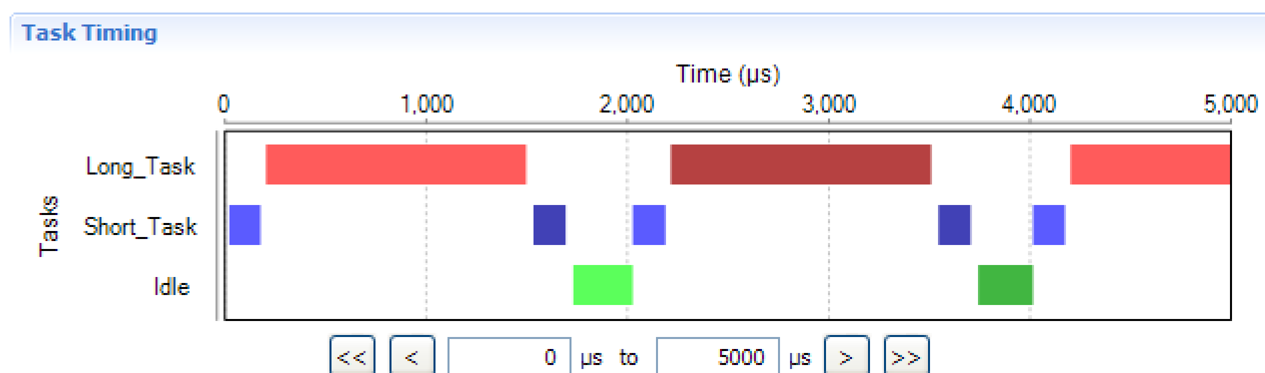

Figure 7-5: The task timing section, showing every task in the system.

The buttons and text-fields at the bottom of Figure 7-5 control the range that is displayed in the chart. The single left and right arrows ("<" and ">") move the chart range by one tick, whereas the double left and right arrows ("<<" and ">>") move the chart by the current range (i.e. they move by one page). The range can be set manually by altering the two text-fields directly.

> Note that the colours are alternating light and dark as we look along the x-axis; this allows us to distinguish between subsequent runs of a task (as opposed to a task that is resumed after being pre-empted). Also note that idle time is represented as a separate task.

> In some architectures, the JTAG debugging connection is lost when the scheduler enters idle mode. For this reason, it is impossible for us to gather timing data for such devices unless they remain active all the time (which is the default in RapidiTTy MCU for these architectures).

In addition to the task timing section, the overview tab also shows details of the total processor utilisation, as shown in Figure 7-6.

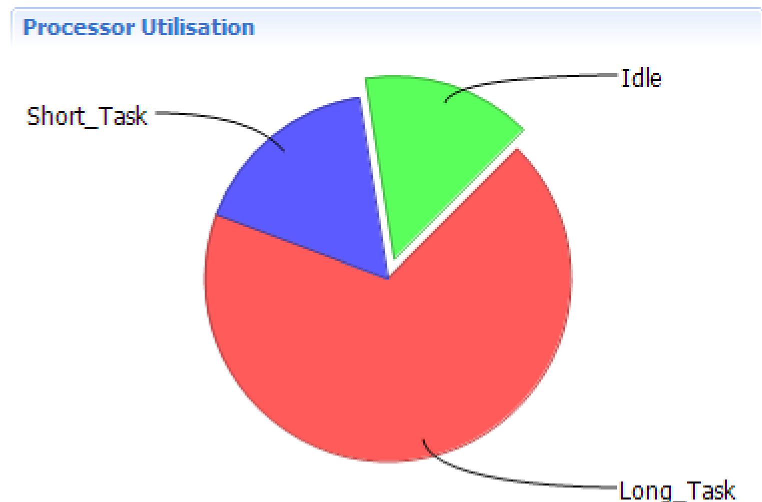

Figure 7-6: The processor utilisation section.

The processor utilisation chart in Figure 7-6 simply shows the proportion of processor time used by each task (including idle time). This helps us to see at a glance which task may be hogging the microcontroller.

7.3.2 Details for individual tasks

The timing details tab and raw data tab both show the same information, but the former displays its data graphically (as shown in Figure 7-7).

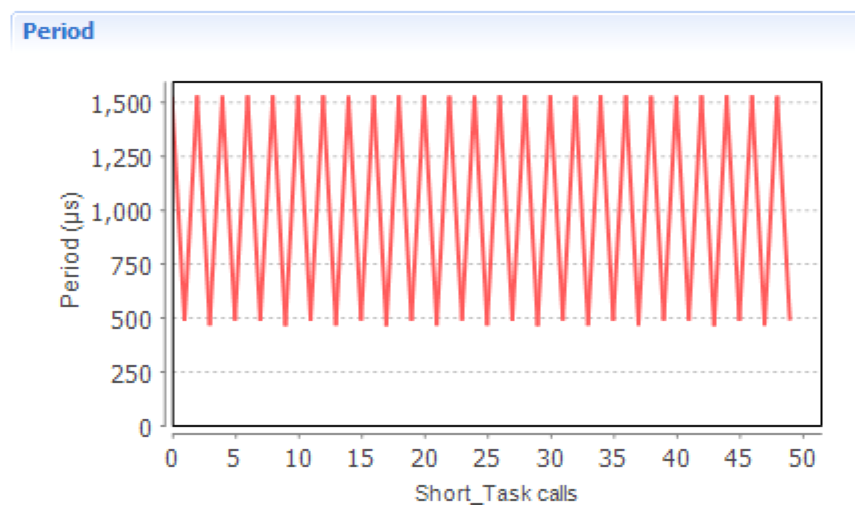

Figure 7-7: The measured period of Short_Task.

In addition to the period chart shown in Figure 7-7, the timing details tab also shows a graph of the tasks execution times, which makes it easier to see any potentially problematic variation. There is also a section dedicated to summarising the

statistics behind these two charts – including minimum and maximum values, averages, etc.

> The graphs and statistics shown on the timing details and raw data tabs change when you select different tasks, so be sure to select the task you are interested in!

7.4 Making changes

Of particular note in Figure 7-7 is that, although `Short_Task` is executed with the correct average period of `1000` μs, the actual period varies considerably. Combine this with Figure 7-5 and we can clearly see the cause – `Long_Task` is overrunning the scheduler's tick interval.

Assuming that this behaviour is undesirable for the system we are creating, we can fix it by returning to the software tab of project properties and setting `Short_Task` up to be pre-emptive (in the task properties section). Simply by saving, rebuilding and gathering more timing statistics, we get the result shown in Figure 7-8.

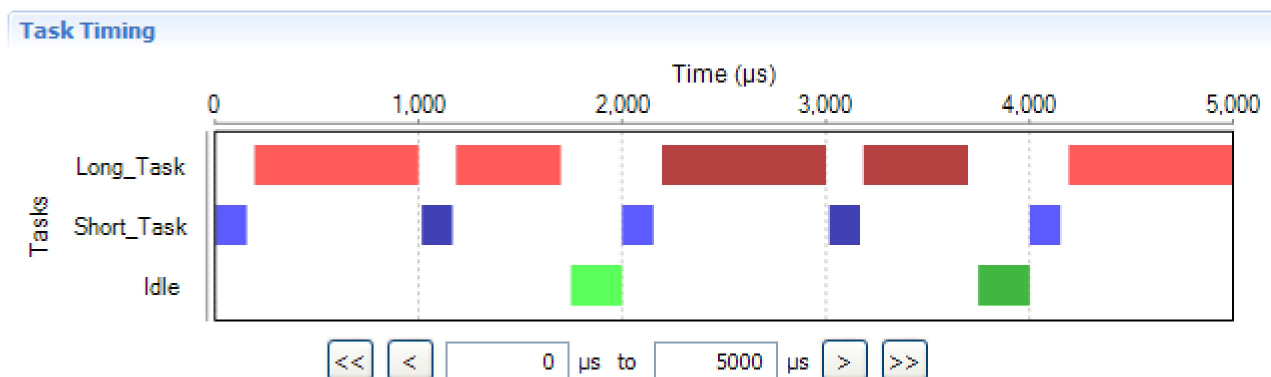

Figure 7-8: Task timing for a hybrid scheduler with a pre-emptive task.

In Figure 7-8 we can clearly see that `Short_Task` is pre-empting `Long_Task` at the tick interval. As mentioned previously, the alternating light and dark colours show separate runs of a task, whereas two blocks of the same colour indicate that the task was pre-empted and then resumed.

As before, we can look at the timing details tab to see information about specific tasks. In this case, the chart showing the period for the pre-emptive version of `Short_Task` can be seen in Figure 7-9.

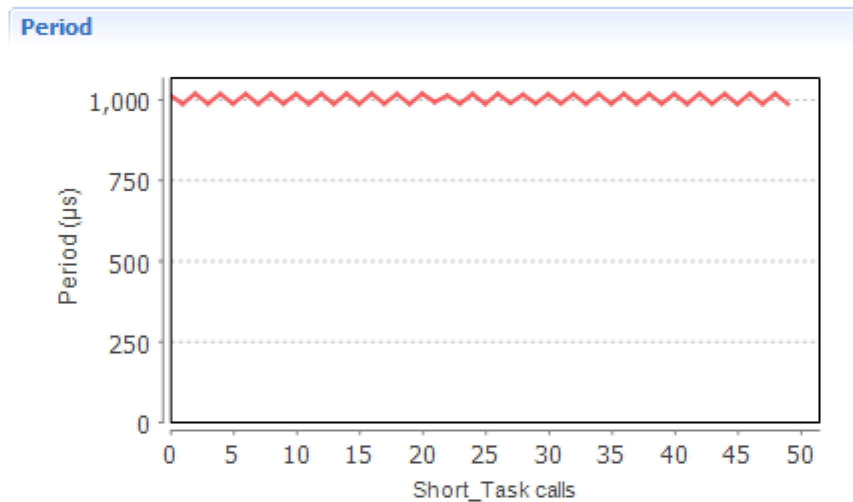

Figure 7-9: The measured period of the pre-emptive version of Short_Task.

Notice in Figure 7-9 that the measured period is now *much* closer to the value that we specified back in Section 7.1 on page 40. This clearly demonstrates the benefits of timing analysis – we no longer need to guess what might be going on, or invest in expensive equipment that may not provide the full picture.

8 Observing memory usage with analytics

RapidiTTy MCU supports a variety of different target microcontrollers, covering a wide range of capabilities in terms of the included peripherals and memory. We have already seen how drivers and components work together to handle differing peripherals between targets, but what about memory usage? How do we know if the program will fit in flash or RAM? How can we tell if there is enough stack space available?

To answer these questions, RapidiTTy MCU provides *analytics*, which are gathered through a static analysis of the compiled executable, assembly listing and memory map.

8.1 Memory meters

The most obvious display of the gathered analytics information can be seen in the memory meters, which are always present on the status bar (at the bottom-right of the screen) and are shown in Figure 8-1.

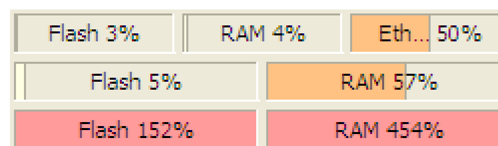

Figure 8-1: Code meters for the same program on three different targets.

The memory meters show the current memory usage of the compiled application for the currently selected target. Figure 8-1 shows combined screenshots for the same application, but each time with a different target – at the top is the LPC2378 that we used previously, but with half of its Ethernet RAM dedicated to timing analysis. The middle meters are for an LPC2129 and show that, without Ethernet RAM available, the log must be placed in ordinary RAM. Finally, the lower meters show that the application is too large to fit into the LPC2101 at all.

> When comparing targets that are in the same family or use the same toolchain, a recompile is not always necessary – RapidiTTy MCU will automatically update the meters to match the memory layout of the current device (using analytics acquired from the previous build).

This information is useful in assessing different architectures, but it doesn't help us to find out where the memory is going and how we can get some of it back. For that, RapidiTTy MCU provides the analytics page.

8.2 Detailed analytics reporting

The analytics page can be found by clicking on the “Analytics” tab at the bottom of project properties. The page is shown in Figure 8-2.

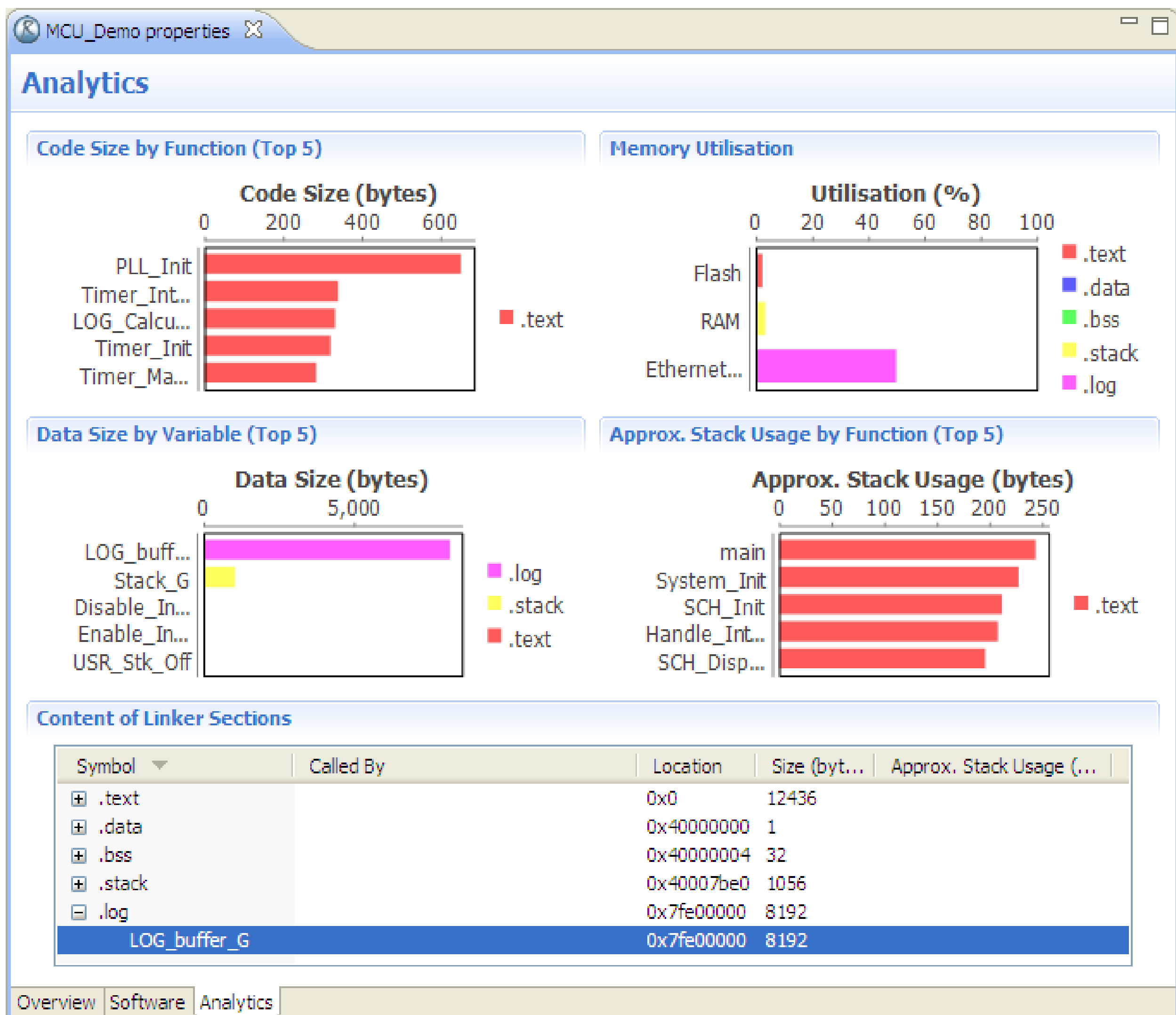

Figure 8-2: The analytics page of project properties.

At the top-right of Figure 8-2, we can see the memory utilisation graph. This shows the same information as was visible in the memory meters, but contains additional details showing which top-level sections are present in each memory region, and their respective sizes.

Figure 8-3 shows a closer view of the graph, where we can see at a glance that the log buffer is located in Ethernet RAM and that the `.data` and `.bss` sections are not significant components of the overall memory map.

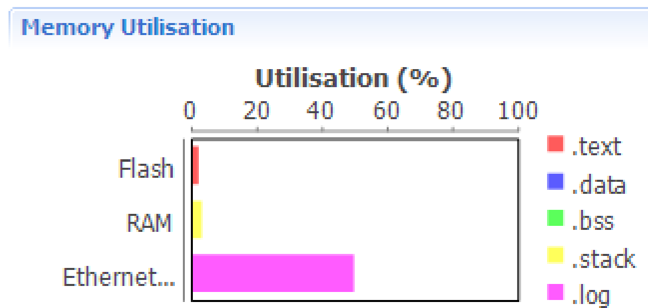

Figure 8-3: A closer view of the memory utilisation graph.

Figure 8-4 shows a closer view of the code and data size graphs, which list the top five largest functions and variables, respectively. From the latter we can see that the log and stack effectively dominate the application's RAM usage.

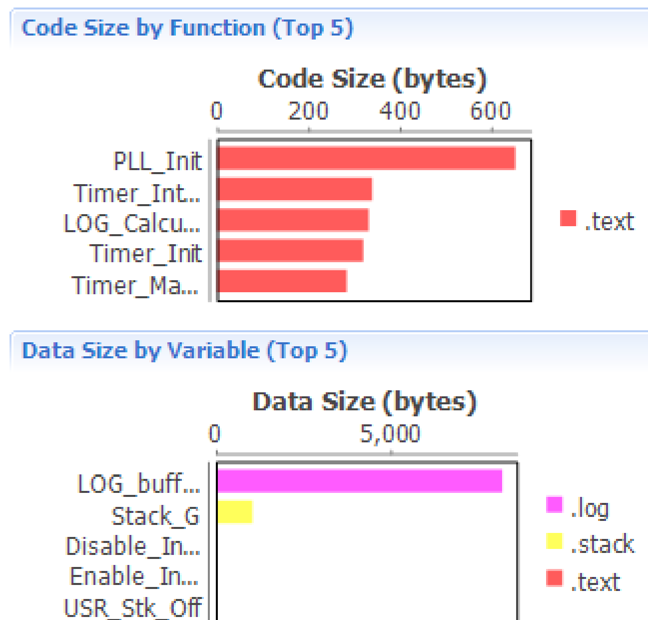

Figure 8-4: A closer view of the code and data size graphs.

The stack usage graph, shown in greater detail in Figure 8-5, shows the stack space that the function may use, including space used by any functions that it calls.

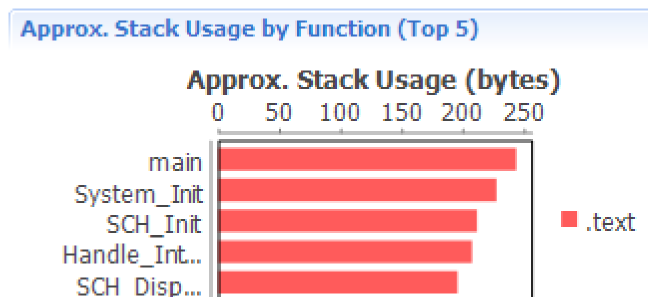

Figure 8-5: A closer view of the stack usage graph.

Figure 8-6 shows a more detailed view of the final section in the analytics page, which details the contents of the various linker sections.

Content of Linker Sections

Symbol	Called By	Location	Size (bytes)	Approx. Stack Usage (bytes)
SCH_Init	System_Init	0x1330	244	212
SCH_Start	main	0x2020	48	180
SCH_Update	Handle_Interrupts	0x2750	84	192
Select_System_Interrupt	SCH_Start	0x26b0	68	20
Sys_Tick_Int_Flag_Reset	SCH_Start, SCH_Update	0x2588	28	68
System_Init	main	0x1034	48	228
System_Scheduler_Sleep	SCH_Go_To_Sleep	0x2f73	1	20
System_Scheduler_Start	SCH_Start, LOG_Check_Buffer	0x2738	24	60
System_Scheduler_Stop	LOG_Check_Buffer	0x2508	24	60
System_Tick_Init	SCH_Init	0x1424	68	188
Target_Mapping_Init	System_Init	0x1158	36	16
Tasks_Init	main	0x117c	16	16
Timer_Clear	LOG_Init	0x2144	100	44
Timer_Frequency	LOG_Init, Timer_Init	0x19b0	116	132

Figure 8-6: A closer view of the linker sections table.

As we can see in Figure 8-6, RapidiTTy MCU provides full details of the analytics data for every function, including a list of which other functions call it. From this table and Figure 8-5, we can clearly see that no function uses more than 256 bytes of stack space, so that is the minimum we should use for our user stack.

The linker sections table can be sorted by name or by any of the numerical columns (i.e. any column except for "Called By"). This allows us to see at a glance which functions or variables are taking too much space.

If RapidiTTy MCU detects that too little stack space is available for a given function or task, a warning will be shown on the overview page of project properties.

9 Where do we go from here?

Now that we have completed our first project, where can we go from here?

9.1 RapidiTTy MCU

For RapidiTTy MCU support, or to ask general questions about embedded systems, you can visit our online discussion forum at http://www.tte-systems.com/forum. There is also an additional tutorial demonstrating the development of a data acquisition system – it can be found at http://www.tte-systems.com/downloads.

9.2 Other products in the RapidiTTy family

RapidiTTy MCU is just one of the products in the RapidiTTy family. The other products are described in this Section.

9.2.1 RapidiTTy FPGA

Where RapidiTTy MCU is designed to work with systems based on commercial-off-the-shelf (CotS) microcontrollers, RapidiTTy FPGA is designed to work with "soft" processor cores running on a Field Programmable Gate Array (FPGA).

RapidiTTy FPGA offers many of the features in RapidiTTy MCU, in addition to a number that are unique. The complete feature list includes:

- Timing analysis for time-triggered systems.
- Full source-code for the Time-Triggered Co-operative Operating System (TTCos).
- Full source-code for the TT3 soft processor core, which consists of:
    - A 32-bit processor core compatible with the MIPS I™ Instruction Set Architecture and capable of running at up to 50 MHz.
    - Predictable timing – one cycle for each of the five pipeline stages.
    - Timer, UART, PWM, GPIO, watchdog and hardware debugging peripherals.
    - Expansion through a simple peripheral bus.

9.2.2 RapidiTTy x86

RapidiTTy x86 provides the functionality of RapidiTTy MCU, but for higher-end hardware based on the Intel® x86 architecture (including Intel® Atom).

RapidiTTy x86 can be used:

- As a platform for rapid prototyping (designs and code can be moved easily to other members of the RapidiTTy family.
- For deployment of low-volume, high-end applications (taking advantage of the 4 Gb+ address space and high CPU performance of embedded PC hardware).

RapidiTTy x86 applications can be booted directly from hard disk, CD-ROM and even USB memory stick. You can also use the integrated simulator to develop and test on the same machine.

10 Migrating and importing projects

This section will discuss two forms of project migration – importing projects that may have come from an older version of RapidiTTy MCU, and importing projects from the old RapidiTTy Builder development environment.

10.1 Importing an existing project

Migrating projects from an older version of RapidiTTy MCU is a simple matter. In most cases, the migration will take place automatically when the new version of RapidiTTy MCU is first opened. However, if we wanted to import a project from another workspace or developer, then the procedure is slightly more involved.

This procedure is only valid for existing RapidiTTy MCU projects!

First open the import wizard (shown in Figure 10-1) by either selecting "Import" from the file menu, or by right clicking in the project explorer view and selecting "Import" from the context menu.

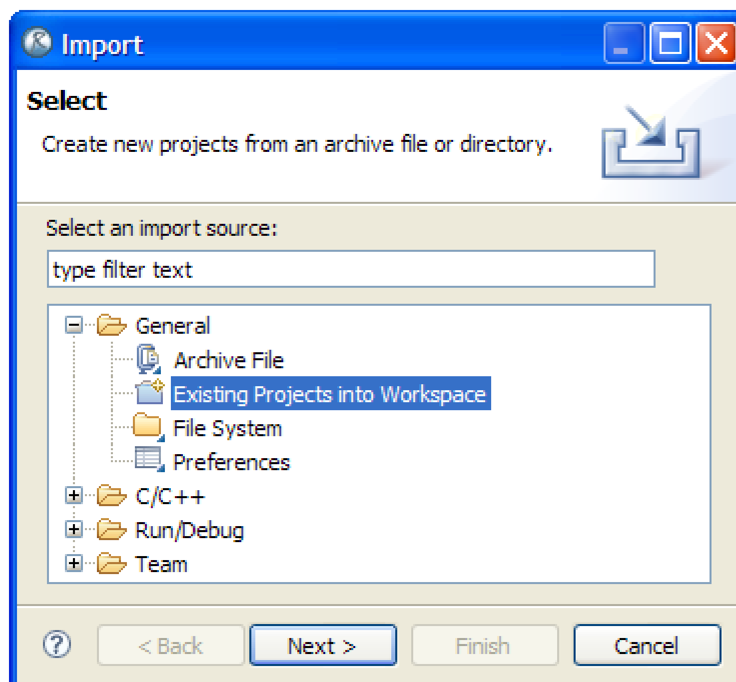

Figure 10-1: The first step in the import wizard.

As shown in Figure 10-1, we must choose the “Existing Projects into Workspace” option on the first wizard page, then press next. The second wizard page is shown in Figure 10-2.

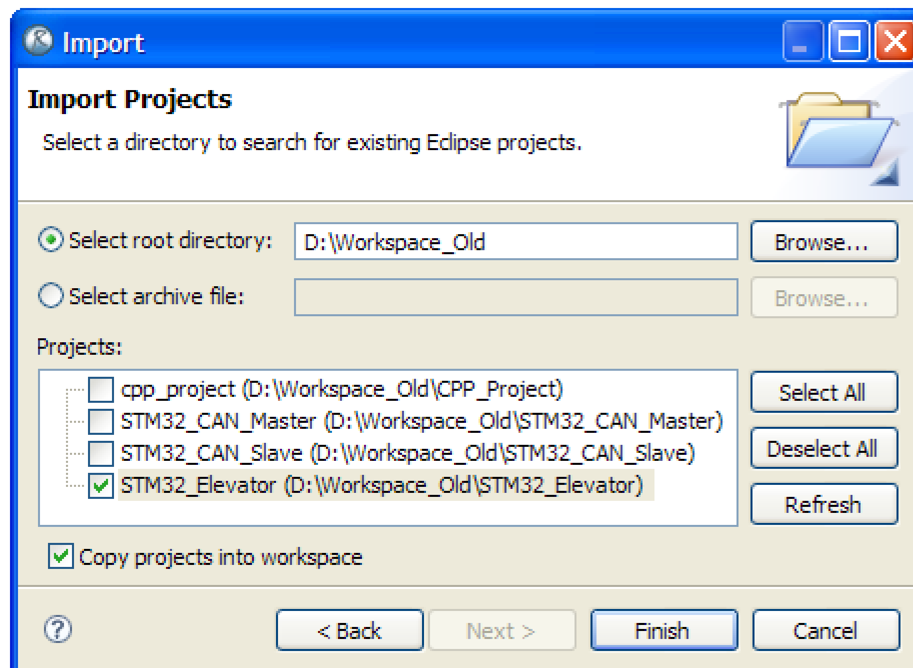

Figure 10-2: Choosing the projects to import.

Figure 10-2 shows the final wizard page, where the project to import is selected. First we must choose the root directory, which will be the workspace folder if we have one or the parent folder of the project if not. The projects contained in the root folder will then be shown in the list and we can select the projects to import then simply press “Finish” to import them.

When this process is complete the projects should be available and updated to match the current versions. However, new versions of RapidiTTy MCU may include updated components and schedulers, which will not be updated. To fix this, we could generate a new project with the desired code and simply copy it across to the imported project.

10.2 Migrating from RapidiTTy Builder

RapidiTTy Builder is RapidiTTy MCU's predecessor – it used a very different approach to projects and components, which makes it difficult to import them directly. The easiest approach is to create a new empty project (without a scheduler) and copy the source-files directly from the old project.

> Source files can be copied by dragging them from Windows Explorer and dropping them into the source folder in the project explorer view.

There are some potential problems that may be encountered with this process – these include the new build system options, generated scripts (for the linker and startup) and the relative paths used in `#include` statements.

10.2.1 The new build system

RapidiTTy Builder employed Unix makefiles to build projects. This is a fairly standard approach, but has many problems. For example, numerous processes are spawned to perform compilation, which can make it hard to cancel a long-running build. Also, preprocessor definitions and other switches selected on the command line (inside the makefile) greatly affect the meaning of the source-code but were unknown to the IDE, which causes code indexing (used for syntax highlighting, as well as code folding and completion) to fail.

To solve these issues, RapidiTTy MCU employs an internal builder that launches the compiler and linker directly. As a consequence of this, compiler and linker options are no longer maintained inside a makefile, but instead stored in the project settings. The options contained in an existing makefile can be placed directly in the settings, as discussed in Section 3.3.1 on page 12.

10.2.2 Generated startup and linker scripts

If the old project contained startup and linker scripts, these too can be copied to the new project, but not directly. For convenience, RapidiTTy MCU generates startup and linker scripts for us based on the settings provided in project properties. This generation can be turned off by clearing the checkboxes in the managed build settings (in the lower right of project properties), shown in Figure 10-3.

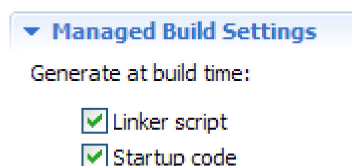

Figure 10-3: The managed build settings section, from project properties.

With the settings shown in Figure 10-3 deactivated, we can replace "`startup.strt`" (under the "System" folder in the project's source) with our old startup scripts – but they must be either assembly files (ending in "`.s`") or C files (ending in "`.c`").

The linker script ("`linker_script.ld`" found under the "Scripts" folder) should be left as it is (the name and location are required by the build system), but the content can be replaced with the version from our old project.

There are some default settings in the new build options (the "Function-specific sections" and "Data specific sections options" shown in Figure 10-4) that are incompatible with the default linker script that was used in RapidiTTy Builder. Please ensure that these options are deselected for any projects that do not use the new linker scripts generated by RapidiTTy MCU.

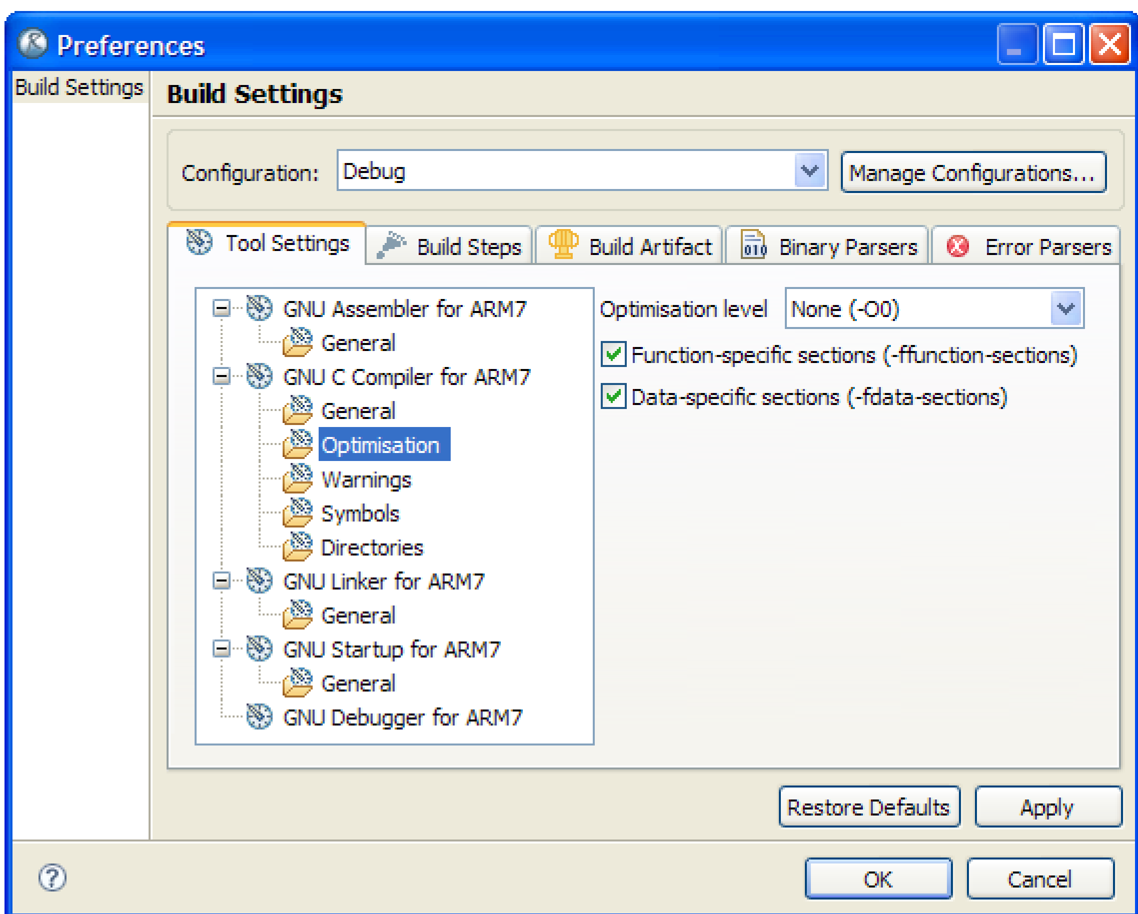

Figure 10-4: The build settings dialog, showing the two new options.

10.2.3 Relative paths in preprocessor includes

The makefiles used in older projects added all source directories to the local include file lookup path. This is no longer the case in RapidiTTy MCU, so the preprocessor will be unable to find the included header files. There are two easy solutions – either we alter each source file to add relative paths to each `#include` statement, or we could place all the source and header files in the same directory.

11 Troubleshooting

This Section discusses some basic techniques that may help troubleshoot some of the problems that we might face. If a problem is encountered that is not in this Section, please feel free to use our support forums[5].

11.1 Problems uploading and debugging

As is often the case when working with JTAG devices, there may be occasional problems with uploading and debugging. Here are a few tips that may provide a solution to the problem:

- For USB-based JTAG devices, try unplugging and reconnecting the device to reset it.
- Perform a power cycle on the development board, to fully reset it.
- Some devices (notably the ARM7 devices from NXP) can only have two breakpoints in flash memory, and one of these may be required for breaking at main on startup and single-stepping.
- In the debug perspective, check that all active launches have been terminated before starting new launches.
- Check in the Windows Task Manager for debugging processes that have not terminated correctly, including `<toolchain>-gdb.exe` and `openocd.exe`. Terminate them if they are present.

11.2 Common causes of runtime problems

Even after uploading the binary, the application still may not appear to run correctly. Here are some common causes of runtime problems:

- Tasks may be overrunning – if a task takes longer to execute than its own period, the system will not operate correctly (or not for long, in any case). With some schedulers, this may also cause a stack overflow. Try increasing task periods and the tick interval and run timing analysis to see what is actually going on.
- The stack may overflow, causing unpredictable behaviour. This is especially true of pre-emptive systems where each task must have its own stack – remember that all registers are saved on a context switch, which uses a significant amount of stack space for each task!

5 The support forum can be found at http://www.tte-systems.com/forum/support/mcu.

RapidiTTy MCU

- A debug assertion may have been raised. Try debugging instead of uploading, select "resume" and then pause execution. If an assertion has been raised, it will stop in the `DEBUG_Assert_Failed` function and the stack trace can be used to find the cause (see Section 6.2 on page 36).

11.3 Problems with timing analysis

If timing analysis fails to run, or runs but never completes, the following are possible causes and ways to fix them:

- As above, if a single task is overflowing its own period then timing analysis may fail. Try increasing periods and the tick interval, or make tasks run just once (set the period to zero) to find its execution time.

- A stack overflow will have serious and unpredictable affects on timing analysis, as the logging is carried out by the system itself. Try increasing stack sizes.

11.4 Finding potential stack overflows

Overflowing the stack is one of the most serious problems that can be encountered when creating a low-level embedded system. Pre-emptive schedulers make this especially difficult, as each task must have its own stack space – consider that (for a 32-bit architecture) a context switch will save four bytes for every register in the microcontroller. For ARM7 targets, this works out at `64` bytes of stack space just for the context.

RapidiTTy MCU includes basic runtime stack checking in every scheduler (i.e. every project type except for "Empty Project"), which will trigger a debug assertion (see Section 6.2 on page 36) if a specific value at the bottom of each stack is no longer present. As a very basic test, we recommend running the project in debug mode, selecting "resume", then simply hitting "pause" at any time to see if the system has stopped in a debug assertion.