Getting Started with

RapidiTTy x86 1.2

TTE Systems

Rapid development of reliable embedded systems

http://www.tte-systems.com

Version

Getting Started with RapidiTTy x86 v1.2 (November 2009)

Copyright

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

Trademarks

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.

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 Aim of this guide....6
2 Installation and hardware requirements....6
3 Working with projects....7
3.1 Workspaces....7
3.2 Project creation....7
3.3 Project-wide actions and settings....11
3.4 Renaming projects....12
4 Debugging a project....13
4.1 Using the simulator....13
4.2 Basic debugging....15
4.3 Creating a bootable USB flash drive....16
4.3.1 Formatting the USB flash drive....16
4.3.2 Installing SysLinux....17
4.3.3 Configuring SysLinux....17
4.4 Debugging on hardware....18
5 Timing analysis....18
6 Drivers....20
6.1 Keyboard driver....20
6.2 Serial port driver....21
6.3 VESA graphics driver....21
6.4 Hardware watchdog timer....22
6.5 Parallel port driver....22
6.6 Expander module drivers....23
6.6.1 SPI driver....23
6.6.2 GPIO driver....24
6.6.3 ADC driver....24
6.6.4 DAC driver....25
6.6.5 CAN driver....26
6.7 Hard disk drivers....26
6.7.1 ATA driver....26
6.7.2 FAT filesystem driver....27
7 Where do we go from here?....28
7.1 RapidiTTy x86....28
7.2 Other products in the RapidiTTy family....28
7.2.1 RapidiTTy MCU....28
7.2.2 RapidiTTy FPGA....28
7.2.3 RapidiTTy Professional....29
7.3 Time-triggered systems....29

1 Introduction

RapidiTTy x86 supports the development of reliable embedded systems which are based on “embedded” or “desktop” PC hardware.

RapidiTTy x86 can be used:

- For deployment of low-volume, high-end applications (taking advantage of the 4 GB+ address space and high CPU performance of embedded PC hardware).

- As a platform for rapid prototyping (designs and code can be moved easily to other members of the RapidiTTy family).

Key benefits of RapidiTTy x86 are as follows:

- A tightly integrated tool suite with the familiar RapidiTTy IDE: provides support for x86 targets (suitable for the “386” – and above – processors, including Intel® Atom).

- RapidiTTy x86 includes full source-code for a royalty-free version of the TTCos[1] and sEOS[2] operating systems which are suitable for use in single-processor embedded systems.

- Includes an extensive suite of high-quality library code covering: text-mode video; VESA; watchdog timer; time-stamp counter (TSC); keyboard; serial (RS-232) port; parallel port; PCI bus calls; hard-disk support; and FAT filesystem support. Example code is provided for all drivers.

- Includes a complete suite of drivers for the TTE Systems I/O Expander Module.[3] This provides direct support for: two 8-bit digital I/Os; two 12-bit analogue input channels; two 12-bit analogue output channels; two Controller Area Network (CAN) interfaces; two test switches; and eight test LEDs.

- Operates completely in 32-bit protected mode with access to the full 4 GB+ address space.

- Automatically generates a hard-drive image and configuration for simulation and debugging through the integrated QEMU simulator.

- Provides extensive debug support fully integrated with the IDE (including a GDB stub).

- Minimises the effort involved in precise timing analyses (including worst-case

1 More information about TTCos can be found online at http://www.tte-systems.com/books/pttes.
2 More information about sEOS can be found online at http://www.tte-systems.com/books/embedded-c.
3 More information about the TTE Systems I/O Expander Module can be found online at http://www.tte-systems.com/products/x86.

execution time measurements).

- Special support for developing time-triggered[4] (TT) technology. Use of TT technology helps to ensure that your applications have very predictable behaviour, and makes them easier to debug and test.

1.1 Aim of this guide

This guide will show you how to "get started" with RapidiTTy x86 by means of a tutorial. In this tutorial, we will be demonstrating (step-by-step) how you can use RapidiTTy x86 to develop a simple example system.

Before beginning the tutorial, RapidiTTy x86 must first be installed and configured to work with our hardware.

2 Installation and hardware requirements

RapidiTTy x86 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.

RapidiTTy x86 supports any Intel® x86 compatible[5] computer, including Intel® Atom. Programming and debugging is carried out with GDB over a serial connection, so both target and development environment require a free serial port if the application is to be run on hardware.

Unlike most modern microcontrollers, typical Intel® x86 computers do not ship with a boot loader for in-system programming and debugging. To compensate, we can produce a bootable USB flash drive with the executable that RapidiTTy x86 generates. Once the executable has been loaded, any subsequent programming and debugging operations may be carried out via the serial connection.

Obviously, in order for this to work the target computer must be configured to boot from a USB device. This can usually be accomplished through the BIOS settings – please refer to the user manual for the computer's motherboard for more information.

Finally, we could use any bootable media for this operation, but only the USB method will be discussed in this manual. Please see Section 4.3 for more details.

4 More information about time-triggered systems can be found online at the TTE Systems website: http://www.tte-systems.com/books/pttes.
5 The target must be compatible with the Intel® 386, or above.

3 Working with projects

3.1 Workspaces

We can start by opening RapidiTTy x86. The first time we do this, we are presented with a dialog asking us to choose a "workspace", as shown in Figure 3.1.

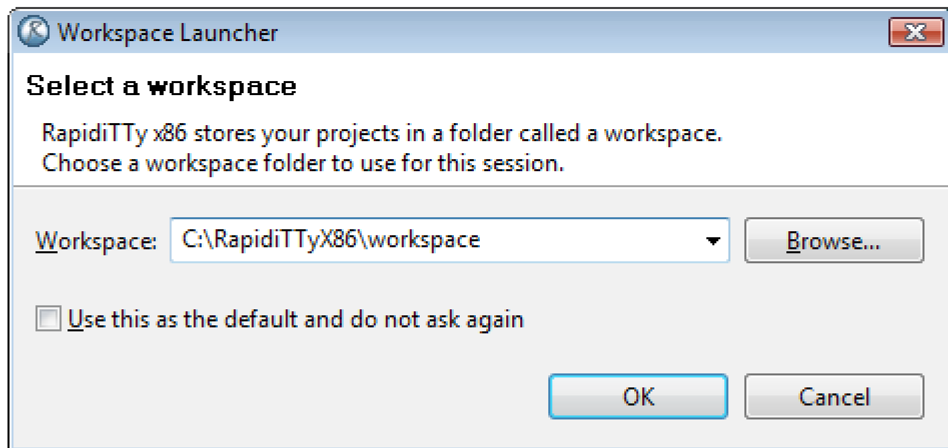

Figure 3.1: The workspace selection dialog.

The workspace is simply the directory under which all current projects are stored. For this tutorial, we will be using the default "`C:\RapidiTTyX86\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 x86 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.

In Figure 3.2, we have chosen to name the project "x86_Demo". This name will also be the default name of the compiled binary that is created, should it be required outside of RapidiTTy x86 (such as when we use it to create a bootable USB flash drive, later on).

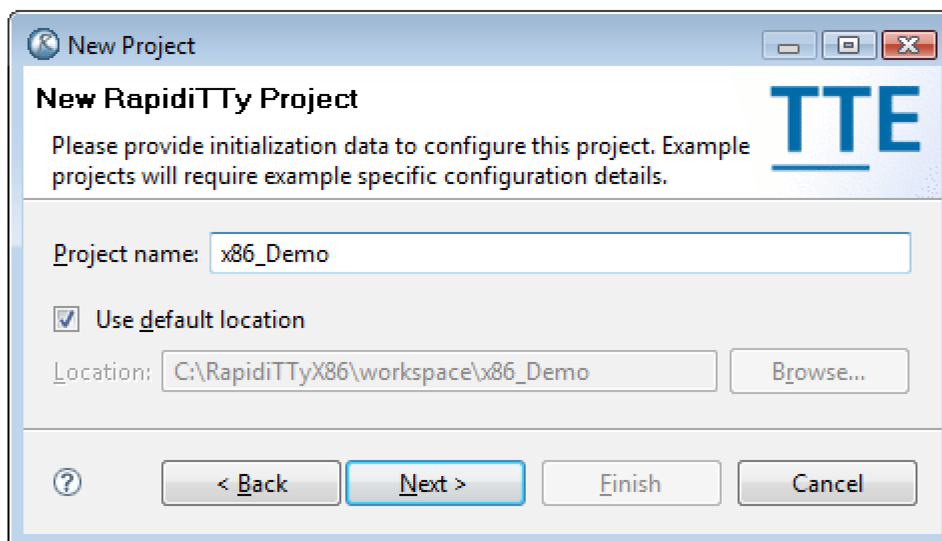

Figure 3.2: The new project dialog.

Once we have selected an appropriate name for the project, we must then select the desired processor configuration. For RapidiTTy x86 there is no actual choice here at present – simply select the “PC” option, as shown in Figure 3.3, and move on to the next page.

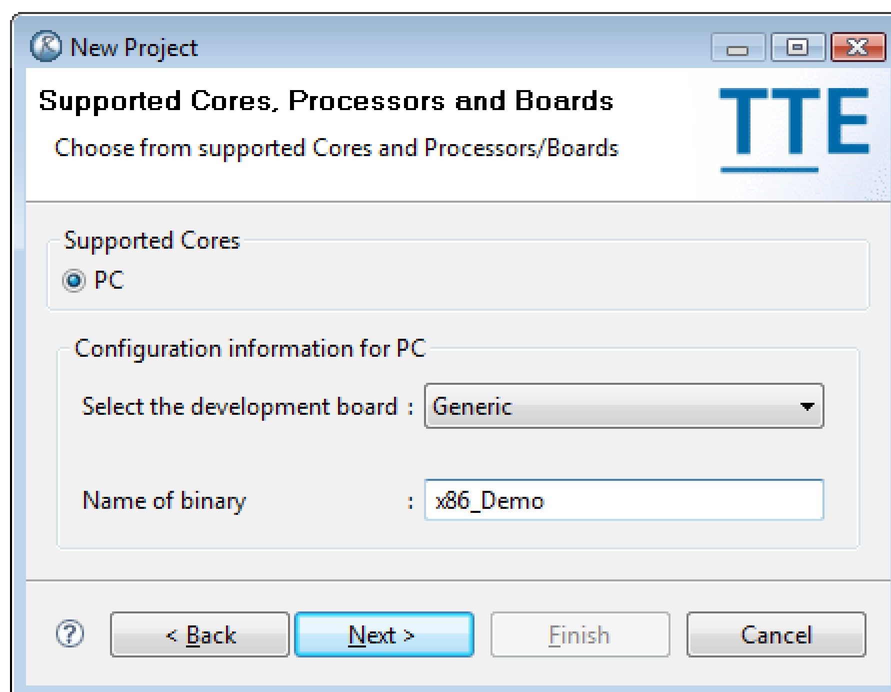

Figure 3.3: Selecting a processor core.

Next comes the driver selection, shown in Figure 3.4.

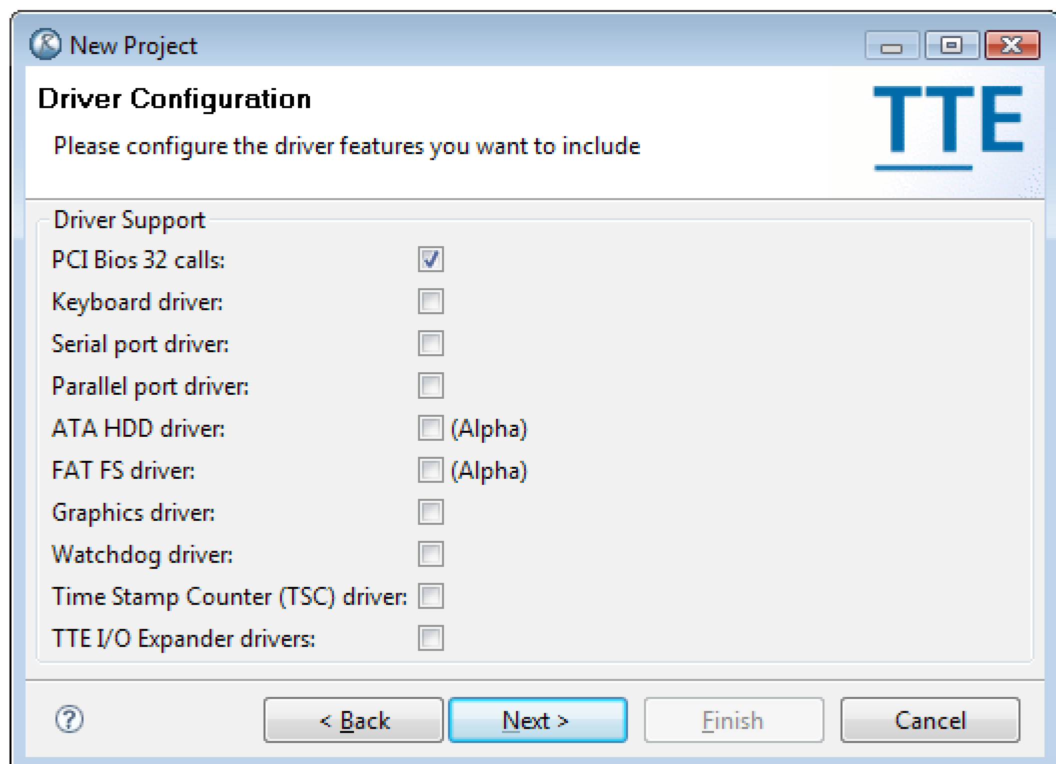

Figure 3.4: Choosing the drivers to add to the project.

Figure 3.4 shows the drivers that may be optionally added to the project. These are discussed in Section 6. Later, when we select an example project, it will automatically add any drivers that it requires, so for now we can leave this as it is.

> Drivers support is added when your project is created. If we wanted to use additional drivers after project creation, we can create a separate project with the drivers in and copy them across – the source-code for the specific driver is all that is required.

In order to program and debug the application, it must include a "GDB stub". This is the code that will interface with GDB over the serial link – RapidiTTy x86 will automatically generate the required code and add it to the project, but first it must be configured.

Figure 3.5 shows the configuration options for the stub. The settings refer to the target platform, so specifying COM port zero will make the stub use that port on the embedded computer (and not the host!).

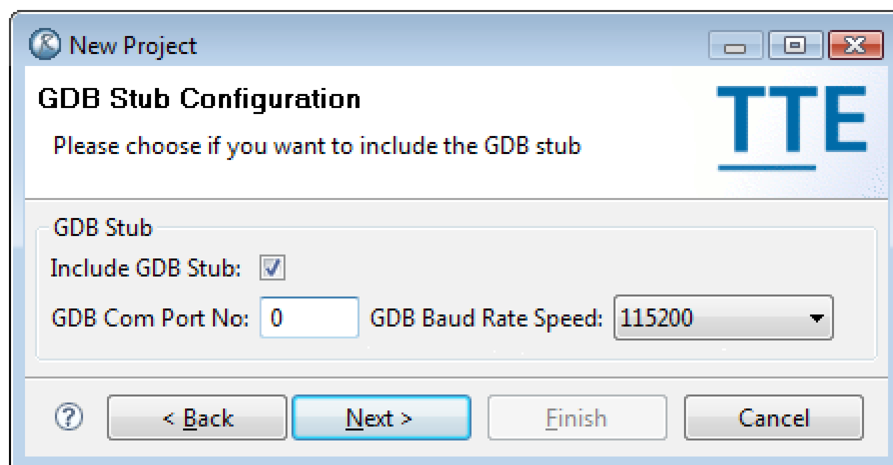

Figure 3.5: Configuring the GDB stub.

After project creation, you have full control of the GDB stub settings by editing the "`startup.s`" file.

Next, we can create a project from an existing example, as shown in Figure 3.6.

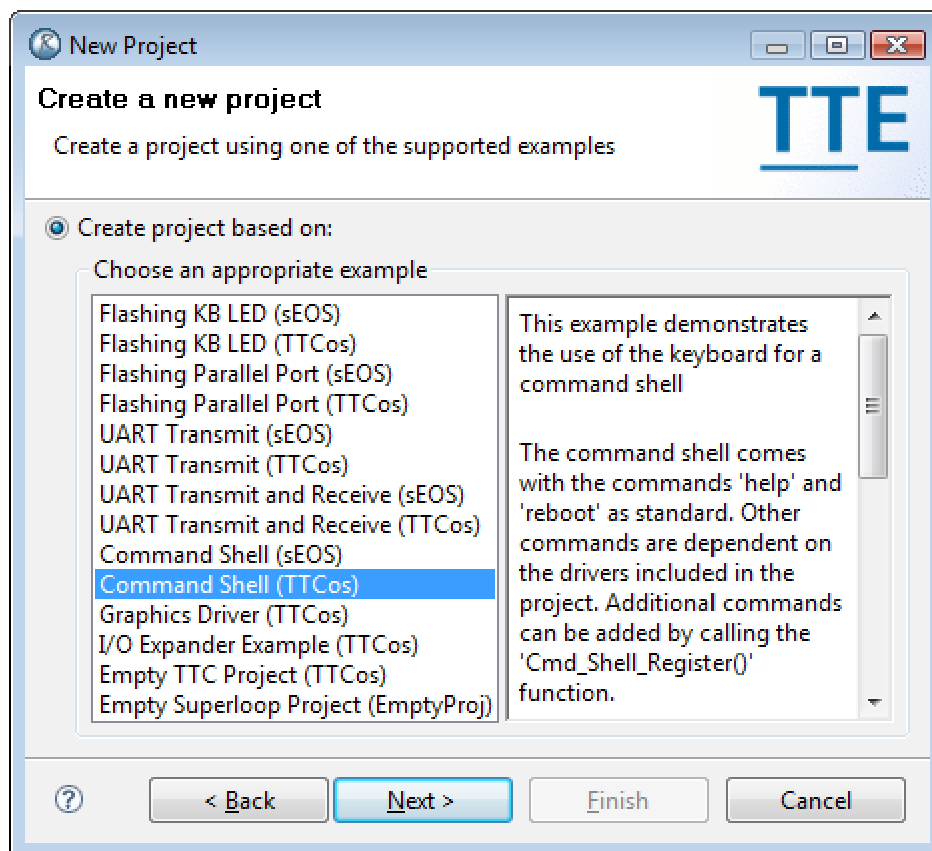

Figure 3.6: Choosing an example project.

Once an example project has been selected, we can alter configuration options for it, as shown in Figure 3.7.

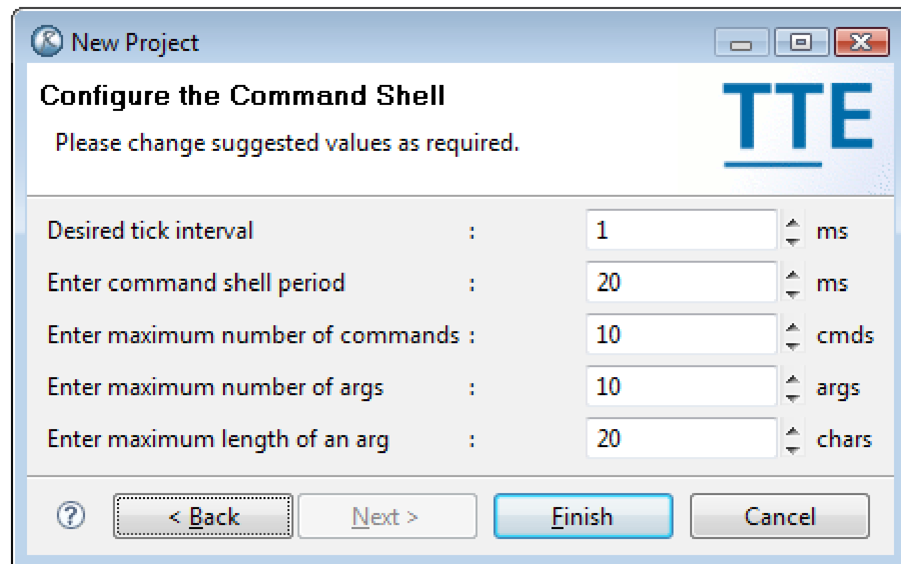

Figure 3.7: Configuring the command shell example.

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 example project (the same options shown in Figure 3.7), rebuild the project and start debugging or timing analysis.

> 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 3.8.

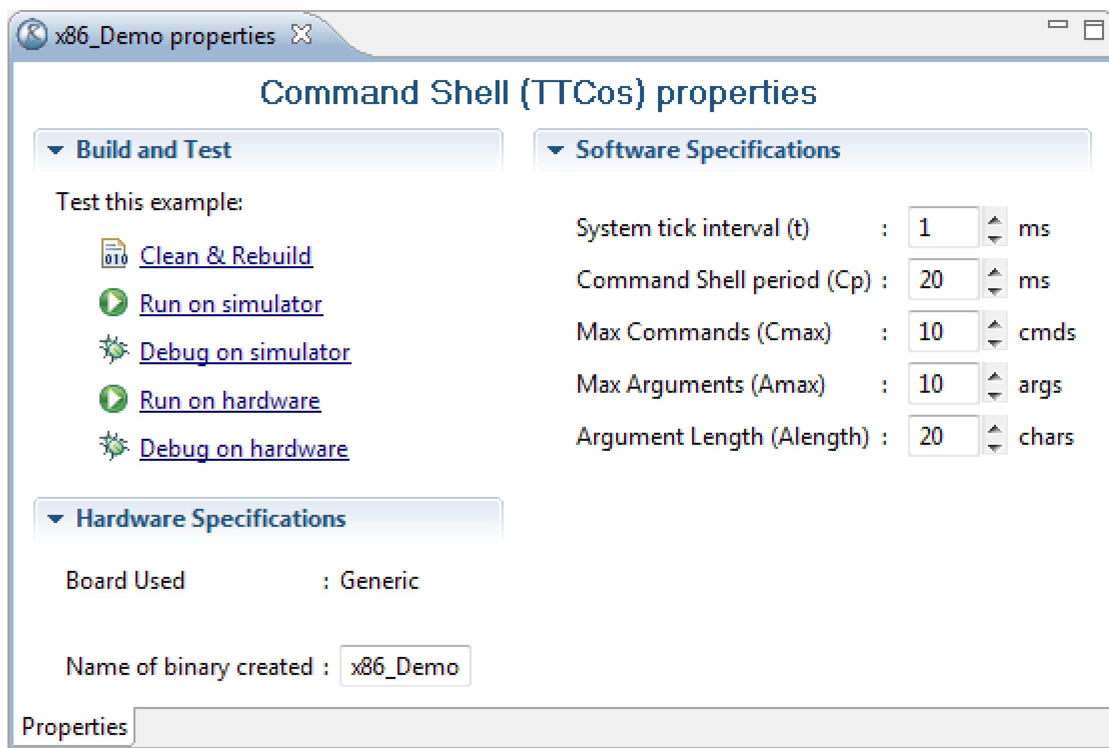

Figure 3.8: The initial project properties view.

At the right of the project properties in Figure 3.8 are the settings for the example project – the same settings that were originally set at project creation. To the left is the "Build and Test" section, where we can rebuild the project or start a debugging session.

> The final stage of rebuilding the project is linking. RapidiTTy x86 is slightly unusual in that the linking stage must also add the GDB stub to the final binary file. For this reason, linking can take longer than may be expected – but it must finish before we attempt to run or debug the final application.

3.4 Renaming projects

At the highest level, renaming a project is a simple matter of right-clicking on it in the project explorer view and selecting the "rename" action. Unfortunately, renaming the project leads to some issues with the old name still being used for makefiles and other internal properties. This means that some additional changes need to be made, in order to get the newly renamed project working. The remainder of this Section details how to go about making these changes.

First, right-click on the project again and this time select “properties”. In the resulting dialog, select “C/C++ Make Project” and replace the old project name in the “build directory” option. This allows the make utility to use the correct directory and so find our makefile, which will give errors without this change.

We also need to change the settings for the include paths. This is done from the same dialog as before, but in the “C/C++ Include Paths and Symbols” section. Remove all the paths under the “Source” folder and, with “Source” still selected, click on “add include path from workspace”. Add each of the subdirectories under the source folder in the project.

We should now be able to compile, run and debug the project. However, there will still be some traces of the old project name that we can also replace. For example: the name of the created binary is set in project properties (shown in Figure 3.8), the name of the generated map file is specified in the makefile (under “`LDFLAGS`”) and the name of the disk image can be changed in “`syslinux.cfg`” (detailed in Section 4.3.3, but another copy is located in the “Image” directory of the project).

4 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, as discussed previously.

If we choose to debug the project, RapidiTTy x86 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 x86 window).

4.1 Using the simulator

As with RapidiTTy FPGA, RapidiTTy x86 allows you to test your code in a processor simulator before committing to real hardware: this can help to accelerate the early stages of the development process. Selecting “run on simulator” in project properties will result in the dialog shown in Figure 4.1.

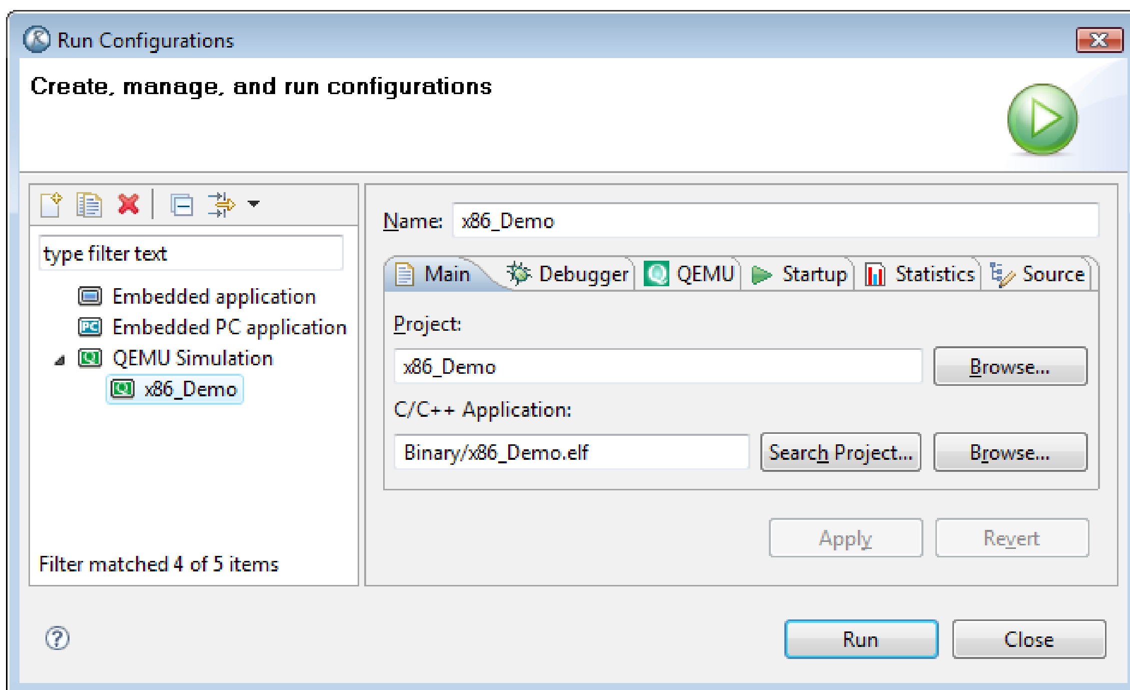

Figure 4.1: The run dialog for the simulator.

> The run dialog will only appear the first time you select the run link. The dialog can be seen at any time by selecting the "Run configurations" option from the "Run" drop-down menu on the toolbar.

For the simulator, the default settings should work correctly without alteration. If we simply click the run button, a QEMU window will open showing the output of the application.

As we chose the command shell example project, we are presented with a DOS or Unix style command prompt. This is shown in Figure 4.2, in which several commands have been issued ("help" and "lspci").

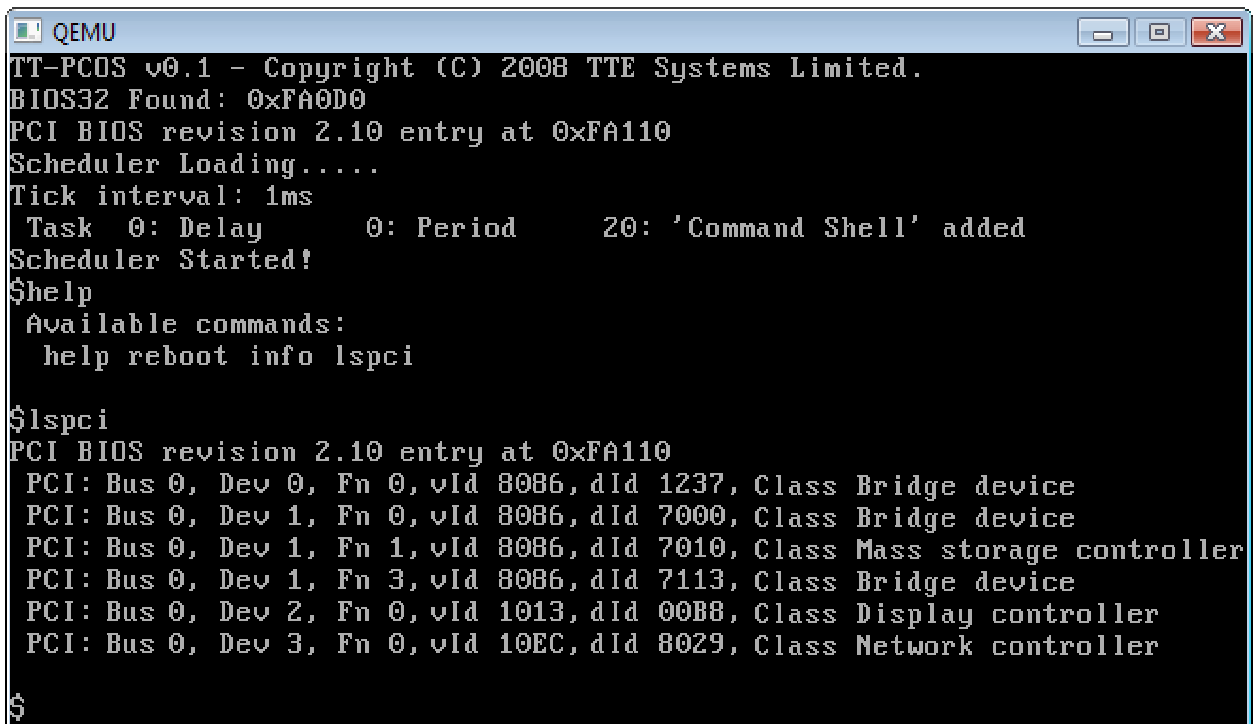

Figure 4.2: The command shell example running in the simulator.

4.2 Basic debugging

When we select one of the debugging actions from project properties we are taken to the debugging perspective. 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 4.3).

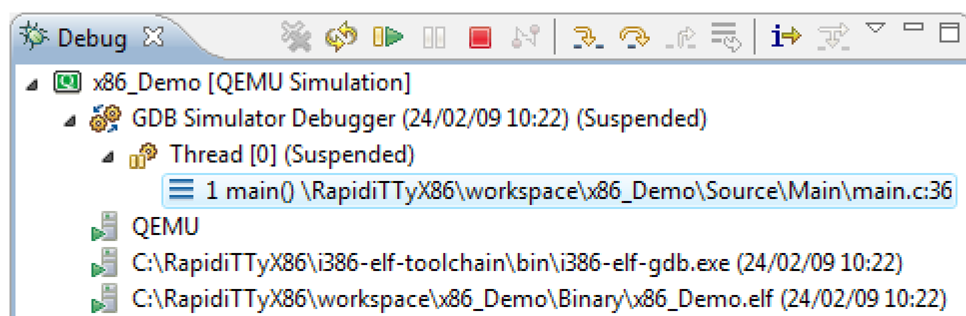

Figure 4.3: The debug view, showing the current stack trace.

There are two important things to note in Figure 4.3: the stack trace and the debugging toolbar. 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.

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 4.3), contains a number of essential actions for debugging. Some of the most important actions are shown in Figure 4.4. 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 4.4: The debug toolbar.

4.3 Creating a bootable USB flash drive

Before we can run or debug directly on the target system, the hardware must already be running an application with the GDB stub linked into it. This is something of a paradox – we must be running our application before we can run our application!

The simplest solution to this problem is to use the compiled binary of our new project to create a bootable USB flash drive. In this Section, we will accomplish this using SysLinux, but in reality any boot loader that conforms to the Multiboot specification should work.

4.3.1 Formatting the USB flash drive

Some USB flash drives come pre-installed with additional software and partitions that can be difficult to remove. If you do not have administrator privileges, or the method described here does not work, then you may wish to try a third party formatting utility.

With administrator privileges, formatting a USB flash drive should be relatively easy. Simply open "My Computer", right-click on the drive and select "Format...". In the resulting dialog we need to select FAT12, FAT16 or FAT32 for the filesystem type. We recommend that you carry out a full format (i.e. leave the "quick format" option unticked). Please be aware that this will erase all data on the flash drive!

4.3.2 Installing SysLinux

After downloading the latest version of SysLinux,[6] we can extract the "`syslinux.exe`" file (from the "win32" folder) and run it from a command prompt (usually found under the "Accessories" section of the start menu), as shown in Figure 4.5.

```
syslinux f:
```

Figure 4.5: Running SysLinux from the command prompt ("f" is the USB drive letter).

In Figure 4.5, the "`f`" is the drive letter of the USB flash drive. Replace this with the letter that Windows assigns to your specific device when you plug it in.

Next, we need to extract the "`mboot.c32`" file (from the "`com32/modules`" folder) and place it in the root directory of the USB flash drive.

4.3.3 Configuring SysLinux

The final step to prepare the USB flash drive is to include our application, and tell SysLinux about it. First, copy the application binary (the ".bin" file in the project) to the root directory of the USB flash drive. Next, create a "`syslinux.cfg`" text file (also in the root directory of the flash drive), based on Figure 4.6.

```
DEFAULT x86_Demo
LABEL x86_Demo
    KERNEL mboot.c32
    APPEND x86_Demo.bin
```

Figure 4.6: An example SysLinux configuration file.

In Figure 4.6, "`x86_Demo`" is the name of the project and "`x86_Demo.bin`" is the name of the generated binary file (that we just placed on the flash drive). These should be replaced by the correct names for the project in question.

With all these steps completed, it should now be a simple matter of booting the target hardware from the flash drive, which then runs our program. This may require alterations to the BIOS setup – please see the motherboard manual for details.

6 SysLinux can be downloaded from http://www.kernel.org/pub/linux/utils/boot/syslinux.

4.4 Debugging on hardware

With a GDB stub enabled application running on the target hardware, we can program and debug it directly through the serial link. To do this, select one of the debug or run "on hardware" actions from project properties. The first time we do this, we will be presented with the "Run" dialog (or the essentially identical "Debug" dialog).

When we used the simulator, we did not need to alter any configuration settings in the run and debug dialogs. With hardware, this is not the case. As we require a serial link for debugging, RapidiTTy x86 must be provided with configuration details, so it knows which COM port to use and at what speed. These settings are shown in Figure 4.7.

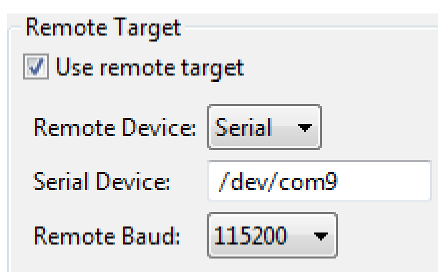

Figure 4.7: Configuring the serial port for debugging.

The COM number of the "Serial Host Device" setting in Figure 4.7 should be changed to the number of the COM port being used on the **host** machine (the one that RapidiTTy x86 is running on). The baud rate should be set to match the value chosen for the GDB stub at project creation.

Once these values are set, we can simply click on the debug (or run) button and RapidiTTy x86 will take care of programming the target hardware for us.

5 Timing analysis

In the debug dialog, there is a tab for "statistics". We can use this functionality to acquire data about the timing of tasks in the system. The basic configuration is very simple, for the command shell example we can simply select the options shown in Figure 5.1, then click debug to start acquiring data.

> Although the statistics tab is present in the launch configurations for the simulator, and for the "run" dialog, none of these are supported. If we want to acquire timing data, it must be done when debugging on actual hardware!

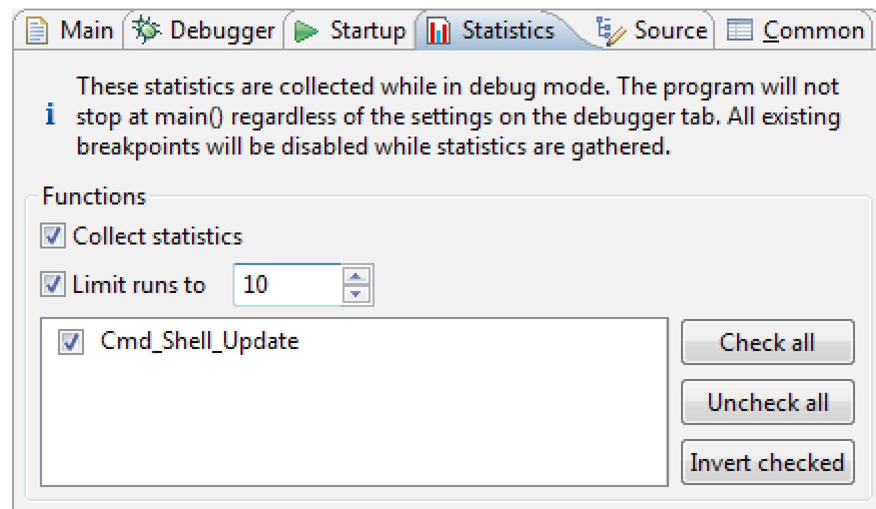

Figure 5.1: Setting configuration options for the acquisition of timing data.

Once the process of acquiring timing data has completed, debugging will automatically stop and RapidiTTy x86 will present a report of the data in a new editor. From here, we can see the tasks that were analysed and export the data to a variety of formats, as shown in Figure 5.2.

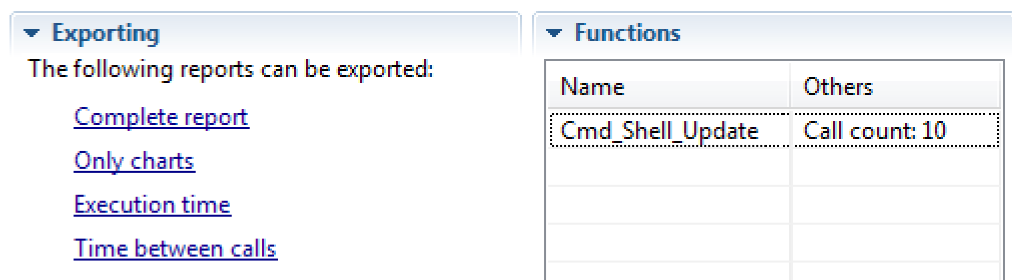

Figure 5.2: Export options in the timing statistics report.

Below this is a series of graphs showing the minimum and maximum task execution times, and the minimum and maximum time between task executions (which can be useful for jitter measurements). These are shown in Figure 5.3.

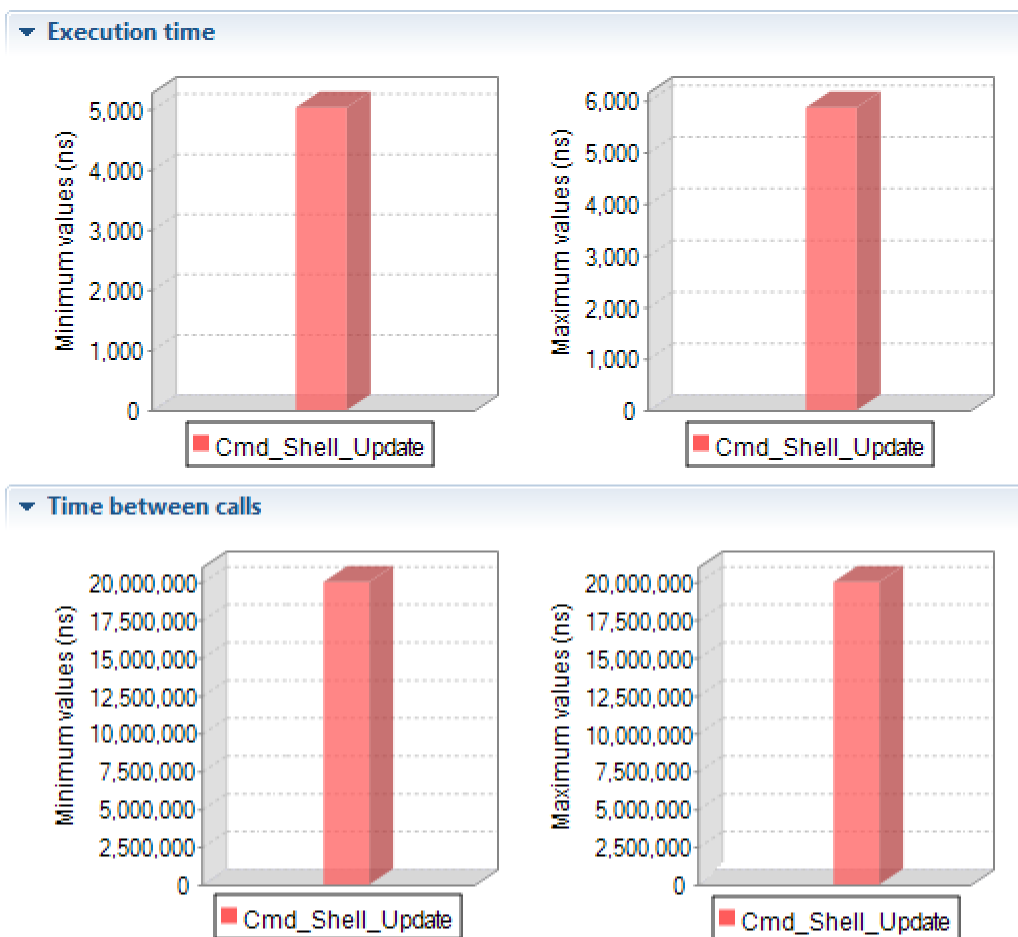

Figure 5.3: Graphical representations of the acquired data.

6 Drivers

RapidiTTy x86 includes a number of drivers for common activities in embedded systems. Some of these are detailed in this Section.

6.1 Keyboard driver

The keyboard driver contains a number of functions for accessing the keyboard data at a low-level, but there are two main functions that we can use ourselves: `kbhit` and `getch`. The former checks for available data, buffers it, and returns a non-zero value if data is available. The latter simply returns the next item from the buffer and increments the buffer index to point to the next value.

6.2 Serial port driver

By default, the serial ports are initialised with the `Serial_Init` function, which is called from the "`sys_init.c`" file. No further initialisation is required, although certain configuration options can be altered with the functions in "`serial.c`", such as: `Serial_SetBaud`, `Serial_SetFifo` and `Serial_SetMode`.

The two main functions needed to actually read and write data to the serial port are `Serial_PutChar` and `Serial_GetChar`. While these are self explanatory, simply writing and reading a single character at a time, we also need to know when we can transmit data and when there is data ready to receive. This is done with `Serial_IsTxRdy` and `Serial_IsRxData`, respectively.

6.3 VESA graphics driver

RapidiTTy x86 starts in text mode, but if the VESA driver (located in "`vesa_vbe.c`") is included, then the system initialisation will call the `VESA_Init` and `VESA_SetMode` functions for us. This sets the display resolution to 1024x768 with a 16-bit colour depth. The mode can be changed by passing a different value to `VESA_SetMode`, which takes one of the codes shown in Table 1.

Colour Depth	320x200	640x400	640x480	800x600	1024x768	1280x1024
4-bit				`0x102`	`0x104`	`0x106`
8-bit		`0x100`	`0x101`	`0x103`	`0x105`	`0x107`
15-bit	`0x10D`		`0x110`	`0x113`	`0x116`	`0x119`
16-bit	`0x10E`		`0x111`	`0x114`	`0x117`	`0x11A`
24-bit	`0x10F`		`0x112`	`0x115`	`0x118`	`0x11B`

Table 1: Codes representing the VESA graphics modes.

The remainder of the functions in the VESA driver are drawing routines designed to create simple user interfaces. The framebuffer can also be accessed directly through the `PhysBasePtr` member of the `modeInfoBlock` global variable.

> If we change the graphics mode for an example project, we may also have to change any source-code that accesses the framebuffer directly.

> The driver changes the VESA mode – going back to the default text-only mode will require removing the driver and a hard-reset of the target.

RapidiTTy x86

6.4 Hardware watchdog timer

Some motherboards include an IT8718 hardware watchdog timer. The driver for this can be found in "`hwatchdog.c`", which includes two main functions: `Init_WDT` and `Watchdog_Update`. The latter must be called periodically to reset the timer and prevent a reset from occurring. The exact period is set through the call to `Init_WDT`, which takes two parameters – the time-out period of the watchdog, and the units that the time-out is specified in.

6.5 Parallel port driver

Lacking the general purpose I/O pins found on embedded microcontrollers, applications running on desktop hardware may use the parallel port as a replacement.

Pin Number (DB-25)	Pin Number (Centronics)	Direction	Register	Bit	Inverted
1	1	Output	PAR_CNT	0	Yes
2	2	Either	PAR_DAT	0	No
3	3	Either	PAR_DAT	1	No
4	4	Either	PAR_DAT	2	No
5	5	Either	PAR_DAT	3	No
6	6	Either	PAR_DAT	4	No
7	7	Either	PAR_DAT	5	No
8	8	Either	PAR_DAT	6	No
9	9	Either	PAR_DAT	7	No
10	10	Input	PAR_STA	6	No
11	11	Input	PAR_STA	7	Yes
12	12	Input	PAR_STA	5	No
13	13	Input	PAR_STA	4	No
14	14	Output	PAR_CNT	1	Yes
15	32	Input	PAR_STA	3	No
16	31	Output	PAR_CNT	2	No
17	36	Output	PAR_CNT	3	Yes
18-25	19-30	Neither	N/A	N/A	N/A

Table 2: Pin details for typical parallel port hardware.

The parallel port driver (found in "parport.c") provides four functions to setup and access the ports. ParPort_Init is automatically called at system initialisation to find and setup all available parallel ports; we can check which ports are available at any time through the use of the ParPort_IsPort function, which takes the (zero-based) number of the parallel port to check for as a parameter.

The ParPort_OutP function sets the value of one of the parallel port registers, and the ParPort_InP function reads a register. The relation between individual pins on the port and the register that must be used can be seen in Table 2.

6.6 Expander module drivers

Although the parallel port provides up to 17 usable GPIO pins, they are restricted in how they may be used. Table 2 shows that some of the pins may only be used in a single direction, and some are actually inverted in hardware. This is a problem as one of the primary uses of RapidiTTy x86 is as a prototyping platform for projects that will eventually use microcontrollers. These may have a number of GPIO pins, as well as Analogue-to-Digital Converters (ADCs), Digital-to-Analogue Converters (DACs) and even Controller Area Network (CAN) interfaces.

In order to provide these facilities to users of RapidiTTy x86, we have created the TTE Systems I/O Expander Module. This consists of a number of external devices for GPIO, ADC, DAC and CAN support, all mounted on a single board and connected to the target computer with the Serial Peripheral Interface (SPI) via the parallel port.

6.6.1 SPI driver

As with many of the other drivers, the SPI driver will be initialised (through the use of the SW_SPI_Init function) at system initialisation, if the expander module drivers are present. As with most SPI implementations, the driver consists of two main facilities: chip selection (with SPI_CS_Write) and value transfer.

Value transfer works through the use of two shift registers, one in the hardware of the I/O module and one emulated in software on the target. We send and receive data at the same time through the use of the SPI_Transfer_Byte function. This takes one parameter that is placed in the target's emulated register, which is then swapped with the shift register in the expander module. The function then returns its new data (acquired from the swap), providing full-duplex communication.

> The SPI driver is not intended to be used directly. Instead, we can use any of the following drivers to access the individual hardware devices.

6.6.2 GPIO driver

The expander module provides two eight-bit GPIO ports, which are controlled from the GPIO driver (located in "`SPI_GPIO.c`"). Once again, the initialisation function (`GPIO_Init`) is called at system initialisation (by default), if the GPIO driver is a part of the project. We can set the mode of the pin (effectively the direction) using `GPIO_Set_Pin_Mode`. This function takes a parameter specifying the pin to be used, where values beginning at zero represent port A and values starting at 100 represent the pins on port B. The complete list of values for the mode setting is shown in Table 3.

Mode Setting	Meaning
GPIO_OUTPUT	Output
GPIO_INPUT	Non-inverting input with no pull-up
GPIO_MODE_IN_NO_PU_NON_INV	Non-inverting input with no pull-up
GPIO_MODE_IN_PU_NON_INV	Non-inverting input with pull-up
GPIO_MODE_IN_NO_PU_INV	Inverting input with no pull-up
GPIO_MODE_IN_PU_INV	Inverting input with pull-up

Table 3: The complete set of available options for the GPIO pin mode.

The remainder of the GPIO functions are duplicated for each port – that is, each function uses either "`GPIO_PortA_`" or "`GPIO_PortB_`" as its prefix. For example, the functions to read from and write to a specific pin are `Read_Pin` and `Write_Pin` (so the functions are actually named "`GPIO_PortA_Read_Pin`", etc).

Most of the GPIO functions come in an additional form – one to operate on a *range* of pins. For example, we can use `GPIO_Set_Pin_Mode` or we could use `GPIO_Set_Pins_Mode`. The latter takes an additional parameter that specifies how many pins are to be used; pins are always consecutive, starting from the original pin specified.

6.6.3 ADC driver

The ADC driver (located in "`SPI_ADC.c`") consists of just one function, `ADC_Read`, which sends a command to the ADC device and returns the result.

The return value of `ADC_Read` will range from zero to 4095, which will represent an input voltage of zero to 5 V.

The expander module provides an ADC with two channels, which can be used in either single or differential mode. As the ADC driver has no initialisation, we can specify these options directly to `ADC_Read`, as the command parameter. The values that can be used for this are shown in Table 4.

Command Parameter	Meaning
`ADC_READ_SINGLE_CH_0`	Single mode, channel zero
`ADC_READ_SINGLE_CH_1`	Single mode, channel one
`ADC_READ_DIFFERENTIAL_CH0_POSITIVE`	Pseudo differential mode, with channel zero as IN+
`ADC_READ_DIFFERENTIAL_CH1_POSITIVE`	Pseudo differential mode, with channel one as IN+

Table 4: Possible values for the command parameter of `ADC_Read`.

Table 4 shows that the ADC also provides a "pseudo differential mode". This mode produces a boolean comparison of the two channels, by assigning one of them as positive (IN+) and one as negative (IN-).

> In pseudo differential mode, if the IN+ channel value is greater than the IN- channel, the output will be 4095. Otherwise the output will be zero.

6.6.4 DAC driver

The DAC driver (located in "`SPI_DAC.c`") does not require initialisation. It uses two key parameters: gain (G, updated with `DAC_Gain_Change`) and the digital value (D_N, updated with `DAC_Write`).

> The gain parameter (G) must be either one or two – no other values are allowed. Similarly, the digital value (D_N) must be between zero and 4095 (as we are using a 12-bit DAC).

The relation between these parameters and the actual output voltage (V_{out}) is shown in Figure 6.1.

$$V_{out} = \frac{2.048 \cdot G \cdot D_N}{4096}$$

Figure 6.1: The output voltage, specified in terms of gain and digital value.

If we insert the minimum and maximum values into the equation shown in Figure 6.1, we can see that the ideal output voltage will range from zero to either 2.0475 V (if gain is one) or 4.095 V (if gain is two).

The expander module provides a DAC with two channels. In order to deal with this, each of the DAC driver's functions takes a channel parameter, which must be either `DAC_CH_A` or `DAC_CH_B`.

6.6.5 CAN driver

The CAN driver (found in “`SPI_CAN.c`”) provides functions for writing and reading to each of the two CAN controller's registers. These functions are `CAN1_Write`, `CAN2_Write`, `CAN1_Read` and `CAN2_Read`. The functions require only the register location – a complete listing of values for these registers can be found in “`SPI_CAN.h`”. No initialisation or configuration is necessary.

A full discussion about how to use the CAN interfaces is beyond the scope of this tutorial. Please see the datasheet for the Microchip MCP2515 CAN controllers for further details.

6.7 Hard disk drivers

RapidiTTy x86 provides driver support for accessing hard-disks and their associated filesystems. Both drivers are detailed in this Section.

These drivers are very new and still under heavy development. Please be aware that some features will be missing.

6.7.1 ATA driver

The ATA driver (located in “`ata.c`”) contains low-level code to work with storage devices – the main application of this driver is to provide support for hard-disks in RapidiTTy x86. Initialisation is carried out automatically at system initialisation with a call to `ATA_Probe`. After this has completed, we can simply use the functions `ATA_Read_LBA` and `ATA_Write_LBA` to read and write sectors, as needed.

The ATA driver is not intended to be used directly – consider using the FAT filesystem driver instead.

6.7.2 FAT filesystem driver

The File Allocation Table (FAT) filesystem driver (which can be found in "`fat.c`"), uses the ATA driver to provide support for FAT12, FAT16 and FAT32 filesystems for RapidiTTy x86. As with the other drivers, FAT support is (by default) initialised at startup, through a call to `FAT_Init`.

The FAT driver's initialisation sets up and enumerates all the volumes (drives, devices and partitions) present in the system at startup. Volumes are referred to by number – we can find out if a given volume exists by passing the number to the `FAT_VolExists` function. Further information about a given volume can be found by directly accessing the relevant index of the global "`Volume`" array, which is setup at initialisation.

The FAT driver supports both files and directories contained within each volume. These are specified by a number, which corresponds to the first cluster of the file or directory in question. The cluster number for a directory, when combined with a volume number and a name, can be passed to the `FAT_FindDir` function to get the cluster number corresponding to the subdirectory with the given name.

> When using the functions in the FAT driver, the cluster number for the root directory (of any volume) is always zero.

Once we have the cluster corresponding to a directory, we can find a file within that directory (and volume) using the `FAT_FindFile` function. A given cluster can be read into memory using the `FAT_ReadCluster` function, which must be provided with a pointer to a suitably sized area of memory – the size of a cluster, in bytes, can be calculated using the source shown in Figure 6.2.

```
uint32_t clusterSize = Volume[volNum].SecPerClus * Volume[volNum].BytsPerSec;
```

Figure 6.2: Calculating the size of a cluster for a given volume.

The `FAT_ReadCluster` function returns the number of the next cluster, so we can continue reading a file into memory by repeated calls to this function, until the `FAT_IsEOF` function indicates that a cluster number corresponds to the end-of-file marker.

> The FAT driver is still in heavy development and does not currently support write operations. It is, however, stable and easy to use for reading pre-recorded data from drives and devices.

7 Where do we go from here?

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

7.1 RapidiTTy x86

For RapidiTTy x86 support, or to ask general questions about embedded systems, you can visit our online discussion forum at http://www.tte-systems.com/forum.

7.2 Other products in the RapidiTTy family

RapidiTTy x86 is just one of the products in the RapidiTTy family. The other products are described in this Section.

7.2.1 RapidiTTy MCU

RapidiTTy MCU supports the development of code for "Commercial off-the-Shelf" (CotS) microcontrollers with ARM7® and Cortex-M3® cores. RapidiTTy MCU ships with the TTE Builder engine which helps the designer integrate, customise and configure an extensive suite of library code plus a royalty free version of the TTCos operating system.

Some of the key benefits of RapidiTTy MCU include:

- Full source-code for the Time-Triggered Co-operative Operating System (TTCos).
- A tightly integrated tool suite with support for ARM7 and Cortex-M3 cores.
- The TTE Builder engine, for rapidly building applications from prebuilt components.

7.2.2 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 PH 03 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 and hardware debugging peripherals.
  - Expansion through a simple peripheral bus.

7.2.3 RapidiTTy Professional

The RapidiTTy Pro product suite consists of all three development environments: RapidiTTy MCU, RapidiTTy FPGA and RapidiTTy x86 in one package.

7.3 Time-triggered systems

If you would like to know more about time-triggered systems, or embedded systems in general, you can find more information on our website (http://www.tte-systems.com). There you will find two free text books, “Rapid Development of Reliable Embedded Systems” and “Patterns for Time-Triggered Embedded Systems”, which discuss the development of time-triggered embedded systems in much greater detail.