Predictable performance on a multiprocessor system with RTEMS

Using RTEMS in a Nios II processor system

ROGER DAHLQVIST

Master's Thesis at ICT/ECS
Supervisor/Examiner: Ingo Sander

TRITA-ICT-EX-2009:165

Abstract

Developing applications for multiprocessor systems can be a really complex matter in terms of efficiency, communication and synchronization. This project aimed to run the Real Time Executive for Multiprocessor Systems (RTEMS), on a multiprocessor system with Altera Nios II processors. The current experimental port of RTEMS to Nios II, that was intended for single processor systems, was used as a starting point. A big part of the project was to tackle the problems caused by lack of tool support. After successfully obtaining a working toolchain, drivers were developed for the Multiprocessor Communications Interface layer utilizing all the multiprocessing aspects of RTEMS, resulting in a high performance, highly configurable and deterministic multiprocessor system.

Contents

Chapter 1

Introduction

The development today is going towards multi-core systems both in the field of embedded real-time systems as well as in desktop computers.

A real-time system has a set of complex characteristics that distinguish them from other computer systems. In general they have to be adhere to much more rigorous requirements regarding to time. The most important characteristic of a real-time system is therefore the need to respond to external events within a critical time frame. It doesn't matter if the calculations are correct if they don't complete in time, since breaking the time constraints can be catastrophic. The critical response times must be maintained at all times, even as the computational load may vary over time.

Another important characteristic of a real-time system is the ability to coordinate a large number of concurrent activities. Multitasking real-time systems allow an application to be divided into a number of logical tasks. A task represents activities that logically execute in parallel with other activities. In order to achieve the desired parallel execution, unlike the general synchronous nature of software, the activities combined must execute asynchronously.

The design of a real-time application is then further complicated if the tasks are distributed over multiple processors. This introduces some new challenges such as interprocessor communications and the management of global resources. Within real-time embedded systems it is not unusual with heterogeneous systems where different processor types perform different tasks. This obviously increases the complexity of the systems, but either way, there is most likely a need for mechanisms for the different processors to communicate and synchronize in order to cooperatively solve a task.

The Real Time Executive for Multiprocessor Systems (RTEMS) is a real-time operating system, or executive, that addresses all the above issues. Beyond the need for such an executive, another motivation for the development of RTEMS was the need

for a reusable foundation for developing portable, reusable, and efficient real-time applications.

RTEMS is a free open source project that has been ported to over a dozen architectures, and among them the Nios II processor from Altera. The port for Nios II should be considered experimental and has so far only been used in single processor systems. Another obstacle is the fact that there is no tool support for Nios II, since Altera don't submit their ports of the GNU tools to the Free Software Foundation (FSF). Due to this, the RTEMS project can't offer support for the Nios II port and since there is no obvious way to obtain a functional toolset for Nios II and RTEMS, the port itself and the included Board Support Package (BSP) has been neglected for a long time.

The first goal for this project was to obtain a toolset for Nios II, get the Nios II port up to date and finally have RTEMS up and running on a single processor system.

The next phase of the project was to setup a multiprocessor system with RTEMS. In a multiprocessor system RTEMS transcends the physical boundaries allowing the software developer to view the system logically as a single processor system. This was achieved by developing a driver based on a shared memory structure that is used by the RTEMS multiprocessing server. This driver defines the glue between the processors, the Multiprocessor Communications Interface (MPCI) layer.

This thesis resulted in a website (www.nios2rtems.com) with useful information and downloads for using RTEMS in a Nios II system.

Chapter 2

Background

2.1 Altera Nios II

The Altera Corporation is a major manufacturer of programmable logic devices, with the first re-programmable logic device invented in 1984. Their main products are a set of Field Programmable Gate Arrays (FPGAs), with the Cyclone, Arria and Stratix series devices, where Stratix is the largest and fastest with most features. while the Cyclone series devices are suitable for smaller, less demanding applications.

The FPGA device can be programmed to contain any hardware design, anything from simple logic designs to entire CPUs. CPUs used in this fashion on FPGAs are called soft-core processors. Once the device is programmed it will behave just like any traditional hard-core processor. Along with the hardware devices, Altera also offer a range of design software, where Quartus II is the main software suite for analysis and synthesis of HDL designs. Included in the Quartus II suite is the SOPC builder (System On a Programmable Chip), where you can automatically generate HDL designs for complete systems that incorporate both CPUs, memory and peripherals [4].

Altera provides the Nios II soft-core processor. Nios II is a 32-bit RISC (Reduced Instruction Set Computer) type CPU. The RISC architecture is an architecture that only allows a limited number of instructions, but on the other hand these instructions can be executed very fast and efficiently [3].

There are three types of Nios II processors available, compared in figure 2.1.

The Nios II EDS (Embedded Design Suite) that contains the toolchain (compiler etc) is used to build applications from C/C++ or Nios II assembler sources. Along with the IP cores provided by Altera, there is a software programming library, the Nios II HAL (Hardware Abstraction Layer). The HAL is a single-threaded environment that contains drivers for the different peripherals within the IP cores.

Model	**Nios II/e**	**Nios II/s**	**Nios II/f**
Features	**RISC 32-bit**	RISC 32-bit **Instruction cache** **Branch prediction** **Hardware multiply** **Hardware divide**	RISC 32-bit Instruction cache Branch prediction Hardware multiply Hardware divide **Barrel shifter** **Data cache** **Dynamic branch prediction**
Perf at 50 MHz	Up to 5 DMIPS	Up to 25 DMIPS	Up to 51 DMIPS
Logic usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory usage	Two M4K	Two M4K + cache	Three M4K + cache

Figure 2.1. Table showing features of the Nios II processor models.

2.2 μC/OS-II

μC/OS-II is a low-cost, priority based and preemptive Real Time Operating System (RTOS) intended for embedded systems. μC/OS-II is currently maintained by Micriμm Inc [6]. Use of the operating system is free for educational and non-commercial use, and is integrated with the Nios II EDS.

μC/OS-II is ported to most popular processors in the market and is suitable for use in safety critical embedded systems such as aviation, medical systems and nuclear installations.

The next generation of the kernel, μC/OS-III was released in March 2009. It includes many of the same features available in μC/OS-II, but there are some important differences. For example, it manages an unlimited number of tasks and features an interrupt disable time of near zero [7].

2.3 RTEMS

RTEMS is an acronym for the Real Time Executive for Multiprocessor Systems. It is a real-time executive (kernel) that provides a high performance environment for embedded systems. It is a free open source solution that supports multi-processor systems and has been ported to over a dozen CPU architectures and includes support for over 100 boards.

Among the features are multitasking capabilities, event-driven priority-based preemptive scheduling, priority inheritance, responsive interrupt management and a high level of user configurability.

RTEMS is designed to support applications with the most stringent real-time requirements while being compatible with open standards such as POSIX. RTEMS

includes optional functional features such as TCP/IP and various file systems while still offering minimum executable sizes below 20 KB in useful configurations.

The RTEMS project uses an open development environment in which all users collaborate to improve RTEMS. The RTEMS cross development toolset is based upon the free GNU tools and the open source C library newlib. Many development hosts are supported.

The development of RTEMS began in 1988 by the U.S. Army. Initially RTEMS stood for the Real-Time Executive for Missile Systems but as it became clear that the application domains that could use RTEMS extended far beyond missiles, the "M" changed to mean Military. When the maintenance of RTEMS was transferred to the OAR corporation in 1988, the "M" was changed to Multiprocessor.

RTEMS has been used by many companies and organizations over the years, and most recently in April 2009, the ESA (European Space Agency) Herschel and Planck satellites launched, and both contain systems with RTEMS. In September 2007 NASA's Dawn mission launched on a journey to the asteroid belt. It contains systems with SPARC Leon2 processors with RTEMS.

2.3.1 Key concepts

RTEMS provides directives which can be used to dynamically create, delete, and manipulate a number of object types, such as message queues, semaphores, memory partitions, timers and many more [5].

All objects are created on the local node (processor), and are automatically assigned an ID by RTEMS. Every object also has a user defined name, which is completely arbitrary, but is commonly used as a tag reflecting the use of the object in the application.

The managers designed specifically for communication and synchronization are semaphores, message queues, event and signal.

The semaphore manager supports mutual exclusion used to synchronize access to a shared resource. For a binary semaphore the optional priority inheritance algorithm may be used to avoid priority inversion. The message manager supports both communication and synchronization and the event manager will provide a high performance synchronization mechanism. The signal manager only provides asynchronous communication and is normally used for exception handling.

RTEMS maintains and supports time related operations based on the basic unit of time, known as a tick. The number of ticks per second is completely depending on the application and it determines the accuracy of the interval and calendar time operations. By tracking time in the units of ticks, RTEMS supports interval timing functions such as task delays, timeouts and the rate monotonic scheduling of tasks.

An interval is defined as the number of ticks relative to the current time, e.g. when a task delays for an interval of ten ticks, the task will not execute until 10 ticks have occurred.

What might be a problem if the clock granularity is large is that the interval period may be a fraction of a tick less than the interval requested. This is because of the time where a delay is set up may occur between two clock ticks. Because of this the first countdown tick occurs in less than the complete interval for a tick.

For some applications interval timing might not be sufficient when wall time or true calendar form is required, and due to this fact RTEMS also maintains date and time, i.e. time operations can be scheduled at an actual calendar date and time.

For all the directives that use intervals or wall time can't operate unless there is an external mechanism that provides a clock tick, i.e. a real time clock or a counter/timer device.

2.3.2 Board Support Package (BSP)

The core of RTEMS is a range of services that implement real-time tasking suitable for embedded systems. This core is generic, allowing it to be used on a wide range of processors. Within the core is support specific to a family of processors, yet generic to each target hardware that uses a specific processor. The BSP provides the glue between RTEMS and your target hardware.

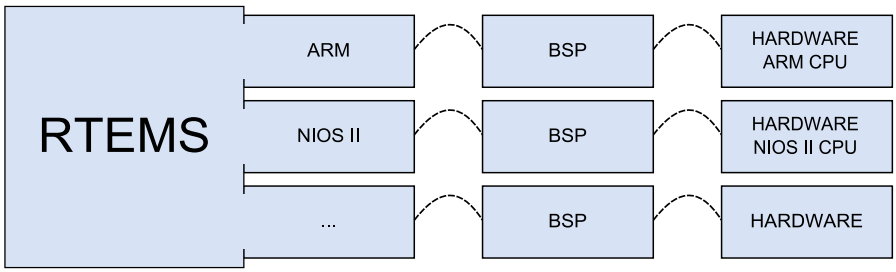


Figure 2.2. A BSP is the glue between RTEMS and the target hardware.

A BSP is a collection of pieces supporting your target hardware. It includes the linker script (memory layout), GCC customization script (e.g. bsp_specs), and startup code. If you want standard I/O, then the BSP must have a console device driver. Other drivers that may be needed include a clock tick driver to measure the passage of time, a real-time clock driver (RTC) and perhaps network drivers for networking.

You may also need a shared memory support driver for shared memory multiprocessing systems. An example of such a driver is shown in chapter 5.

2.3.3 Multiprocessor systems

Real time multiprocessor systems that share data and global resources require an efficient and reliable communications mechanism to communicate with each other when necessary. RTEMS provides simple and flexible real time multiprocessing capabilities, both for tightly-coupled and loosely-coupled target systems. RTEMS also supports both homogeneous and heterogeneous systems.

A design goal for the executive was to transcend the physical boundaries of the target hardware. This has been done by presenting the application software with a logical view of the target system where the processor boundaries are considered transparent, i.e. RTEMS allows both hardware and software to be viewed logically as a single system.

The tasks in a particular application can be spread among as many processors as needed to satisfy the timing requirements of the application. The tasks interact by using the RTEMS directive as if they were on the same processor. This allows the application tasks to communicate and synchronize regardless of which processor they reside upon.

RTEMS use the Multiple Instruction stream, Multiple Data stream (MIMD) execution model, having each of the processors executing code independently of the other processors in the system. Due to this parallelism, the application developer can more easily guarantee deterministic behavior.

In RTEMS a processor is referred to as a node. Each node is assigned, by the designer, a unique non-zero node number consecutively from one up to the number of nodes in the system. RTEMS use the node numbers to identify the node when performing remote operations, i.e. the MPCI layer must be able to route messages based on the node number. The node number zero is reserved and used by the executive when "broadcasting" to all nodes in the system. The MPCI layer is treated in chapter 5.

RTEMS objects can be created either as local objects or global objects. When an object is created with the GLOBAL attribute it will be known to, and referred to by all nodes, even though some restrictions may apply (e.g. you can't delete a remote object). The number of maximum global objects is configured by the designer (default is 32). Dynamic task relocation is not supported by RTEMS.

To keep track of the objects, RTEMS on all nodes maintain two tables containing information about the objects, a local objects table and a global objects table. The local objects table is unique and contains information on all the objects created on this node, i.e. the local node, regardless if the objects are local or global. The global objects table contains all the global objects in the system and consequently it is identical on all nodes. When a global object is created or deleted, all nodes in the system are informed and update their tables.

The application is unaware of the location of the objects it acts upon, however executing directives on remote objects take slightly longer. The amount of overhead for a remote operation is mostly depending on the node connection media, and to a lesser degree on the user provided MPCI routines. Overhead measurements are shown for a test system in chapter 7.

Chapter 3

Building RTEMS

This chapter will cover the general procedure of building an RTEMS development environment. Building RTEMS for Nios II is described in detail in section 3.2.2.

The process to setup a working RTEMS environment can be divided into three stages as follows.

1. Install GNU tools (General tools such as autoconf and automake)
2. Build toolchain (GCC and binutils)
3. Build RTEMS and BSP

It is very important that the tools installed meet the requirements for the version of RTEMS intended to build. For example, the tools below are recommended for RTEMS version 4.10.

- autoconf 2.63
- automake 1.10.1
- binutils 2.19
- gcc 4.4.x for C/C++ with newlib 1.17.0
- gdb 6.8

This thesis resulted in a website (www.nios2rtems.com) where verified sources, patches, scripts and examples are available for download.

3.1 RTEMS and Nios II

A port to the Altera Nios II processor was merged with RTEMS on August 9, 2006. This port was written and submitted by Kolja Waschk.

A sample BSP (Board Support Package) was included for use in the Altera ISS (Instruction Set Simulator) with a simulated JTAG UART and two timers. This BSP will work on an actual system if the memory layout and I/O addresses are changed accordingly.

As toolchain (compiler, binutils etc), Altera's nios2-elf-tools from the Nios II EDS (Embedded Design Suite) can be used with wrapper scripts turning them into nios2-rtems-tools. The new tools use the newlib C libraries compiled outside of the gcc tree. Section 3.2.2 includes detailed instructions on how to setup a working toolchain.

This port should be considered somewhat experimental, and there are still things to do before it can be considered production quality. For example data cache is not yet supported, so there is no need to use the Nios II/f processor. The author of the port has pointed out some specific things that should be fixed.

- Add C++ support
- Add support for data cache, FPU, ...
- Support more typical peripherals, like OpenCores.org Ethernet MAC

3.1.1 Problems and solutions

It might be hard for new users to find the RTEMS sources with the Nios II target included. This is because Nios II is not included in the release branches, and can only be obtained from CVS. The reason for this is that Altera has not added the tool support for Nios II to the FSF. In released versions, the RTEMS project can only support what they can build, and unfortunately Nios II doesn't meet that criterion.

There are ongoing efforts from the free software community to encourage Altera to submit the port of gcc, binutils, gdb and newlib to the upstream packages.

Another side of the problem is that the GNU tools used by Altera aren't really up to date. For example, RTEMS 4.10 won't build due to some errors caused by the old binutils used by Altera. Nios II EDS 8.1 uses binutils 2.15, but version 2.19 is what's recommended by RTEMS. Furthermore newlib 1.17.0 is required, but nios2-elf-gcc uses version 1.12.0.

Solutions to problems caused by old tools

Newlib 1.17.0 is built outside the gcc tree, as described in section 3.1, and is then included in the compilers search path. The binutils errors are easily fixed by applying a reverse patch on the two files affected (see 3.4.1).

3.2 Toolchain

Most RTEMS developers use the GNU Development Tools from the Free Software Foundation including the GNU Compiler Collection (GCC), GNU Binary Utilities (binutils), GNU Debugger (gdb), Autoconf, and Automake.

For some development hosts, prebuilt tools are available. If no prebuilt tools are available, the toolset for the desired architecture on your host has to be built from sources.

GCC supports numerous architectures and includes front ends for C, C++, Java, ADA and several other programming languages. RTEMS uses GCC with the newlib C library. The Binutils are a collection of binary tools, where the main tools are the GNU linker and the GNU assembler.

GCC and Binutils with Nios II support is maintained by Altera separately from the Free Software Foundation version. Unfortunately Altera haven't renewed their toolset versions, making them incompatible with recent versions of RTEMS. Detailed instructions for how to obtain a toolchain for Nios II is described in section 3.2.2

3.2.1 Newlib

Newlib is a C standard library implementation intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products. Newlib was created by Cygnus Support as part of building the first GNU cross-development toolchains. It is now maintained by Red Hat developers Jeff Johnston and Tom Fitzsimmons, and is used in most commercial and non-commercial GCC ports for non-Linux embedded systems.

Newlib is also used as the standard C library in Cygwin.

3.2.2 Toolchain for Nios II

As discussed in section 3.1, there is no easy way to build a toolchain for Nios II for use with RTEMS. Instead wrapper scripts will be used to turn the nios2-elf-tools [2] into nios2-rtems-tools and update the old version of newlib to a newer version required by RTEMS.

The following have been tested with Altera tools version 8.1 (Quartus II and Nios II EDS) and newlib-1.17.0 source snapshot. This toolchain has been used

to build RTEMS 4.9.99 as of the 6th of July 2009, i.e. the CVS head that is estimated to become release branch 4.10 in the autumn of 2009. Verified sources, scripts and patches are also available on the dedicated website[1].

Getting started

First of all some tools need to be installed, and the newlib sources need to be patched and prepared both for RTEMS and Nios II.

- Make sure autoconf and automake are installed on the host
- Download and Install Quartus II and Nios II EDS version 8.1 [1]
  http://www.altera.com
- Download the newlib-1.17.0 source snapshot
  ftp://sources.redhat.com/pub/newlib/index.html
- Obtain a newlib-1.17.0 patch for Nios II
  http://sourceware.org/ml/newlib/2009/msg00310.html
- Obtain the newest newlib-1.17.0 patch for RTEMS
  http://www.rtems.org/ftp/pub/rtems/SOURCES/4.10/
- Unpack the newlib sources and apply the patches. Start with the RTEMS patch, and then apply the Nios II patch.
- Generate files in folder *./newlib-1.17.0/newlib/libc/machine/nios2* with aclocal, autoconf and automake. Include the directory *./newlib-1.17.0/newlib* when running aclocal.

Setup the nios2-rtems-tools with wrapper script

A shell script is shown in appendix A.1. It may need to be edited to meet with your install paths of the Altera tools. The install point (`PREFIX`) is the one you intend to use for the entire RTEMS build.

These tools are used to build newlib, and then nios2-rtems-gcc will use the upgraded newlib libraries instead.

A required fix in Nios II EDS

Add the following lines to the end of the file below. Otherwise the newlib build will crash during the final stages, since some constants would be undefined. This was not an issue when building older versions of newlib, i.e. newlib-1.16.0 and older.

[1] www.nios2rtems.com

altera8.1/nios2eds/bin/nios2-gnutools/H-i686-pc-linux-gnu/lib/gcc/nios2-elf/3.4.6/include/limits.h

```
#ifndef _POSIX2_RE_DUP_MAX
#define _POSIX2_RE_DUP_MAX 255
#endif

#ifndef ARG_MAX
#define ARG_MAX 4096
#endif

#ifndef PATH_MAX
#define PATH_MAX 4096
#endif
```

Build newlib

The section below shows an example for how to build and install newlib. Be aware that the build may take anything from one hour up to several hours depending on the system you are using. Change the *prefix* tag to your install point for the entire RTEMS build.

```
> mkdir b-newlib
> cd b-newlib
> ../newlib-1.17.0/configure --target=nios2-rtems --prefix=/opt/rtems
> make all install
```

When the build is complete, it is recommended that you backup your install folder. This is a good idea if you need to have a clean install folder for RTEMS and don't wish to re-build newlib.

3.3 Nios II BSP

When using the HAL or μC/OS-II (discussed in 2.1) there is a tool available within Nios II EDS (nios2-bsp) that automatically creates a BSP for a specific CPU. Unfortunately no such tool is available for Nios II and RTEMS, and the BSP for a specific CPU has to be edited and built manually. This section covers the editing of the included Nios II BSP, to work with a simple system.

3.3.1 Configuring basic peripherals

The BSP included in RTEMS includes drivers for a small number of peripherals. There is a console driver, a clock driver and a timer driver (for RTEMS test suite). In this BSP, the addresses and IRQs are defined in the header file bsp.h.

rtems/c/src/lib/libbsp/nios2/nios2_iss/include/bsp.h

```
...
#define JTAG_UART_BASE 0x08000000
#define JTAG_UART_IRQ 2

#define CLOCK_BASE 0x08001000
#define CLOCK_FREQ 50000000
#define CLOCK_VECTOR 1

#define TIMER_BASE 0x08002000
#define TIMER_FREQ 50000000
#define TIMER_VECTOR 3
...
```

3.3.2 Memory layout

The linkcmds file is a linker script which is passed to the linker at linking time. This file describes the memory configuration of the board as needed to link the program. Specifically it specifies where the code and data for the application will reside in the memory.

The linkcmds must meet with the memory structure on your system, and the easiest way to get a functional script for a new system is to first generate it with the nios2-bsp tool, that is a part of the Nios II EDS. It generates the files for a HAL or μC/OS-II BSP, but only the linker.x file is of interest, i.e. the linker script. First copy this file and overwrite the old linkcmds file and make the changes shown below, but obviously with the necessary changes.

rtems/c/src/lib/libbsp/nios2/nios2_iss/startup/linkcmds

```
...
+ RamBase = DEFINED(RamBase) ? RamBase : 0x00800000;
+ RamSize = DEFINED(RamSize) ? RamSize : 0x00400000;
+ HeapSize = DEFINED(HeapSize) ? HeapSize : 0x0;
+ StackSize = DEFINED(StackSize) ? StackSize : 4096;
 MEMORY
 {
...
 __alt_mem_onchip_memory = 0x1004000;

+ __nios2_icache_size = 4096;
+ __nios2_icache_line_size = 32;
+ __nios2_dcache_size = 0;
+ __nios2_dcache_line_size = 32;
 OUTPUT_FORMAT( "elf32-littlenios2",
...
         . = ALIGN(4);
         __bss_end = ABSOLUTE(.);
+        _stack_low = ABSOLUTE(.);
+        . += StackSize;
+        _stack_high = ABSOLUTE(.);
+        WorkAreaBase = .;
     } > sdram
...
- PROVIDE( __alt_stack_pointer = __alt_data_end );
- PROVIDE( __alt_stack_limit   = __alt_stack_base );
+ PROVIDE( __alt_stack_pointer = _stack_high );
+ PROVIDE( __alt_stack_limit   = _stack_low );
...
```

3.3.3 Additional drivers

Since the default Nios II BSP, only has a small number of drivers supported, it might be a need to add additional drivers in order to use all aspects of RTEMS. Such drivers may include network drivers or a shared memory driver for use with RTEMS in multiprocessor mode.

Within the BSP folder, e.g. nios2_iss, edit the files *Makefile.am* (sources and headers) and *preinstall.am* (headers) accordingly, and then regenerate files manually or run bootstrap (see 3.4).

3.3.4 Additional BSPs

Before building/re-building RTEMS additional Nios II BSPs can be added if required, e.g. it might be convenient to build RTEMS with support for different hardware configurations at the same time. At compile time it is then easy to choose what BSP to use.

To include/create a new BSP. Make a copy of the nios2_iss folder and rename it to whatever the new BSP should be called.

1. Make a copy of *rtems/make/custom/nios2_iss.cfg*, and rename it to the new BSP name. Also edit the file according to the new BSP.
2. Add the cfg file created above to *rtems/make/Makefile.am*
3. Include the BSP subdirectory in *rtems/c/src/lib/libbsp/nios2/acinclude.m4*

Before building RTEMS you need to either run *bootstrap* or manually generate the appropriate auto-tools files.

3.3.5 Multiprocessor BSPs

The current BSP for Nios II is for a single processor system. One approach for a multiprocessor system would be to create a BSP for each CPU in the system, with drivers included for interprocessor communication, i.e. drivers defining the MPCI layer. This is the approach used in the test system analyzed in chapter 7.

3.4 Configuring and building RTEMS

As mentioned in section 3.1.1, RTEMS sources with Nios II support are only available from the CVS source repository.

```
cvs -d :pserver:anoncvs@www.rtems.com:/usr1/CVS -z 9 co -P rtems
```

3.4.1 Patching RTEMS for Nios II

Before bootstrapping there are some files to edit or patch. The most important for RTEMS to build at all for Nios II, are the makefiles in the *wrapup* directories, as shown below. This is required due to the old binutils used by Altera.

rtems/c/cpukit/wrapup/Makefile.am
rtems/c/src/wrapup/Makefile.am

```
...
-  ls $(ARCH)/* > $@-list
-  $(AR) rc $@ @$@-list
-  rm -f $@-list $(ARCH)/*.$(OBJEXT) $(ARCH)/*.rel
+  $(AR) rc $@ $(ARCH)/*
+  rm -f $(ARCH)/*.$(OBJEXT) $(ARCH)/*.rel
...
```

The changes below are only needed for sources dated prior to the 3rd of August 2009, as they were committed to the CVS repository. Furthermore, the changes in *shm_driver.h* are only required if RTEMS is built with multiprocessing enabled.

rtems/c/cpukit/score/cpu/Makefile.am

```
-  DIST_SUBDIRS = arm bfin h8300 i386 lm32 m68k m32c m32r mips no_cpu
+  DIST_SUBDIRS = arm bfin h8300 i386 lm32 m68k m32c m32r mips nios2 no_cpu
```

rtems/c/src/libchip/shmdr/shm_driver.h

```
...
  #elif defined(_AM29K)
  #define SHM_LOCK_VALUE    0
  #define SHM_UNLOCK_VALUE  1
+ #elif defined(__nios2__)
+ #define SHM_LOCK_VALUE    1
+ #define SHM_UNLOCK_VALUE  0
  #elif defined(__sparc__)
  #define SHM_LOCK_VALUE    1
...
```

3.4.2 Configure and install

Make sure the proper versions of autoconf and automake are installed for the version of RTEMS obtained, since RTEMS rely heavily on those tools. When the patches have been applied and the BSPs are properly configured, run bootstrap from the top level of the RTEMS source tree. It may take a few minutes to complete.

It's convenient to divide the configuration- and installation process into three stages, even though the BSP can be installed at the same time as RTEMS.

1. Run configuration
2. Make and install RTEMS
3. Make and install BSPs

The shell script in appendix A.2 shows an example of how to configure and install RTEMS and a BSP.

3.4.3 Exporting a build to another system

When the RTEMS build is complete it can be migrated to other computers and hosts. Just archive the install point folder (e.g. /opt/rtems), and unpack it on the other host. The nio2-rtems-tools needs to be wrapped if the install path of Nios II EDS is different on the new host, which it most likely is.

And finally, for each BSP, the *prefix*, and the *exec_prefix* variables located in the file *rtems/nios2-rtems/bsp_name/Makefile.inc* where *rtems* is the install point and *bsp_name* is the name of the BSP.

The test system in chapter 7 was built on Ubuntu 8.10 and then also migrated for use with the Nios II EDS on both Windows XP and Windows Vista.

Chapter 4

Building applications

RTEMS applications are built using makefiles for automatic building of executables. The makefile specifies how to derive the target program from each of its dependencies. An example makefile for an application is available in appendix A.3.

4.1 Configure an application

Before compiling an application, it has to be properly configured. This configuration includes the length of each clock tick, the maximum number of each RTEMS object that can be created, the initialization task and the device drivers used in the application. All this information is placed in data structures (configuration tables) and are passed on to RTEMS during initialization.

Most systems can easily be configured automatically by the RTEMS header file *rtems/confdefs.h*. It contains macros that specifies the values of almost all the configuration tables required by an application.

The code in appendix B shows a very simple "Hello world!" program. That system will begin execution with the single initialization task named *Init*. It's configured to have a console device driver (for standard I/O), but a clock tick device driver is not needed.

To make *rtems/confdefs.h* instantiate the configuration tables, the `CONFIGURE_INIT` constant has to be defined. It can only be defined in one source file per application, or errors will occur. By default, the values are normally set as low as possible and no application resources are configured.

`CONFIGURE_RTEMS_INIT_TASKS_TABLE` defines that the application should use the Classic RTEMS API. Further information about configuration can be found in the RTEMS C User's Guide [5].

4.2 Multiprocessor configuration

In the same way as most configuration tables, the multiprocessor configuration table can also be configured by the *rtems/confdefs.h* mechanism. The multiprocessor configuration table contains information about the node number, maximum nodes in the system, maximum global objects etc. It also contains a pointer to the multiprocessor communications interface table (MPCI table) which is a user defined table defining the MPCI layer, i.e. the "glue" between the processors in a multiprocessor system. The MPCI layer and the MPCI table are discussed in detail in chapter 5.

First of all RTEMS must be built with multiprocessing enabled and then the application is easily configured for multiprocessing by defining `CONFIGURE_MP_APPLICATION`. This will tell RTEMS to use *rtems/confdefs.h* to automatically configure the multiprocessor configuration table.

The MPCI pointer to the MPCI table defaults to *MPCI_table*, but can be changed by defining `CONFIGURE_MP_MPCI_TABLE_POINTER` to a pointer to the required MPCI table, e.g. `CONFIGURE_MP_MPCI_TABLE_POINTER  &my_MPCI_table`

The node number is normally set by the makefile at compile time, by defining `NODE_NUMBER`. That value is then automatically inserted in the multiprocessor configuration table.

The `NODE_NUMBER` constant can easily be used with the preprocessor within the source code of the application to distribute tasks over the nodes in the system.

Chapter 5

The MPCI layer

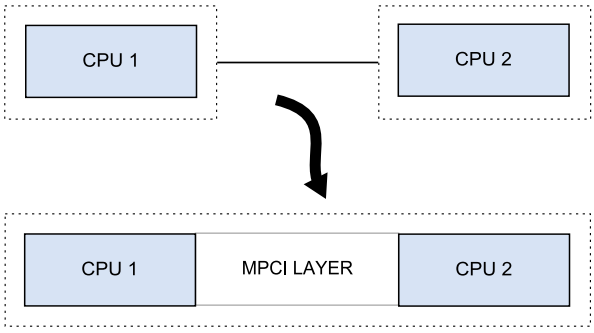


Figure 5.1. The MPCI layer transcends the node boundaries.

Since the design goal for the executive has been that both hardware and software should be viewed logically as a single system, there is a need for a mechanism to achieve this goal. The "glue" between the processors giving transparent node boundaries is the Multiprocessor Communications Interface (MPCI) layer, a number of user-provided routines providing for the interprocessor communication.

The RTEMS object managers can be used for communication and synchronization just as they are used on a single processor system, as long as the objects are created as global objects. The only difference when acting on global objects is that if the object resides on another node, there will be some overhead due to the interprocessor communication.

The user-provided implementation should define the multiprocessor communications interface table, the MPCI table, which defines the MPCI layer. The MPCI table is the connection to RTEMS and is the entry point of the entire MPCI layer.

The MPCI table contains the entry points to the MPCI components, but also the default timeout for the multiprocessing operations and the maximum packet size.

RTEMS requires a minimum packet size of 64 bytes.

```
typedef struct {
  uint32_t default_timeout; /* In ticks */
  uint32_t maximum_packet_size;
  rtems_mpci_initialization_entry initialization;
  rtems_mpci_get_packet_entry get_packet;
  rtems_mpci_return_packet_entry return_packet;
  rtems_mpci_send_entry send_packet;
  rtems_mpci_receive_entry receive_packet;
} rtems_mpci_table;
```

Besides the MPCI components, the user should also provide a mechanism for information sharing between the processors, i.e. some way to actually post and pickup packets.

During the initialization of the executive the multiprocessing server is started. It initially broadcasts a message to the other nodes to check for system consistency.

RTEMS needs to send a packet to other nodes in the following cases:

- A request message (RQ) is generated on the originating node
- A request response (RR) is generated on a destination node
- When a global object is created, to update the global object tables
- When a global object is deleted, to update the global object tables
- A local task blocked on a remote object that is deleted
- During initialization of the executive, to check for system consistency

When a new message arrives, the `rtems_multiprocessing_announce` directive should be called to announce the arrival of the new packet and invoke the multiprocessing server to process the information.

The `rtems_multiprocessing_announce` directive is typically called from an ISR. When available on the target hardware, the arrival of a new packet at a node should generate an interrupt. Alternatively a timer ISR can poll and check for new packets in a given interval. The first method is obviously the most efficient. The server will process the packet directly after exiting the ISR.

5.1 RTEMS data packets and envelopes

Before building the MPCI layer and the required communication structure, it is good to know what data RTEMS is using in the communication. The main information structure is the buffer, called a packet. The MPCI layer is responsible for managing a pool of packets available for the executive to send between system

nodes when required. The number of packets available is dependent on the MPCI layer implementation.

The packets contain the messages sent between the nodes. The following record contains the prefix for every packet passed between nodes in a multiprocessor system, the `rtems_packet_prefix` data type.

	Component	**Description**
0	the_class	Indicates the API class of the operation being performed.
1	id	The id of the object to be acted upon.
2	source_tid	The ID of the originating thread.
3	source_priority	The priority of the originating thread.
4	return_code	This is where the return status will be passed.
5	length	The length of the data following the prefix.
6	to_convert	The length of the data that require conversion.
7	timeout	Requested timeout for this operation.

The prefix contains all information required by RTEMS. The prefix is followed by the application data, e.g. messages and RTEMS internals. The maximum size of the data is the size defined in the *maximum_packet_size* component of the MPCI table minus the size of the packet prefix (32 bytes).

The packet is typically encapsulated within an envelope (data structure) which can contain additional information defined by the developer that might be needed by the MPCI layer. Such extra information might be information regarding endianness for conversion in heterogeneous systems, which is discussed further in section 5.2.

5.2 Heterogeneous environments

When developing the MPCI layer for a heterogeneous system, the understanding of the differences between the processors in the system is of great importance. One problem is the different orders of the bytes which compose a data entity, i.e. the endianness.

Processors which place the least significant byte at the smallest address are classified as little endian processors. Little endian byte-ordering is shown below:

Byte 0	Byte 1	Byte 2	Byte 3

On the other hand, processors which place the most significant byte at the smallest address are classified as big endian processors. The big endian byte-ordering is shown below:

Byte 3	Byte 2	Byte 1	Byte 0

When sharing data structures between little endian and big endian processors, endian conversion is required and is typically performed in either the sending or the receiving component of the MPCI layer. Unfortunately this kind of conversion gives an undesirable overhead, so choosing a common endian format is something to strive for.

Yet another problem that may occur when designing shared data structures is the alignment of the data elements. For example some processors allow data elements to begin on any address when others impose restrictions. Some common restrictions are that the data elements must begin on either an even address or on a long word boundary, i.e. a four byte boundary.

Some other issues that may impact the design of a shared data structure are floating point number representation, bit fields, decimal data and character strings. The method for representation of negative integers can be one's complement or two's complement.

All the issues above combined increase the complexity of designing data structures and sharing data structures between processors in a heterogeneous system.

In the same way as the developer doesn't have to worry about the interprocessor communication since all parts of the communication are handled by the MPCI layer, all the conversions necessary regarding to a heterogeneous system should be performed by the MPCI layer.

5.3 The MPCI layer components

The MPCI layer consists of five different components caring for the entire interprocessor communication. The send and receive components require some kind of mechanism to send and receive messages respectively. Each of the components should be adhere to a given prototype required by RTEMS. The components and their part of the MPCI layer is discussed in the following five subsections.

5.3.1 INITIALIZATION

This component of the MPCI layer is only called once, during the initialization of the executive, directly after the initialization of the device drivers. It should initialize the MPCI layer and the associated hardware. The *initialization* component should be adhere to the following prototype:

```
rtems_mpci_entry mpci_initialization( void );
```

Since RTEMS requires packet buffers for communication with other nodes in the system, the executive needs to be able to obtain a packet whenever needed. The *initialization* routine is supposed to create and initialize a pool of packet buffers, or envelopes if the packets are encapsulated. There MUST be enough packets so that RTEMS always can obtain a new packet from the pool of packets.

If for some reason, the MPCI layer isn't properly initialized, the fatal error manager should be invoked.

5.3.2 GET PACKET

When RTEMS needs to obtain a packet buffer to send or broadcast a message to the other nodes in the system, this component of the MPCI layer is called. It should be adhere to the following prototype:

```
rtems_mpci_entry mpci_get_packet( rtems_packet_prefix ** packet );
```

The *packet* argument is the address of a pointer to a packet. Upon return *packet* should contain the address of a packet.

RTEMS is optimized to reuse packets when possible in order to avoid having to obtain new packets every time a message has to be sent to other nodes. For example a response message (RR) is sent back to the originator using the same packet as the request message (RQ) arrived.

The fatal error manager should be invoked if a packet cannot be obtained from the pool of packets.

5.3.3 RETURN PACKET

When RTEMS has used a packet and have no more use of it, the packet needs to be released to the pool of packets. This component should be adhere to the following prototype:

```
rtems_mpci_entry mpci_return_packet( rtems_packet_prefix * packet );
```

The *packet* argument is the address of a packet, i.e. the address of the packet to return.

Unless the packet can be successfully returned, the fatal error manager should be invoked.

5.3.4 SEND PACKET

Whenever RTEMS need to send a packet with a message to another node in the system, the *send_packet* routine is called. This component should be adhere to the following prototype:

```
rtems_mpci_entry mpci_send_packet( uint32_t node, rtems_packet_prefix * packet );
```

Here *node* is the node number of the recipient and *packet* is the address of the packet containing the message generated by RTEMS.

This routine is responsible for routing the messages to the destination nodes. If the node number is zero, the packet should be broadcasted to all other nodes in the system. i.e. a copy of the message should be sent to all other nodes.

For a heterogeneous system this might be a conversion point. How much of the packet to convert is given in units of 4 bytes in the *to_convert* field of the packet prefix.

If the packet cannot be successfully sent, the fatal error manager should be invoked.

5.3.5 RECEIVE PACKET

When a new packet has arrived to the node and RTEMS needs to obtain it, this routine is called. It should be adhere to the following prototype:

```
rtems_mpci_entry mpci_receive_packet( rtems_packet_prefix ** packet );
```

Here *packet* is the address of a pointer to a packet. The RTEMS receive server will loop this routine for as long there are new messages in queue. If a message exists, then *packet* will contain the address of the received package. If, on the other hand, there are no messages available, *packet* should be set to NULL.

Chapter 6

An MPCI implementation

This chapter will discuss how the MPCI layer was implemented within the boundaries for this thesis, and this is also the MPCI layer used in the dual core system used for testing in chapter 7.

Even though this implementation was for the Nios II processor, the approach is generic and should be feasible to implement on any processor family. The aim with this chapter is not to present the entire source code for the working MPCI layer, but in detail describe the MPCI components, ISRs, encapsulation, useful macro functions etc, i.e. the overall design issues. The entire implementation is available for download on the dedicated website[1].

The MPCI source code files are organized as shown in the figure below, and the drivers are included in the individual BSPs in the way described in section 3.3.3.

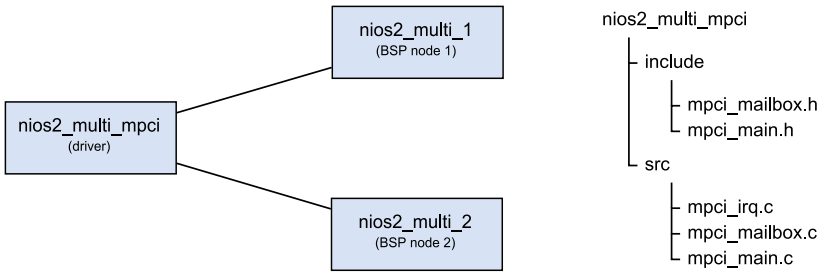


Figure 6.1. Single core BSPs joined together by the MPCI layer.

As a starting point the goal was to create a MPCI layer that was easy to configure and implement using a shared memory device. The routines are built for a system with single processor BSPs, where all BSPs in the multiprocessor system include the same MPCI sources, as shown in figure 6.1. This makes it easy to automatically

[1] www.nios2rtems.com

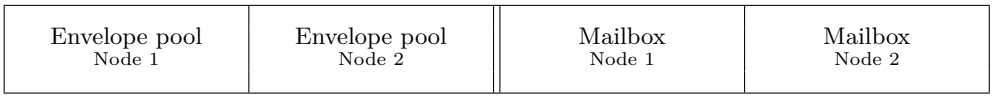
configure the respective MPCI layer depending on their node numbers, given at application compile time.

This MPCI implementation currently doesn't support any heterogeneous systems.

6.1 Structures and configuration

The header file in appendix B.2 gives an idea of the structure of the MPCI layer.

The MPCI layer uses a memory device shared between all the nodes in the system. The memory is divided into two sections where the first one is dedicated to the pools of envelopes (encapsulated packets) and the other section is dedicated to the mailboxes, i.e. message passing mechanisms that passes package pointers between the nodes.



Within each of the two sections each node is consecutively assigned a region of the shared memory starting from the base address, i.e. node 1 gets the first region, node 2 gets the second and so on. Given the node number, it is easy to calculate what memory regions correspond to it. This fact is used throughout the MPCI layer and it keeps the manual configuration to a minimum.

The excerpt below show the small configuration part of the MPCI sources. If polling is used to check for new messages, these are the only things that ever need to change. If the interrupt generation method is used, there are a few routines that has to be defined by the user to meet the hardware used.

Excerpt from the file mpci_main.h shown in appendix B.2

```
#define MPCI_SHARED_MEMORY_BASE (void *)  0x01004000  // Base address for shared memory
#define MPCI_MAILBOX_SIZE                 1024        // Mailbox size in bytes
#define MPCI_ENVELOPE_POOL_SIZE           1024        // Envelope pool size in bytes
#define MPCI_ENVELOPE_SIZE                128         // Envelope size in bytes
#define MPCI_NODES                        2           // Number of nodes (= master node)
#define MPCI_INTERRUPT                    1           // 1 = interrupt, 0 = polling using timer
#define MPCI_IRQ                          3           // IRQ if MPCI_INTERRUPT is set to 1
```

In the configuration above 1024 bytes are reserved for the envelope pool, and with the envelope size of 128 bytes this means that there are 8 envelopes in the pool on each node.

The envelope is defined as the data structure below:

Excerpt from the file mpci_main.h shown in appendix B.2

```
typedef struct {
  vol_u32   index;
  vol_u32   in_use;
  vol_u32   not_used1;
  vol_u32   not_used2;
  vol_u8    packet[MPCI_MAXIMUM_PACKET_SIZE];
} mpci_envelope;
```

Here the first two elements are used by the *get_packet* and *return_packet* components of the MPCI layer. The two following elements were deliberately left unused, free to use in future development when other information may need to be passed between the nodes. The `MPCI_MAXIMUM_PACKET_SIZE` constant is defined as the envelope size minus the envelope prefix size, and in this case it's 96 bytes which is more than the minimum of 64 bytes required by RTEMS.

6.2 The Mailbox

The message passing mechanism used to send envelope pointers between the nodes, the mailbox, is implemented as a circular buffer used as a FIFO queue. The first part of the mailbox base address (24 bytes) consist of the mailbox control structure, with the elements owner, value, slots, used, head and tail. The *owner* and *value* elements are used for mutual exclusion. The element *slots* is the maximum number of objects that fit in the queue and is an integer calculated from the setting in the configuration for the MPCI layer, while *used* is the number of used slots. The *head* and *tail* elements are the pointers to the beginning and the end of the circular buffer respectively.

These are the directives within the mailbox mechanism:

```
void mpci_mailbox_init( void * base, uint32_t size );
void mpci_mailbox_lock( void * base );
void mpci_mailbox_release( void * base );
uint32_t mpci_mailbox_post( void * base, uint32_t msg, uint32_t target_node );
uint32_t mpci_mailbox_get( void * base, uint8_t * error );
uint32_t mpci_mailbox_messages( void * base );
```

The `mpci_mailbox_post` directive also calls the user defined routine `mpci_intr_set` when the MPCI layer is configured to generate an interrupt for new messages. The header file as a whole is available in appendix B.3

6.3 MPCI components

Initialization

The node with the highest node number is the master node, and that node should be the first to be downloaded to the FPGA. The initialization routine begins with synchronizing the nodes, i.e. they wait until all nodes are downloaded and started. When all nodes are started they aren't released until the shared memory has been cleared by the master node.

When released, the initialization process continues on all nodes, by initializing the mailboxes and the pools of envelopes. The last thing done in the initialization routine is to set an ISR for the selected method to detect new messages.

Below are example ISRs for both methods, and they should be set using the RTEMS interrupt manager or the timer manager, depending on the method selected.

```
/*************************************************************************
 * ISR for interrupt mode. User defined functions are required
 *************************************************************************/

rtems_isr mpci_isr( rtems_vector_number vector ){
  mpci_intr_reset( MPCI_NODE_NUMBER );
  if ( _System_state_Is_up(_System_state_Get()) ){
    if( mpci_mailbox_messages( mpci_mailboxes[MPCI_NODE_NUMBER] ) > 0 )
      rtems_multiprocessing_announce();
  }
}

/*************************************************************************
 * TSR for polling mode using the timer to check for new messages
 *************************************************************************/

rtems_timer_service_routine mpci_tsr( rtems_id  id, void * data ){
  if ( _System_state_Is_up(_System_state_Get()) ){
    if( mpci_mailbox_messages( mpci_mailboxes[MPCI_NODE_NUMBER] ) > 0 )
      rtems_multiprocessing_announce();
  }
  rtems_timer_reset( id );
}
```

If the interrupt generation method is used the hardware generating the interrupt should also be initialized if necessary.

Get packet and return packet

The *get_packet* routine first creates an envelope pointer pointing to the base of the envelope pool belonging to the node. The routine then iterates thru the envelopes until an unused envelope is found. That envelope is marked as being used by setting the *in_use* element in the envelope prefix to one. Then the envelope pointer is converted to an RTEMS packet prefix pointer and is returned to RTEMS according to the prototype (5.3.2).

```
#define MPCI_envelope_to_packet_ptr(env)((void *)(env)->packet)
#define MPCI_packet_to_envelope_ptr(pkt)((mpci_envelope *)((uint8_t *)(pkt)-MPCI_ENVELOPE_PREFIX_SIZE))
```

The two macro functions above are included in the main header file for the MPCI layer, and are very useful for conversions between envelope pointers and packet pointers.

When RTEMS is done using a packet and wish to return it, The packet prefix pointer RTEMS provides as argument is converted to an envelope pointer which is used to set the *in_use* element of the envelope to zero, i.e. unused and free to use in the pool.

Sending packets

As previously seen in section 5.3.4, the prototype for the *send_packet* routine is:

```
rtems_mpci_entry mpci_send_packet( uint32_t node, rtems_packet_prefix * packet );
```

The *node* above is the destination node for the message. If *node* is set to zero, it means that the message should be broadcasted to all other nodes in the system.

The *packet* argument is a pointer to the packet that RTEMS previously obtained from the *get_packet* routine and that now contains the message to be sent. This pointer is converted to an envelope pointer, which is possible since all packets are encased in envelopes.

If node does not equal zero, there is only one recipient and the message, i.e. the envelope pointer, is posted to the mailbox corresponding to the destination node. If the message is to be broadcasted in the system the package generated by RTEMS has to be copied so every node gets a unique copy. Obviously there is only a need to make a copy if the number of nodes in the system are greater than 2. Since the source packet also can be sent as it is, the number of copies is the number of nodes minus two.

To create a copy of a packet, first create a packet prefix pointer and use the address of it as argument when calling the MPCI component *get_packet*. Upon return the created pointer now points to a newly obtained packet. After conversion to an envelope pointer the original packet has to be copied.

The structure below, from the main MPCI header file is very useful when copying the packet.

```
struct pkt_cpy {
  uint32_t   packet[MPCI_MAXIMUM_PACKET_SIZE/4];
};
```

When the packet has been copied, it's posted to the appropriate mailbox. This procedure is repeated for as many times as required.

Receiving packets

The *receive_packet* routine is called by the RTEMS receive server until the mailbox is empty. As discussed in earlier chapters, the prototype for this routine is:

```
rtems_mpci_entry mpci_receive_packet( rtems_packet_prefix ** packet );
```

When getting a message from the mailbox, the "error" variable is used to tell if the message received is valid, If the error variable is zero then *packet* is set to NULL. If the message received is valid, *packet* is set to the package pointer received after conversion of the received envelope pointer.

6.4 A communication example

The following will explain the resulting behavior of the MPCI components and show how the shared memory is used in the interprocessor communication. In reality it is not unusual that the envelope is reused, i.e. the same envelope received at node 2 below is used to send a response to node 1.

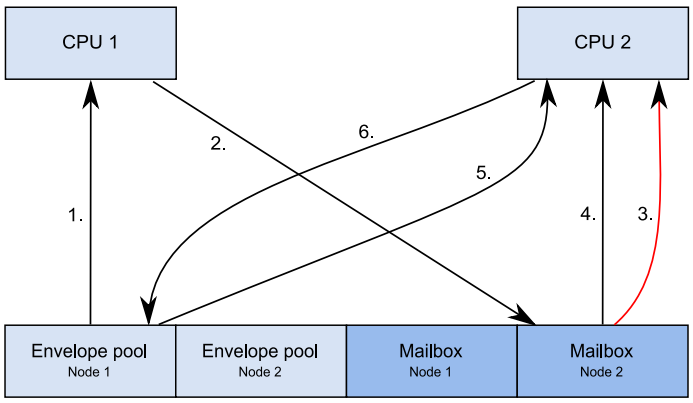


Figure 6.2. How the shared memory is used in the communication.

1. RTEMS on node 1 needs to send some kind information to node 2. An envelope is obtained from the pool of envelopes and filled with data.
2. The pointer to the obtained envelope is posted in the mailbox for node 2.
3. After the pointer has been posted, an interrupt is generated on the destination node when supported. Otherwise a polling ISR will detect the message arrival.

4. The newly arrived envelope pointer is retrieved from the mailbox.
5. The data in the envelope is processed by RTEMS on node 2.
6. The envelope is released and returned to the pool of envelopes.

Chapter 7

Performance analysis and testing

A simple test system has been generated in SOPC Builder[1] for use on the Altera DE2 development board. The test system was used during the testing of RTEMS and the development and verification of the MPCI layer. The system was also used to carry out some performance measurements both for RTEMS and μC/OS-II.

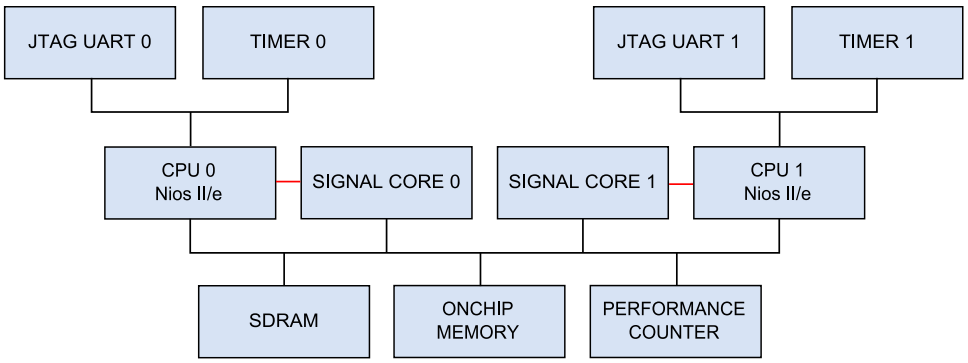


Figure 7.1. Hardware structure of the test system.

The test system consists of two Nios II/e processors where each of them are connected to both a timer device and a JTAG UART. The CPUs are assigned 4 MB each from the 8 MB SDRAM. The performance counter, with drivers re-written to support RTEMS, is used for the performance measurements.

RTEMS will use the onchip memory as the shared memory used by the MPCI layer. The signal cores, created as a part of another KTH thesis [8], are used for the MPCI layer to generate interrupts on the destination nodes when sending messages. The red lines in the figure above, indicate the IRQ lines.

[1] Part of the Quartus II design software by Altera

As the final stages of the project required lots of testing of the MPCI layer, there was only time to carry out a limited amount of performance measurements within the time-frame of the thesis. However, some characteristics were measured and are presented in the following sections.

7.1 RTEMS multiprocessing

One very important aspect of the multiprocessor system is the communication and synchronization between the system nodes. As discussed earlier in this report, the application is unaware of the location of the objects it acts upon. The only difference when executing directives on remote objects, is the overhead for the interprocessor communication. This might be an important factor when distributing the tasks in the system.

Section	Time (sec)	Time (clocks)
IRQ latency	0.00012	6338
Post remote	0.00298	149222
Receive remote	0.00296	147831
Post local	0.00024	12133
Receive local	0.00023	11332

From the measurements above the overhead for acting on remote objects is estimated to about 137000 clock-cycles or 2.7 ms on the 50 MHz Nios II/e processor.

Below are test results from sending messages between tasks on the local node and also between tasks on different nodes. They are sent in a round-trip fashion, i.e. a message is sent from one task, received on the other task and then returned. Two message queues are used to do this, one for each direction. The message receive call is blocking which leads to context switches.

Section	Time (sec)	Time (clocks)
Round trip G	0.00492	246196
Round trip L	0.00137	68434

The time for one round-trip on the local node contains two *post* calls, two *receive* calls and two *context switches*. From the results for the local node test and the previous measurements the context switch time can be estimated to about 11000 clock-cycles or 0.2 ms.

As mentioned in section 2.3.3, the amount of overhead for a remote operation is mostly depending on the node connection media and to a lesser degree the MPCI routines. When acting on a remote object a request message and a request response are sent. Using this knowledge and the fact that there are also two *context switches* and two *receive* calls, the estimated context switch seems like a proper value.

Another parameter that might be of interest is the IRQ latency. Measurements determines the IRQ latency of the system to about 6500 clock-cycles. Even if it's obvious that the use of "interrupt generation" in the MPCI layer is the most efficient way to go, the IRQ latency shows the motivation for it. When using the polling mode, each timer interrupt will "cost" the IRQ latency of 6500 clock-cycles but also the time it takes to check if there are new messages waiting. The amount of lost time is depending both on the polling rate and the rate of incoming new messages from the other nodes.

7.2 RTEMS vs μC/OS-II

Similar measurements as the ones carried out on the system with RTEMS were made with μC/OS-II instead. Since μC/OS-II don't have multiprocessor support, only the communication between tasks on a local node was analyzed.

The round trip message transfers using two message queues will on average take about 55000 clock-cycles which is a bit less time than the 69000 clock-cycles for RTEMS. Included in these "round trips" are two posts to the queue, two received messages and two context switches.

Section	Time (sec)	Time (clocks)
IRQ latency	0.00032	16102
Round trip	0.00111	55456
Post message	0.00008	3749
Receive message	0.00005	2453

Using the same approach as in the previous section the context switch time is estimated to about 21 000 clock-cycles or 0,4 ms.

The difference in *post* and *receive* times between μC/OS-II and RTEMS may be explained by that the message queues in RTEMS are more complex and offer more options, variable message sizes and automatic memory allocation.

For μC/OS-II the IRQ latency is approximated to about 16000 clock-cycles, which is almost 2,5 times the IRQ latency for RTEMS.

Chapter 8

Conclusion

The Real Time Executive for Multiprocessor Systems (RTEMS) is a powerful, high performance real-time operating system that provides the application developer with highly configurable mechanisms to address the common issues that arise when designing real-time embedded systems.

RTEMS allow a high level of user configurability both in the object managers and in the general application configuration, but when using the preset defaults only a minimum amount of configuration is required from the developer. Even compared to a small RTOS like μC/OS-II (see 2.2), RTEMS is much more stringent and easy to use. This is thanks to the different object managers in RTEMS that not only manages the objects, such as semaphores, message queues and tasks, but also automatically handle things like the memory allocation which for example has to be manually allocated in μC/OS-II.

Besides the basic features expected from a real-time executive, RTEMS supports networking with everything from TCP/IP to web servers, and file systems such as NFS and IMFS using the POSIX standard. μC/OS-II also supports these features but only as separate add-ons in order to keep the kernel small and easy to use. In RTEMS all extra features such as networking and POSIX are enabled or disabled during configuration before building the executive.

RTEMS is without doubt more heavyweight than μC/OS-II, but another major difference is that RTEMS also provides support for multiprocessor systems. RTEMS successfully allows the multiprocessor system to be viewed logically as a single processor system. The application makes no difference between acting on an object on the local node or on an object residing on another node in the system.

Unlike μC/OS-II, RTEMS is a free open source solution and is always in development by the users in the open source community. New users are encouraged to start using RTEMS from the sources in the CVS head instead of the release branches. The benefit of doing this is that new developments can be committed to the CVS

repository, and will be available in future release branches. That was the case within this project with several updates committed. An unfortunate downside of working with the "current sources" is that unexpected bugs may appear. Within this project one major bug introduced itself in the *multiprocessor receive server* that executes directives received from other nodes. This caused the entire system to fail and halted the project for weeks, until a workaround was made and the project could continue. Finally other users in the community were able to identify the cause and commit a proper solution to the CVS making the workaround obsolete.

RTEMS relies on the different vendors to submit their ports of the GNU tools to the Free Software Foundation. The toolchains used to build RTEMS can then easily be built from sources or downloaded as prebuilt tools for the specific processor. Unfortunately Altera has not submitted the tool support for Nios II to the FSF, even though they have been encouraged to do so. This makes it much harder for users who wish to use RTEMS with Nios II as seen in this project, and it most certainly has impacted further development of the Nios II port and BSPs in a negative way.

If Altera start to submit the tool support to the FSF like most major vendors do, the development of the Nios II port and BSPs would most likely increase rapidly, making it feasible that Nios II some day may be used in an embedded system on board a space mission.

Bibliography

[1] James Bassett. *Installation Guide for Quartus on Ubuntu Linux*, 2007.

[2] Altera Corporation. *Nios II Software Developer's Handbook*, 2007.

[3] Altera Corporation. *Nios II Processor Reference Handbook*, 2008.

[4] Altera Corporation. *Quartus II Handbook Version 9.0*, 2009.

[5] OAR Corporation. *RTEMS C User's Guide*.

[6] Micrium Inc. *μC/OS-II datasheet*.

[7] Micrium Inc. *μC/OS-III datasheet*.

[8] Fredrik Lindberg. Master's thesis. *Embedded Linux - MicroC/OS-II Platform for Real-Time Systems*, 2009.

Appendix A

Shell scripts and makefiles

A.1 Toolchain wrapper script

```
#!/bin/bash

BASEDIR=$PWD

# Install point
PREFIX=$BASEDIR/install # (i.e. path to rtems install folder)

# Paths to Altera tools
SOPC_KIT_NIOS2="/opt/altera8.1/nios2eds"
NIOS2_GNUTOOLS="$SOPC_KIT_NIOS2/bin/nios2-gnutools/H-i686-pc-linux-gnu"
GCCDIR="$NIOS2_GNUTOOLS/lib/gcc/nios2-elf/3.4.6"

mkdir -p "$PREFIX/bin"
for TOOL in ar as nm objdump objcopy ranlib size strip ld
do
ln -sf $NIOS2_GNUTOOLS/bin/nios2-elf-$TOOL $PREFIX/bin/nios2-rtems-$TOOL
done

cat << EOT > $PREFIX/bin/nios2-rtems-gcc
#!/bin/sh
$NIOS2_GNUTOOLS/bin/nios2-elf-gcc -nostdinc -B "$PREFIX/nios2-rtems/lib/" \
-isystem "$PREFIX/nios2-rtems/include" -isystem "$GCCDIR/include" \
-msys-lib=c -msys-crt0="$PREFIX/nios2-rtems/lib/crt0.o" \
-DSYSTEM_BUS_WIDTH=32 -D__rtems__ -D__USE_INIT_FINI__ "\$@"
EOT

chmod +x $PREFIX/bin/nios2-rtems-gcc

ln -sf nios2-rtems-gcc $PREFIX/bin/nios2-rtems-cc
ln -sf nios2-rtems-gcc $PREFIX/bin/nios2-rtems-c++
```

A.2 RTEMS build shell script

```
#!/bin/bash

BASEDIR=$PWD

PREFIX=$BASEDIR/install # Install point

RTEMS=$BASEDIR/rtems-4.9.99 # Path to RTEMS sources

RECONFIG_RTEMS=true
REMAKE_RTEMS=true
REMAKE_BSP=false

BSP=nios2_iss
RTEMS_VARIANT=OPTIMIZE

# CONFIGURE RTEMS ##########################################

$RECONFIG_RTEMS && (
    rm -rf    "b-rtems"
    mkdir -p  "b-rtems"
    cd        "b-rtems"

    $RTEMS/configure \
--target=nios2-rtems \
--prefix=$PREFIX \
--disable-rdbg \
--disable-rtems-debug \
--disable-cxx \
--disable-itron \
--enable-posix \
--disable-tests \
--enable-multiprocessing \
--enable-multilib \
--disable-networking
    test $? -eq 0 || ( REMAKE_RTEMS=false; REMAKE_BSP=false )
    cd $BASEDIR
    )

# MAKE RTEMS ###############################################

$REMAKE_RTEMS && (
    cd "b-rtems"
    make RTEMS_BSP="" VARIANT="$RTEMS_VARIANT" all
    test $? -eq 0 && make RTEMS_BSP="" VARIANT="$RTEMS_VARIANT" install
    test $? -eq 0 || REMAKE_BSP=false
    cd $BASEDIR
    )
```

```
# MAKE BSP ##################################################

$REMAKE_BSP && (
    cd b-rtems
    make RTEMS_BSP=$BSP all
    test $? -eq 0 && make RTEMS_BSP=$BSP install
    cd $BASEDIR
    )
```

A.3 Application makefile

```
#
#  Makefile - RTEMS application
#
############################################################

NODE_NUMBER = 1 # For MP systems

BSP = nios2_iss

CSRCS = test.c

############################################################

RTEMS_MAKEFILE_PATH=/home/ruppe/deskstop/workdir/install/nios2-rtems/${BSP}

PGM=${ARCH}/rtems_${BSP}.exe

MANAGERS=all

include $(RTEMS_MAKEFILE_PATH)/Makefile.inc
include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg

AM_CPPFLAGS += -DNODE_NUMBER=${NODE_NUMBER}

COBJS = $(CSRCS:%.c=${ARCH}/%.o)
OBJS= $(COBJS) $(CXXOBJS) $(ASOBJS)

all: ${ARCH} $(PGM)

$(PGM): $(OBJS)
        $(make-exe)
```

Appendix B

Application sources

B.1 Simple example application

```
#include <rtems.h>
#include <stdio.h>

rtems_task Init( rtems_task_argument arg ){
printf("\n******************\n");
printf("* Hello world! *\n");
printf("******************\n");
while( 1 ){}
}

/* Configuration ******************************************/

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

B.2 MPCI implementation header file

```
/*
 * Multiprocessor Communications Interface Layer
 * File: mpci_main.h
 *
 * Author: Roger Dahlqvist (15/06/2009)
 *
 ******************************************************************************/

#ifndef __MPCI_MAIN_H__
#define __MPCI_MAIN_H__

#include <bsp.h>
#include <rtems.h>
#include <stdint.h>
#include <string.h>
#include <rtems/score/sysstate.h>
#include <mpci_mailbox.h>

typedef volatile uint8_t    vol_u8;
typedef volatile uint32_t   vol_u32;


/******************************************************************************
 * Configuration - These are the only settings you may need to change
 ******************************************************************************/

#define MPCI_SHARED_MEMORY_BASE (void *)  0x01004000  // Base address for shared memory
#define MPCI_MAILBOX_SIZE                 1024        // Mailbox size in bytes
#define MPCI_ENVELOPE_POOL_SIZE           1024        // Envelope pool size in bytes
#define MPCI_ENVELOPE_SIZE                128         // Envelope size in bytes
#define MPCI_NODES                        2           // Number of nodes (= master node)
#define MPCI_INTERRUPT                    1           // 1 = interrupt, 0 = polling using timer
#define MPCI_IRQ                          3           // IRQ if MPCI_INTERRUPT is set to 1


/******************************************************************************
 * Configuration - No editing required
 ******************************************************************************/

#define MPCI_NODE_NUMBER _Configuration_MP_table-> node
#define MPCI_ENVELOPE_PREFIX_SIZE 4 * sizeof(vol_u32)
#define MPCI_ENVELOPES MPCI_ENVELOPE_POOL_SIZE / MPCI_ENVELOPE_SIZE
#define MPCI_ENVELOPE_BASE MPCI_SHARED_MEMORY_BASE
#define MPCI_MAILBOX_BASE MPCI_SHARED_MEMORY_BASE + (MPCI_ENVELOPE_POOL_SIZE * MPCI_NODES)
#define MPCI_MAXIMUM_PACKET_SIZE MPCI_ENVELOPE_SIZE - MPCI_ENVELOPE_PREFIX_SIZE

#define MPCI_NO_ERROR      0
#define MPCI_NO_FREE_PKTS  0xf0000

#define MPCI_envelope_to_packet_ptr( env )((void *)(env)->packet)
#define MPCI_packet_to_envelope_ptr( pkt )((mpci_envelope *)((uint8_t *)(pkt)-MPCI_ENVELOPE_PREFIX_SIZE))


/******************************************************************************
 * Declarations
 ******************************************************************************/

rtems_mpci_entry mpci_initialization( void );
rtems_mpci_entry mpci_get_packet( rtems_packet_prefix ** packet );
rtems_mpci_entry mpci_return_packet( rtems_packet_prefix * packet );
rtems_mpci_entry mpci_send_packet( uint32_t node, rtems_packet_prefix * packet );
rtems_mpci_entry mpci_receive_packet( rtems_packet_prefix ** packet );

void mpci_intr_init( uint32_t node_number );
void mpci_intr_set( uint32_t node_number );
void mpci_intr_reset( uint32_t node_number );

extern rtems_mpci_table MPCI_table;
void * mpci_mailboxes [MPCI_NODES + 1];
```

```
#ifdef _MPCI_INIT

rtems_mpci_table MPCI_table = {
  100,                                  // default timeout value in ticks
  MPCI_MAXIMUM_PACKET_SIZE,             // maximum packet size
  mpci_initialization,                  // initialization procedure
  mpci_get_packet,                      // get packet procedure
  mpci_return_packet,                   // return packet procedure
  mpci_send_packet,                     // packet send procedure
  mpci_receive_packet                   // packet receive procedure
};

#endif

typedef struct {
  vol_u32        index;
  vol_u32        in_use;
  vol_u32        not_used1;
  vol_u32        not_used2;
  vol_u8         packet[MPCI_MAXIMUM_PACKET_SIZE];
} mpci_envelope;

struct pkt_cpy {
  uint32_t       packet[MPCI_MAXIMUM_PACKET_SIZE/4];
};

#endif /* __MPCI_MAIN_H__ */
```

B.3 MPCI Mailbox header file

```
/*
 * Multiprocessor Communications Interface Layer
 * File: mpci_mailbox.h
 *
 * Author: Roger Dahlqvist (15/06/2009)
 *
 ******************************************************************************/

#ifndef __MPCI_MAILBOX_H__
#define __MPCI_MAILBOX_H__

#include <stdint.h>

void mpci_mailbox_init( void * base, uint32_t size );
void mpci_mailbox_lock( void * base );
void mpci_mailbox_release( void * base );
uint32_t mpci_mailbox_post( void * base, uint32_t msg, uint32_t target_node );
uint32_t mpci_mailbox_get( void * base, uint8_t * error );
uint32_t mpci_mailbox_messages( void * base );

typedef struct {

  uint32_t owner;
  uint32_t value;
  uint32_t slots;
  uint32_t used;
  uint32_t head;
  uint32_t tail;

} mailbox_control;

#endif /* __MPCI_MAILBOX_H__ */
```

Appendix C

Acronyms

MPCI Multiprocessor Communications Interface

RTEMS Real Time Executive for Multiprocessor Systems

RTOS Real Time Operating System

FSF Free Software Foundation

FPGA Field Programmable Gate Array

BSP Board Support Package

MIMD Multiple Instruction stream, Multiple Data stream