TECHNICAL REPORTS IN COMPUTER SCIENCE

Technische Universität Dortmund

R^2G: Supporting POSIX like semantics in a distributed RTEMS system

Andreas Heinig

Computer Science 12 – Design Automation of Embedded Systems Group

Number: 836
December 2010

Technische Universität Dortmund — Fakultät für Informatik
Otto-Hahn-Str. 14, 44227 Dortmund

http://ls12-www.cs.tu-dortmund.de

http://www.andreasheinig.de/permalink/10121

Andreas Heinig: *R^2G: Supporting POSIX like semantics in a distributed RTEMS system,* Technical Report, Department of Computer Science, Dortmund University of Technology. © December 2010

ABSTRACT

R^2G (speak: R square G) is an extension to the open source RTEMS real-time operating system. The purpose of R^2G is to remove the limitations of RTEMS in the context of multi-threaded applications and to support IMEC's RTLib, which implements the parallelization of the MNEMEE Tool Flow.

R^2G establishes the connection between several tools. The parallelized source code produced either by MPMH (IMEC) or by the MNEMME Tool Flow (MPMH + ICD-C + TGE) can now be executed on several simulators including MPARM and CoMET. Due to the high simulation speed of CoMET, large benchmark applications can be executed. We implemented two new CoMET based platforms with a flat respectively hierarchical memory layout. On the hierarchical platform, memory optimization and mapping tools can fully exploit their optimizations.

ACKNOWLEDGMENTS

This work was fully supported by the Artist Design Network of Excellence EU FP7 Grant No. 39316.

CONTENTS

1 INTRODUCTION

In the last decades multi-core systems were not widely used in the embedded systems community. In contrast to high performance computing, where a very long history of parallel computing exists, most embedded applications are written sequentially.

However, there is a continuing demand for higher performance of information processing. Due to limitations of increasing clock frequencies further, this growing demand stimulates using of parallelism including multiple processors in the embedded world, too. Thus, one of the designers tasks is to provide parallelized code of the application. Parallelizing a sequential application or even writing a parallel application from scratch is not a trivial task. Explicit synchronization has to be added to avoid hazards which will occur when multiple processes access the same data concurrently.

Furthermore, hardware platforms, containing connected processors, are becoming increasingly parallel. This trend also affects the design of embedded systems. Actually, there are various kinds of connectivity. In multi-processors in a system on a chip (MPSoC), processors are tightly connected and communication is fast. In other cases, networked processors may be less tightly connected and communication may be slower. Hence, the another task for a designer will be to address the issues resulting from the use of multiple processors, in particular in the form of multiple heterogeneous processors on a chip, also containing memory hierarchies. The designer has to map the parallelized application onto the platform while respecting the memory hierarchy. In some cases, not every memory is accessible from each processor. Usually, the outermost memories, like DRAM cells, are shared between all processing units, but they are slow; whereas, the innermost memories, like SRAM used as scratch pad, are only exclusively usable by one processor, but they are very fast. Hence, the designer has not only to map the application onto a processor but also to map the data onto the right location to get the maximum performance.

If we consider complex embedded applications which process multimedia and communication workloads, it will be increasingly impossible for designers to map those applications cost-efficiently to any platform, without significant optimization of the initial source code. At this point the MNEMEE[1] project comes into play. MNEMEE addresses those key challenges by introducing an innovative tool flow which optimizes the application on the source code level. Most of the tools are running fully automatically. In various publications [2][3][4][5] the authors have shown large amounts of speedup improvements, memory footprint reductions, and energy savings.

In this work – supported by ArtistDesign[6] – we will concentrate on the underlying hardware platform and operating system software which were used in the MNEMEE tool flow, too.

The big challenge was to provide a POSIX like API which is functional across processor boundaries and within a hierarchical memory layout. In this way, the library (namely RTLib[7]) which implements the parallelization, synchronization and communication can interact with the platform in the same way as on a normal host platform (running Linux for example).

The MNEMEE project benefits very much from this work. It enables the possibility to measure the results obtained by the individual tools on an embedded platform. Previously, only high level simulations and tests based on the host platform were used.

To test the functionality of our solution, a H.264 encoder application is used. H.264 is a very compute intensive video compression algorithm. A parallelized version of H.264 requires much communication between the several threads. Hence, it is the ideal test case for our system.

This report is organized as follows: In chapter 2 the platforms are depicted. In chapter 3 and chapter 4 we go bottom-up through the software stack. Chapter 5 concludes this report. Finally, in Appendix A the R^{2}G API is shown in detail.

2 PLATFORMS

In this work we will concentrate on simulator based platforms. Simulation has the advantage that no hardware is needed and that we can exchange components very easily. Another important point is that we can precisely measure the clock cycles and cache behavior of the applications. Furthermore, energy models included in the simulators can be used to compare the optimizations regarding energy efficiency.

The drawback of simulation is the typically slow speed compared with real hardware. Especially for the MPARM[8] simulator this let to huge problems. Here we have to interrupt the execution of our MPEG4 encoding benchmark after three days. Hence, we used a second simulator called CoMET[9] which simulates the benchmark within a half day.

Both simulators and platforms are described in the next two sections.

2.1 MPARM

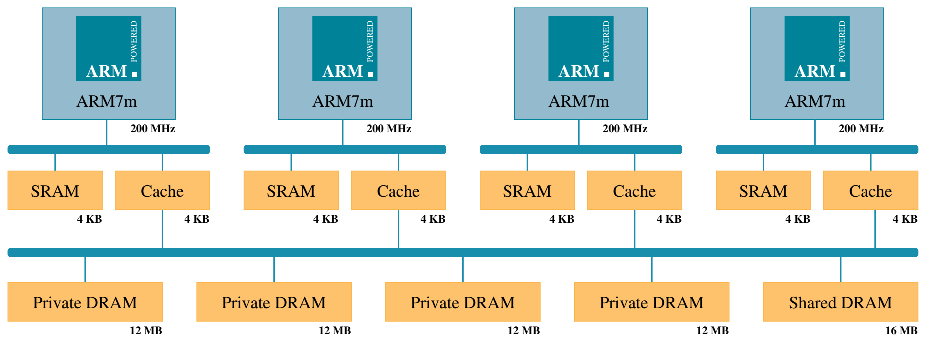


Figure 1: MPARM Platform

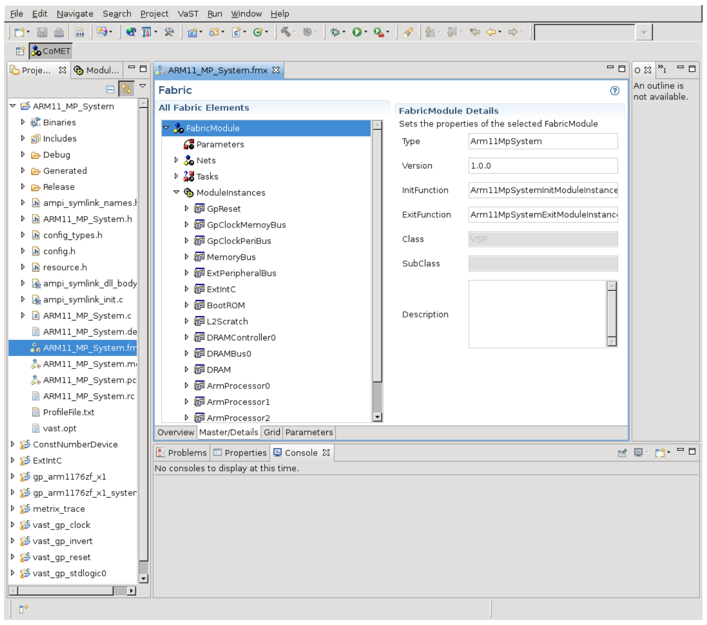
MPARM[8] was developed at the University of Bologna. It is a multi-processor cycle accurate architectural simulator. Several SWARM[10] (SoftWare ARM) processor cores implemented in C++ are connected by an AMBA bus implemented in System C. The SWARM cores simulate the ARM7m processor.

SRAM based memory and caches are connected with the ARM core through an internal bus. The SRAM is typically used as a scratch pad. If enabled, the cache would store accesses to the private memory which is attached to a shared AMBA like interconnect.

Except for the size of the shared memory, we used the default configuration of the simulator. We had to increase the amount of shared memory form 1 MB to 16 MB to enable storage of the frame buffers of our benchmark application. The processors are simulated with the default frequency of 200 MHz. A detailed view is depicted in Figure 1.

With this setup we were very successful in simulating smaller applications. However, the simulation speed of MPARM is very low. Hence, larger applications are not running very well on MPARM. We searched for an alternative solution and found the CoMET simulator of Synopsys, which is already used at our facility to simulate TriCore[11] based systems. CoMET is described in the next subsection.

2.2 COMET



FabricModule Details

Sets the properties of the selected FabricModule

Type	Arm11MpSystem
Version	1.0.0
InitFunction	Arm11MpSystemInitModuleInstance
ExitFunction	Arm11MpSystemExitModuleInstanc
Class	VSP
SubClass	
Description	

An outline is not available.

No consoles to display at this time.

Figure 2: CoMET GUI Embedded in Eclipse

2.2.1 *About CoMET*

CoMET[9] is a system engineering tool that enables the creation of a software simulation-based VSP (virtual system prototype) of an embedded system or a system-on-a-chip. CoMET is used to design, simulate, analyze, and optimize complex embedded systems and to quantitatively evaluate performance while running real software applications. The VSP simulation is fast enough to enable architects to evaluate real software loads, including real-time operating systems, protocol stacks and large software libraries.

CoMET has a high degree of timing accuracy so that the virtual embedded system simulation reflects the actual behavior of even the most demanding real-time systems. Due to fast and accurate cycle simulation, device drivers and other code are enabled to run interacting with the hardware in a normal way. Parameters such as cache size can also be evaluated and optimized, which is impossible in a less accurate environment where cache hits and misses are not modeled precisely.

2.2.2 *Platform*

CoMET supports different virtual processor models (VPM) ranging from micro-controllers (e.g. NEC V850) up to general purpose CPUs (e.g. ARM 7 / 926 / 11). Also other components typically used in real-world systems are supported, for example: clocks, timers, DMA controllers, general purpose memory, etc. .

Due to CoMET's modular design, any of those modules can be connected with each other. Therefore, CoMET provides primitives like simple wires with the typical logic model as known by VHDL up to complete bus models like AMBA, PCI or PLB. To select and connect the components, a system architect can model the VSP within a GUI embedded in the Eclipse[12] framework (see Figure 2). To start simulation the GUI is not necessary anymore.

The two platforms described in the next subsections use the following components:

- **Processor**: ARM1176

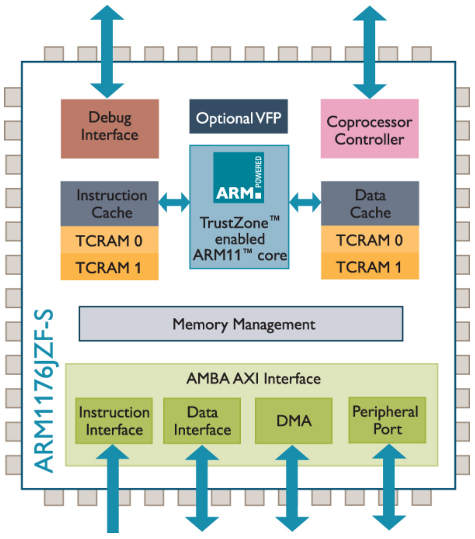


Figure 3: ARM 1176 CPU Block Diagram [13]

Figure 3 shows the ARM 1176 processor. The processor uses the Harvard architecture. Thus, we have separate caches for instruction and data. Each cache stores 16 kB. Beside the caches, the processor has Tightly Coupled Memories (TCM) which can be used as scratch pads. One TCM RAM is 8 kB large. Last but not least, the AMBA AXI Interface is used for off-chip connection.

We made very good experiences when using the ARM architecture within MPARM. Thus, we took the decision to use ARM for the CoMET based platforms, too.

- **Memories**:

  The platform contains several memories:

  - *Private Memory*

    Every processor has a dedicated memory or reserved memory part for its exclusive usage. The data and code segments are mapped to this memory. The heap and non-shared operating system data reside here, too.

    The private memory is the only memory which is cached when caches are enabled.

  - *Shared Memory*

    The shared memory is under complete application control.

  - *L2 Scratch Pad*

    Shared data of the operating system and system libraries are mapped to the L2 scratch pad memory. Applications can use this memory with special linker script adaptations or by dynamic allocations with the function r2g_mem_alloc().

  - *Boot ROM / Flash*

    With this memory we simulate a boot medium which might be a ROM or a flash. Every processor will start execution on the base address of this memory and loads a special boot loader. The loader extracts the executable from the boot image and initially sets the processor up. For the creation of this boot image we provide the tool `mkbootrom` which generates the image of the ELF files for each processor.

- **Peripherals**:

  Every processor has dedicated peripheral units. The only exception is the External Interrupt Controller which is shared between all processors:

  - *UART*

    CoMET provides a module for a serial interface. All outputs of the C-Library are redirected to the UART.

  - *DMA Controller*

Each processor has a DMA controller to copy large amounts of data. The data has to be addressable by the processor. Hence, data residing in the private memory of another processor is not accessible.

- *Timer Controller*

  The time base is maintained through this timer chip.

- *Clock Controllers*

  We used many clock controllers to clock most of all components.

- *Interrupt Controller*

  ARM processors only have one external interrupt, called exception. To support multiple interrupt sources an extra chip is needed. This chip multiplexes the incoming interrupt wires.

- *Processor ID ROM*

  The Processor ID ROM was developed at TU Dortmund.

  In a multiprocessor system we need an unique identifier for each CPU. This device provides an arbitrary constant number.

- *External Interrupt Controller*

  The External Interrupt Controller was developed at TU Dortmund.

  Every processor can send interrupts to other processors. This is important for the operating system to signal that new data has arrived in the shared data structures. If there were no interrupts, polling would be necessary. However, according to our measurements, the polling mode reduces the overall performance at least by a factor of two.

In the next two subsections the connection between the processors and the memories are depicted.

2.2.2.1 *ARM1176 – Flat Memory Model*

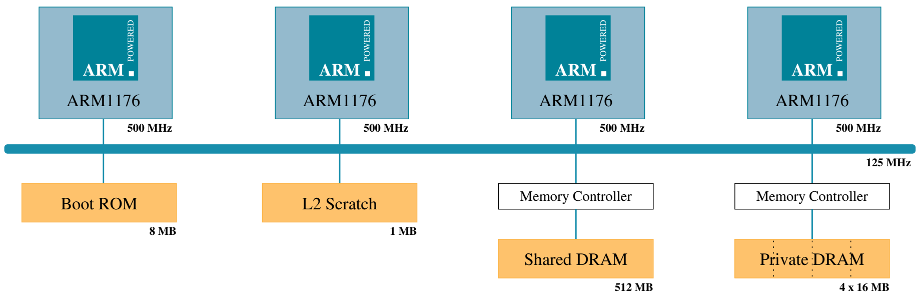


Figure 4: Flat Platform

Figure 4 shows the flat platform. We used four ARM1176 processors. One bus connects all processors and memories. The partitioning of the private memory is done in software. Here, the page table entries block any access to a foreign region. The private memory and the shared memory are two separate modules. With this design we can maintain identical memory mappings for the flat as well as the hierarchical platform. Indeed, which platform is used, is completely transparent for the whole software stack. Hence, we can use both platforms with exactly the same software environment.

2.2.2.2 *ARM1176 – Hierarchical Memory Model*

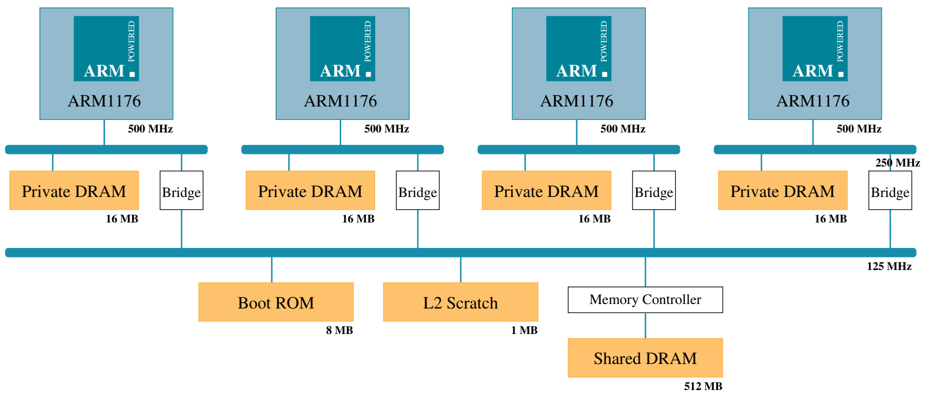


Figure 5: Hierarchical Platform

Figure 5 shows the hierarchical platform. In contrast to the flat platform, every processor has now its own dedicated memory which is connected through an internal bus. This setup lowers the bus contention dramatically.

The hierarchical platform is not directly comparable to an industrial platform. However, many components of this platform can be found in other systems, as well. The Cell Broadband Engine Architecture (CellBE)[14], for example, is also equipped with local on-chip memories. The main purpose of providing such a more or less scientific platform is to show, that it may be beneficial to deal with such hierarchical systems in the future, if powerful memory optimization approaches are available in upcoming compilers.

3 RTEMS

At the beginning of our work we used the MPARM simulator as our platform. We have chosen RTEMS as operating system, because it was the only operating system that was already ported to MPARM. However, in the later progress we developed our own RTEMS port, because the one provided by the University of Bologna only supports RTEMS 4.6. The software developed within this project is based on RTEMS 4.10.

The Real-Time Executive for Multiprocessor Systems or RTEMS is a full featured real-time operating system. It supports a variety of open API and interface standards including POSIX-1003.1b.

3.1 RTEMS ARCHITECTURE

In figure 6 the RTEMS architecture is depicted. Most parts are completely independent regarding the used processor and platform.

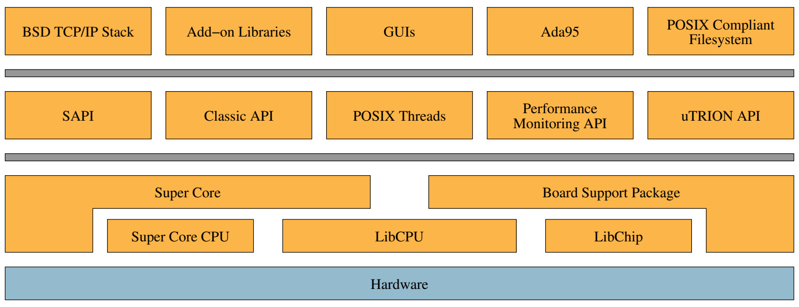


Figure 6: RTEMS Architecture

The adaptation for specific platform is realized within four components:

- **Super Core CPU**

  The major tasks of this component are to implement context switches and low level interrupt handling.

- **LibCPU**

  For some CPUs cache and MMU management are implemented here.

- **LibCHIP**

LibCHIP contains drivers for some network, timer, and serial controllers.

- **Board Support Package**

  The Board Support Package (BSP) provides an abstraction of the currently used platform. Therefore, LibCPU and LibCHIP can be used for already implemented components.

  BSPs provide the following functionality:

  – Booting until the RTEMS entry routine

  – Setup and handle interrupts

  – Initialize timers

  – Provide shared memory and shared locking mechanisms for the multi processor communication

  – Initialize other hardware components such as the UART controller for example

3.2 BSP - Board Support Package

We have developed BSP's for the MPARM based as well as the CoMET based platforms. In addition to the tasks described above, we added extra functionality called "R^{2}G Support" (cf. A.4 on page 50). The most important feature here, is the spinlock based synchronization primitive which works across processor boundaries. In difference to other BSP's this primitive can also be used by the upper software layers and not only by the RTEMS multi processor controller.

Another important feature is the implementation of different hooks for the application. The "`init_mem`" hook for example can be used to initialize additional memories like the scratch pads or the shared memory. Especially on the CoMET platforms, it is not possible to initialize the scratch pads during boot-up, because they have to be explicitly mapped by the BSP into the address space before usage. Hence, the boot loader cannot directly load the contents from the ELF file into the scratch pad.

3.3 What we have achieved so far (and what not)

At this point let us summarize the last two chapters; For the simulation of our software two platforms have been made available. The platforms are fully supported by RTEMS even in multiprocessor mode. And, last but not least, a POSIX interface is provided by RTEMS.

Now, everything seems to be ready: We can take an application and parallelize it on the source code level with the tools[2][15][7] provided by ICD Dortmund[16] and IMEC[17]. Thereafter, we can use IMEC's POSIX based Run-Time Library (RTLib[7]) which implements the parallelization. And finally we should be able to simulate this parallelized application on either MPARM or CoMET.

However, there is a huge limitation which prevents the successful simulation; The POSIX implementation of RTEMS is only working on a per-processor basis. This means we cannot spawn threads on an remote processor and we cannot synchronize with an remotely running thread.

To bridge this gap R^{2}G was developed which is described in the remainder of this report.

4 R^{2}G

R^{2}G is the acronym for "**R**TEMS and **R**TLib **G**lued together". The goal of R^{2}G is to eliminate the limitations of RTEMS in order to execute auto-parallelized code which uses IMEC's RTLib.

4.1 OVERVIEW

In figure 7 a system overview is given:

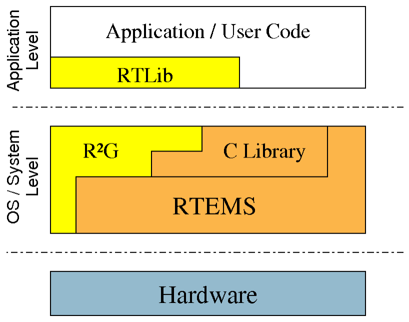


Figure 7: System overview

On the Application Level we can find the application which uses a special version of RTLib. This RTLib version does not use the `pthread` interface anymore. Instead R^{2}G is used for synchronization and thread management.

Inside the OS/System Level the RTEMS OS, C Library (in our case "newlib"), and R^{2}G can be found. R^{2}G can only be used with special RTEMS BSPs which support R^{2}G extensions (see 3.2 on page 10 and A.4 on page 50).

In the next section it will be shown why R^{2}G is necessary.

4.2 REMOTE THREAD SPAWNING

In our research we use normal single threaded applications which should be (semi-) auto-parallelized and optimized using tools provided by MNEMEE[1]. For the parallelization of the application the MPMH-Tool includes a function call to `configComponent()` inside the main() routine of the application. The `configComponent()` function spawns all the threads and initializes the communication primitives, e.g. FIFOs.

If we would now use the pthread interface, all functions would spawn on the same processor. To avoid this, R^{2}G extends the r2g_thread_create() function (cf. A.1.1.6 on page 19) with an additional parameter which specifies the target CPU for thread creation. For the mapping of threads to processors the MNEMEE tool flow provides tools, too.

Other typical problems of embedded systems are the available amount of main memory, and sometimes the heterogeneous structure. To cope with such problems, RTEMS runs completely independent on each processor with only the necessary application parts linked into the binary. This enables a mixture of even different processor architectures on one platform.

However, this is a big problem when a remote thread should be spawned. In Figure 8 a typically program is shown which runs on a shared memory system, e.g. the host:

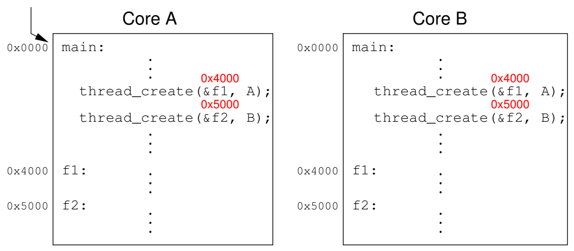


Figure 8: Identical Program Code On Each Node

Execution is started on processor A within `main()`. Now A creates two threads: `f1()` on A resp. `f2` on B. Due to the same memory layout on both processors, the thread entry points are equal.

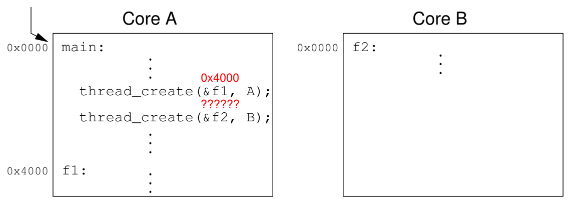
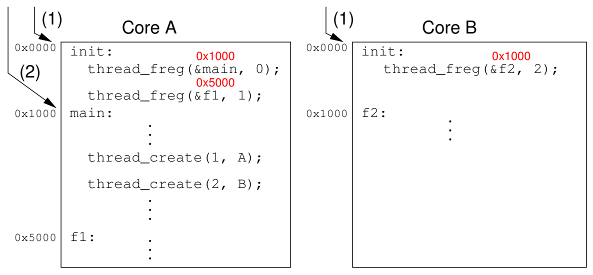


Figure 9: Different Program Code

If we have different architectures or a different memory layout, the linker that links the binary for A will not be able to determine the thread entry points in B and vice versa (cf. Figure 9).

The solution for this problem can be found in Figure 10:



```
init:                       0x1000
  thread_freg(&main, 0);
                            0x5000
  thread_freg(&f1, 1);
main:
  thread_create(1, A);
  thread_create(2, B);
f1:
```

```
init:                       0x1000
  thread_freg(&f2, 2);
f2:
```

Figure 10: R²G Solution: Function Identifiers

In the first step (arrow marked (1)), R²G calls `init()` on every processor. Within this function all possible thread entry points have to be assigned to unique identifiers called **fid** (Function IDentifier). The assignment of a fid to a function is implemented by `r2g_thread_freg()`(cf. A.1.1.11 on page 21). Although not mandatory, it is a good practice to use identical identifiers for the same thread functions distributed on different processors to better keep track of the application, especially for later debugging. The fid zero has a special meaning for R²G: It stands for the main() routine. In the second step, R²G creates a thread for every function with fid equal to zero. Hence, the mapping of `main()` to any arbitrary processor is possible.

4.3 SYNCHRONIZATION

Another problem when using the RTEMS pthread implementation was the missing synchronization of threads running on different processors. Here, R²G implements a subset of the POSIX synchronization primitives Mutex, Condition Variables, and Barriers. Detailed information for each primitive can be found in A.3 on page 35.

4.4 RTLIB PORTING

Another important part of the R²G project was to port RTLib to the new interface. Due to the R²G design, this job was mostly straightforward. The majority of the R²G API implements POSIX like semantics. Therefore, it was sufficient to replace "`pthread_`" with "`r2g_`". To avoid conflicts with the POSIX interface of RTEMS, we do not use the original POSIX names for the functions.

While porting RTLib special attention was paid on the internal data structures. Whenever possible, we locate the data in the private memory. Shared data are statically mapped into the shared memory. All data areas unknown at compile time are allocated with `r2g_mem_alloc()` (cf. A.2.1.1 on page 32) during runtime.

To support our heterogeneous platforms we had to change the RTLib API:

- `mps_config_thread`

  This function gets an additional parameter to specify the mapping of the thread to a processor. The thread entry function parameter was substituted by the fid.

- `mps_config_freg`

  We added this additional API call as a wrapper around `r2g_thread_freg()`. The difference is that this new function takes an RTLib thread entry function as parameter and not a POSIX style thread entry function.

- `app_register_functions`

  This function has to be additionally implemented for the application. It has to exist exactly once on each processor. Basically, this function is a wrapper for the `init()` routine described in the last section. Here, the application has to register the thread entry functions with `mps_config_freg`.

Applications parallelized with the MNEMEE tool flow have not to deal with R^{2}G directly. Instead, a modified RTLib (aka. R^{2}G-RTLib) has to be used. Detailed instructions on how to port an application to the new R^{2}G-RTLib API can be found in [18].

5 CONCLUSION

In this technical report we have presented R^{2}G which is an intermediate layer between the application and the operating system. By introducing function identifiers and additional mapping parameters to the thread creation routine, we managed to spawn threads on different CPUs. The whole R^{2}G API is mostly conforming to POSIX semantics (the exeptions are depicted in Appendix A). In this way we can provide a POSIX like environment even in future heterogeneous systems.

Especially, the MNEMME project has benefited very much of this work. The tools, which were designed to support POSIX interfaces, are now able to produce code for our embedded platforms, too.

With the actual setup we were able to parallelize our MPEG4 encoder benchmark application fully automatically with the MNEMEE tools. Thereafter, we were able to simulate the transformed code with only slight modifications.

The R^{2}G project is completely finished and in a stable state. Thus, future work will only concern the integration of further platforms into the RTEMS operating system.

A APPENDIX: R²G API

A.1 THREAD SUBSYSTEM

The R²G thread API enables the spawning of threads and the usage of different thread operations across processor core boundaries.

Therefore, the interface differs from POSIX. Pointers to the thread entry function are not used anymore. Instead, an application needs to register any thread entry point explicitly. This means to couple the entry point with an unique identifier via the function r2g_thread_freg() on the CPU/CPUs where the thread will be executed.

A.1.1 *Thread Operations*

A.1.1.1 *int __r2g_thread_create (***r2g_thread_t** ∗ thread*, const* **r2g_thread_attr_t** ∗ attr*, r2g_threadfunc_t* ∗ start_routine*, void* ∗ arg*)*

Create a thread on the local cpu.

Parameters:

thread thread identifier object

attr thread attribute object

start_routine thread entry point

arg thread argument pointer

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid cpu and/or fid

ENOMEM No memory to allocate thread structure

EAGAIN Insufficient resources to create thread or system limit reached

This function implements low-level thread creation on the local node. However, you can call it in your application, too. In this way you will create a thread on the same node in the POSIX way.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_create).

A.1.1.2 *r2g_sighandler_t r2g_signal (int* signum*,* *r2g_sighandler_t* handler*)*

ANSI C signal handling.

Parameters:

signum	Signal number
handler	Signal handler routine

Returns:

Returns the previously installed signal handler.

This functions establishes the signal handler *handler* for the signal *signum*.

Remarks:

Conforming to ANSI C (signal).

A.1.1.3 *int r2g_thread_cancel (***r2g_thread_t** thread*)*

Cancel execution of a thread.

Parameters:

thread	thread identifier

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

ESRCH	No thread with the ID *thread* could be found
EINVAL	Invalid request

This function sends a cancellation request to the thread identified by *thread*. Whether and when the target thread reacts to the cancellation request depends on the actual status of the thread.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cancel).

A.1.1.4 *void r2g_thread_cleanup_pop (int* execute*)*

Remove routine from top of thread cleanup stack.

Parameters:

execute	Execute routine while popping

This function removes the routine at the top of the stack of clean-up handlers, and optionally executes it if *execute* is non-zero.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cleanup_pop).

A.1.1.5 *void r2g_thread_cleanup_push (void(∗)(void ∗)* routine*, void ∗* arg*)*

Push cleanup handler to cleanup stack.

Parameters:

routine handler to execute when r2g_thread_exit() or r2g_thread_cleanup_-pop()

arg argument for handler routine

This function pushes *routine* onto the top of the stack of clean-up handlers. When routine is later invoked, it will be given *arg* as its argument.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cleanup_push).

A.1.1.6 *int r2g_thread_create (***r2g_thread_t** *∗* thread*, const* **r2g_thread_attr_t** *∗* attr*, int* fid*, void ∗* arg*, unsigned int* cpu*)*

Create a thread.

Parameters:

thread thread identifier object

attr thread attribute object

fid function identifier of thread entry point

arg thread argument pointer

cpu cpu on which the thread will be created

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid cpu and/or fid

ENOMEM No memory to allocate thread structure

EAGAIN Insufficient resources to create thread or system limit reached

This function starts a new thread. The new thread starts execution by invoking the routine represented by *fid* on the cpu *cpu*. The start routine gets *arg* passed as the sole argument.

The new thread terminates in one of the following ways:

- it calls r2g_thread_exit(), specifying an exit status value that is available to another thread that calls r2g_thread_join().
- it returns from the start routine. This is equivalent to calling r2g_thread_-exit() with the value supplied in the return statement.
- it is canceled (see r2g_thread_cancel()).
- any of the threads calls exit(), or the main thread performs a return from main(). This causes the termination of all threads.

The *attr* argument points to a r2g_thread_attr_t structure whose contents are used at thread creation time to determine attributes for the new thread. this structure is initialized using r2g_thread_attr_init() and related functions. If *attr* is NULL, then the thread is created with default attributes.

Remarks:

> **Not** conforming to the POSIX 1003.1-2001 pthread_create() function. The R^{2}G implementation has an additional argument to specify the CPU where to execute the thread. Also the function identifier is used as thread entry and not the function pointer.

A.1.1.7 *int r2g_thread_detach (***r2g_thread_t** thread*)*

Detach a thread.

Parameters:

> *thread* thread identifier

Returns:

> On success, 0 is returned; on error, an error number is returned.

Return values:

> *EINVAL* Thread is not joinable

This function marks the thread identified by *thread* as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Remarks:

> Conforming to POSIX 1003.1-2001 (pthread_detach).

A.1.1.8 *int r2g_thread_equal (***r2g_thread_t** t1*,* **r2g_thread_t** t2*)*

Compare threads identifiers.

Parameters:

> *t1* thread identifier of first thread
>
> *t2* thread identifier of second thread

Returns:

> If the two threads are equal, r2g_thread_equal() returns a non-zero value; otherwise, it returns 0.

This function compares two thread identifiers.

Remarks:

> Conforming to POSIX 1003.1-2001 (pthread_equal).

A.1.1.9 *void r2g_thread_exit (void* ∗ value_ptr*)*

Terminate Thread execution.

Parameters:

value_ptr Return value of thread

Returns:

This function does not return to the caller.

The r2g_thread_exit() function terminates the calling thread and returns a value via *retval* that (if the thread is joinable) is available to another thread in the same process that calls r2g_thread_join().

Any clean-up handlers established by r2g_thread_cleanup_push() that have not yet been popped, are popped (in the reverse of the order in which they were pushed) and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in an unspecified order.

When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released, and functions registered using atexit() are not called.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_exit).

A.1.1.10 *r2g_threadfunc_t*∗ *r2g_thread_fget (int* fid*)*

Get Address of function.

Parameters:

fid function identifier

Returns:

On success, the address of the routine registered by r2g_thread_freg() is returned; on error, NULL is returned.

This function returns the address of **routine** represented by the function identifier *fid*. If no function is registered NULL is returned.

Remarks:

This function is a R^{2}G specific extension.

A.1.1.11 *int r2g_thread_freg (r2g_threadfunc_t* routine*, int* fid*)*

Add function to call table.

Parameters:

routine entry of thread function

fid function identifier

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL invalid function identifier

This functions registers the procedure *routine* under the function identifier *fid*. The pair (*routine*, *fid*) is only valid for the calling CPU. To use the *fid* also on another CPU (i.e. for creating threads), r2g_thread_freg() has to be called on that CPU, too.

Function identifiers are used in thread creation (r2g_thread_create()) and for destructors (r2g_key_create()) of thread specific keys.

Note:

The *routine* represented by *fid* can be overwritten at any time by another call to r2g_thread_freg() on the same CPU.

Warning:

The *fid* **zero** is used to identify main(). Other functions should not use this identifier. After startup, R^{2}G will search and execute any function with *fid* = 0.

Remarks:

This function is a R^{2}G specific extension.

A.1.1.12 *int r2g_thread_join (***r2g_thread_t** thread*, void* ∗∗ value_ptr*)*

Wait for thread termination.

Parameters:

thread thread identifier

value_ptr Return pointer of thread

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Thread is not joinable

This function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then r2g_thread_join() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then r2g_thread_join() copies the exit status of the target thread (the value that the target thread supplied to r2g_thread_exit() or via return) into the location pointed to by ∗*retval*. If the target thread was canceled, then R2G_THREAD_CANCELED is placed in ∗*retval*.

After a successful call to r2g_thread_join(), the caller is guaranteed that the target thread has terminated.

Warning:

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling r2g_thread_join() is canceled, then the target thread will remain joinable
Joining with a thread that has previously been joined results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_join).

A.1.1.13 *int r2g_thread_kill (***r2g_thread_t** thread*,* *int* sig*)*

Send a signal to a thread.

Parameters:

thread thread identifier

sig signal to deliver

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

ESRCH No thread with the ID *thread* could be found

EINVAL An invalid signal was specified

This function sends the signal *sig* to the thread identified by *thread*. The signal is asynchronously directed to thread.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_kill).

A.1.1.14 **r2g_thread_t** *r2g_thread_self (void)*

Get thread structure of running thread.

Returns:

This function always succeeds, returning the calling thread's ID.

This function returns the identifier of the calling thread. This is the same value that is returned in ∗**thread** in the r2g_thread_create() call that created this thread.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_self).

A.1.1.15 *int r2g_thread_yield (void)*

Yield the processor.

Returns:

On success, 0 is returned; on error, an error number is returned.

This function causes the calling thread to relinquish the CPU. The thread is placed at the end of the run queue for its static priority and another thread is scheduled to run.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_yield).

A.1.2 *Thread Attributes*

A.1.2.1 *int r2g_thread_attr_destroy (***r2g_thread_attr_t** $*$ attr*)*

Destroy thread attribute object.

Parameters:

attr thread attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function destroys the thread attributes object pointed to by *attr*. Destroying a thread attributes object has no effect on threads that were created using that object.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_destroy).

A.1.2.2 *int r2g_thread_attr_getdetachstate (const* **r2g_thread_attr_t** $*$ attr*, int* $*$ detachstate*)*

Get detach state attribute in thread attribute object.

Parameters:

attr thread attribute object

detachstate buffer for detach state

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the detach state attribute of the thread attribute object *attr* in the buffer pointed to by *detachstate*.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_getdetachstate).

A.1.2.3 *int r2g_thread_attr_getflags (const* **r2g_thread_attr_t** ∗ attr*, int* ∗ flags*)*

Get flags in thread attribute object.

Parameters:

attr thread attribute object

flags integer buffer for flags

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the flags of the thread attribute object *attr* in the buffer pointed to by *flags*.

Remarks:

This function is a R^{2}G specific extension.

A.1.2.4 *int r2g_thread_attr_getschedpriority (const* **r2g_thread_attr_t** ∗ attr*, int* ∗ schedpriority*)*

Get scheduling priority in thread attribute object.

Parameters:

attr thread attribute object

schedpriority buffer for scheduling priority

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the scheduling priority attribute of the thread attribute object *attr* in the buffer pointed to by *schedpriority*.

Remarks:

This function is a R^{2}G specific extension.

A.1.2.5 *int r2g_thread_attr_getstackaddr (const* **r2g_thread_attr_t** ∗ attr*, void* ∗∗ stackaddr*)*

Get stack address attribute in thread attribute object.

Parameters:

attr thread attribute object

stackaddr stack address buffer

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the stack address attribute of the thread attribute object *attr* in the buffer pointed to by *stackaddr*.

Warning:

This function is currently not supported in R^{2}G!

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_getstackaddr).

A.1.2.6 *int r2g_thread_attr_getstacksize (const* **r2g_thread_attr_t** $*$ attr, *size_t* $*$ stacksize*)*

Get stack size attribute in thread attribute object.

Parameters:

attr thread attribute object

stacksize buffer for stack size

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the stack size attribute of the thread attribute object *attr* in the buffer pointed to by *stacksize*.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_getstacksize).

A.1.2.7 *int r2g_thread_attr_init (***r2g_thread_attr_t** $*$ attr*)*

Initialize thread attribute object.

Parameters:

attr thread attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function initializes the thread attributes object pointed to by *attr* with default attribute values. After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more r2g_thread_create() calls that create threads.

When a thread attributes object is no longer required, it should be destroyed using the r2g_thread_attr_destroy() function.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_init).

A.1.2.8 *int r2g_thread_attr_setdetachstate (***r2g_thread_attr_t** ∗ attr*, int* detachstate*)*

Set detach state attribute in thread attribute object.

Parameters:

attr thread attribute object

detachstate detach state

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function sets the detach state attribute of the thread attribute object referred to by *attr* to the value specified in *detachstate*. The detach state attribute determines whether a thread created using the thread attributes object *attr* will be created in a joinable or a detached state.

The following values may be specified in *detachstate:*

- R2G_THREAD_CREATE_JOINABLE

  Create thread in joinable state

- R2G_THREAD_CREATE_DETACHED

  Create thread in detached state

Default: R2G_THREAD_CREATE_JOINABLE is set.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_setdetachstate).

A.1.2.9 *int r2g_thread_attr_setflags (***r2g_thread_attr_t** ∗ attr*, int* flags*, int* mask*)*

Sets or clears flags in thread attribute object.

Parameters:

attr thread attribute object

flags flag value

mask flag mask

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function sets or clears flags of the thread attributes object referred to by *attr* to the value specified in *flags* masked by *mask*.

The following flags/masks are defiend:

R2G_THREAD_FPU – Thread uses the floating point unit (masked by R2G_-THREAD_FPU_MASK)

Default: R2G_THREAD_FPU is set.

Remarks:

This function is a R²G specific extension.

A.1.2.10 *int r2g_thread_attr_setschedpriority (***r2g_thread_attr_t** ∗ attr*, int* schedpriority*)*

Set scheduling priority in thread attribute object.

Parameters:

attr thread attribute object

schedpriority scheduling priority value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function sets the scheduling priority attribute of the thread attribute object *attr* to the value specified in *schedpriority*.

Remarks:

This function is a R²G specific extension.

A.1.2.11 *int r2g_thread_attr_setstackaddr (***r2g_thread_attr_t** ∗ attr*, void* ∗ stackaddr*)*

Set stack address attribute in thread attribute object.

Parameters:

attr thread attribute object

stackaddr stack address

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function sets the stack address attribute of the thread attribute object *attr* to the value specified in *stackaddr*.

Warning:

This function is currently not supported in R^{2}G!

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_setstackaddr).

A.1.2.12 *int r2g_thread_attr_setstacksize (***r2g_thread_attr_t** $*$ attr*, size_t* stacksize*)*

Set stack size attribute in thread attribute object.

Parameters:

attr thread attribute object

stacksize value of stack size

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function sets the stack size attribute of the thread attribute object *attr* to the value specified in *stacksize*.

Default stack size is RTEMS_MINIMUM_STACK_SIZE.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_thread_attr_setstacksize).

A.1.3 *Thread Specific Keys*

A.1.3.1 *void$*$ r2g_getspecific (r2g_key_t* key*)*

Get stored value.

Parameters:

key thread specific key object

Returns:

On success, the value associated with *key* is returned; on error, NULL is returned and **errno** is set to indicate the error.

Return values:

EINVAL Invalid Key

This function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling r2g_getspecific() or r2g_setspecific() with a *key* value not obtained from r2g_key_create() or after *key* has been deleted with r2g_key_-delete() is undefined.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_getspecific).

A.1.3.2 *int r2g_key_create (r2g_key_t* ∗ key, *int* fid*)*

Thread-specific data key creation.

Parameters:

key thread specific key object

fid Function ID of destructor called at thread exit

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

ENOMEM No memory space available to allocate *key*

This function creates a thread-specific data key visible to all threads in the process. Key values provided by r2g_key_create() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by r2g_setspecific() are maintained on a per-thread basis and persist for the life of the calling thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-zero function identifier, and the thread has a non-NULL value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_key_create). However, the *destructor* parameter is of cause a function identifier and not a function pointer!

A.1.3.3 *int r2g_key_delete (r2g_key_t* key*)*

Thread-specific data key deletion.

Parameters:

key thread specific key object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid Key

EFAULT Destructor function not executable on this processor

This function deletes a thread-specific data key previously returned by r2g_-key_create(). The thread-specific data values associated with *key* need not be NULL at the time r2g_key_delete() is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after r2g_key_delete() is called. Any attempt to use *key* following the call to r2g_key_delete() results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_key_delete).

A.1.3.4 *int r2g_setspecific (r2g_key_t* key*, const void ∗* value*)*

Store thread specific value.

Parameters:

key thread specific key object

value Value to be stored

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid Key

This function associates a thread-specific value with a *key* obtained via a previous call to r2g_key_create(). Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling r2g_getspecific() or r2g_setspecific() with a *key* value not obtained from r2g_key_create() or after *key* has been deleted with r2g_key_-delete() is undefined.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_setspecific).

A.2 MEMORY SUBSYSTEM

The memory subsystem is responsible for managing the shared memory and to copy huge data via DMA.

A.2.1 *Dynamic Shared Memory Allocation*

The dynamic allocator is a basic part of R^{2}G. The user interface consists of the functions r2g_mem_alloc(), r2g_mem_free(), and r2g_mem_compress().

Both R^{2}G itself and applications can use the function r2g_mem_alloc() to dynamically allocate shared memory. If the allocated memory is not needed anymore, the memory can be freed by calling r2g_mem_free(). It is always a good idea to free unused memory, because running out of shared memory is fatal for R^{2}G. Some management structures (such as r2g_thread_t) require additional memory allocations when performing basic operations.

However, in some cases it will be possible that not enough memory is available, even though memory was freed. This happens when the memory is filled with too much buddy cache objects. To free unused object the application has to call r2g_mem_compress().

A.2.1.1 *void∗ r2g_mem_alloc (size_t* size*)*

Allocation of memory.

Parameters:

size needed amount of memory

Returns:

Returns address of the allocated memory on success; on error, NULL is returned and **errno** is set to indicate the error.

Return values:

ENOMEM Not enough memory available to satisfy the request

Allocates *size* bytes inside the shared memory address range.

A.2.1.2 *void r2g_mem_compress (void)*

Free internal memory.

Shrinks space needed by the buddy cache. Normally the cache is always growing. Tracking each cache element will result in a huge memory and/or computation overhead. However, it is possible that the whole memory gets full with buddy containers. In this case calling r2g_mem_compress can maybe free unused buddies.

Warning:

This function may take some time. R²G restructures the cache objects, so that free cache lines will be created, which then can be freed. During this time the whole memory subsystem is locked for each processor. Maybe you get trouble with hard real-time constrains.

A.2.1.3 *int r2g_mem_free (void* ∗ addr*)*

Free memory.

Parameters:

addr memory to free

Frees memory pointed to by *addr*. The memory has to be previously allocated by r2g_mem_alloc().

'free' means to make the memory available for further allocation and to concatenate belonging memories, so that larger allocation requests can be satisfied.

A.2.2 *DMA Transfer Engine*

To support huge block transfers in form of Direct Memory Access (DMA) two functions are provided; r2g_dma_issue_1d() which issues a new DMA command, and r2g_dma_sync() which synchronizes the DMA transfer. DMA transfers are always asynchronous to CPU execution. Thus, the CPU can perform other tasks until the transfer is finished. There is no way for the application to determine the arrival of the data. Moreover, the DMA engine is explicitly allowed to copy the data in any order. Hence, the source buffer must not be modified until the transfer will be completed. However, the application can synchronize the DMA by calling r2g_dma_sync(). This call blocks until every byte is copied.

A.2.2.1 *int r2g_dma_issue_1d (size_t* size*, unsigned char* ∗ srcaddr*, unsigned char* ∗ dstaddr*, r2g_dma_obj_t* ∗ dmaobj*)*

One dimensional DMA transfer.

Parameters:

size data size

srcaddr source address

dstaddr destination address

dmaobj DMA reference object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EFAULT Error in R²G/RTEMS

EBUSY All units are busy

This functions performs a DMA transfer from *srcaddr* to *dstaddr*.

This function is non-blocking. When this function returns the DMA transfer is possible not finished. To synchronize call r2g_dma_sync() with the *dmaobj*.

Note:

The DMA unit used inside the CoMET platform can maximally transfer 134,215,680 Byte.

Warning:

DMA's are not cache coherent. You have to flush caches manually! (For CoMET use BSP_dcache_flush())

A.2.2.2 *int r2g_dma_issue_2d (unsigned int* rowsize*, unsigned int* rows*, unsigned int* rowlen1*, unsigned int* rowlen2*, int ∗* srcaddr*, int ∗* dstaddr*, r2g_-dma_obj_t ∗* dmaobj*)*

Two dimensional DMA transfer.

Parameters:

rowsize size of the row which has to be copy

rows number of rows to copy

rowlen1 size of one row in the first location

rowlen2 size of one row in the second location

srcaddr source address

dstaddr destination address

dmaobj DMA reference object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EFAULT Error in R^{2}G/RTEMS

EBUSY All units are busy

This functions performs a 2D DMA transfer from *srcaddr* to *dstaddr*. All sizes are in respect to the MPARAM word size (4 byte).

With *addrtype* the address type is specified:

- R2G_ADDRTYPE_LOCAL

  address in the scope of the processor

- R2G_ADDRTYPE_GLOBAL

  address in the scope of the system

This function is non-blocking. When this function returns the DMA transfer is possible not finished. To synchronize call r2g_dma_sync() with the *dmaobj*.

Warning:

Use only **R2G_ADDRTYPE_LOCAL** as address type when possible. Otherwise, the global DMA unit may gets overloaded.
Use this function only on MPARM!
DMA's are not cache coherent, especially when you copy on local memories across core boundaries! Copy operations to the own local store results a D-Cache flush!

A.2.2.3 *int r2g_dma_sync (r2g_dma_obj_t* $*$ dmaobj*)*

Synchronize DMA transfer.

Parameters:

dmaobj DMA reference object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid DMA object

This functions synchronizes the DMA transfer reflected by *dmaobj*. After return the DMA is completely finished.

A.3 SYNCHRONIZATION PRIMITIVES

R²G supports four different synchronization primitives. All of those are usable across processor core boundaries. Moreover, most parts of the POSIX standard are implemented.

A.3.1 *Semaphore*

The basic synchronization primitives are semaphores. The generic implementation of the other synchronization primitives is based on semaphores. The semaphore interface has nearly POSIX semantic.

A.3.1.1 *int r2g_sem_destroy (***r2g_sem_t** $*$ sem*)*

Destroys the unnamed semaphore.

Parameters:

sem address of semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

This function destroys the unnamed semaphore at the address pointed to by *sem*.

Warning:

Destroying a semaphore that other processes or threads are currently blocked on (in r2g_sem_wait()) produces undefined behavior.

Remarks:

Conforming to POSIX.1-2001 (sem_destroy).

A.3.1.2 *int r2g_sem_getvalue (***r2g_sem_t** ∗ sem*, int* ∗ sval*)*

Get value of a semaphore.

Parameters:

sem address of semaphore

sval integer buffer for storing semaphore value

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

This function returns the current semaphore value of the semaphore object *sem* in the buffer pointed to by *sval*.

Remarks:

Conforming to POSIX.1-2001 (sem_get_value).

A.3.1.3 *int r2g_sem_init (***r2g_sem_t** ∗ sem*, int* pshared*, unsigned int* value*)*

Initializes a unnamed semaphore.

Parameters:

sem address of semaphore

pshared semaphore is shared between between processes

value initial value for the semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EINVAL Invalid address for shared semaphore

ENOMEM No memory for internal allocations

EINTR The call was interrupted by a signal handler

This function initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

Warning:

If you need a semaphore across core boundaries, ∗sem has to be allocated in the shared memory

Remarks:

Conforming to POSIX.1-2001 (sem_init).

A.3.1.4 *int r2g_sem_post (***r2g_sem_t** ∗ sem*)*

Increments (unlocks) a semaphore.

Parameters:

sem address of semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EFAULT Error in RTEMS subsystem

EOVERFLOW The maximum allowable value for a semaphore would be exceeded

This function increments the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another thread blocked in a r2g_sem_wait() call will be woken up and proceed to lock the semaphore.

Remarks:

Conforming to POSIX.1-2001 (sem_post).

A.3.1.5 *int r2g_sem_trywait (***r2g_sem_t** ∗ sem*)*

Decrements (locks) a semaphore if possible.

Parameters:

sem address of semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EAGAIN The operation could not be performed without blocking

EFAULT Error in RTEMS subsystem

EINTR The call was interrupted by a signal handler

This function is the same as r2g_sem_wait(), except that if the decrement cannot be immediately performed, then call returns an error (**errno** set to EAGAIN) instead of blocking.

Remarks:

Conforming to POSIX.1-2001 (sem_trywait).

A.3.1.6 *int r2g_sem_wait (***r2g_sem_t** ∗ sem*)*

Decrements (locks) a semaphore.

Parameters:

sem,: address of semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EFAULT Error in RTEMS subsystem

EINTR The call was interrupted by a signal handler

This function decrements the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

Remarks:

Conforming to POSIX.1-2001 (sem_wait).

A.3.1.7 *int r2g_sem_wait_nointr (***r2g_sem_t** ∗ sem*)*

Decrements (locks) a semaphore without signal interruption.

Parameters:

sem,: address of semaphore

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EFAULT Error in RTEMS subsystem

This function decrements the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero). The call restart itself when a signal handler interrupts.

Remarks:

This functtion is **not** part of POSIX. It is just a wrapper around r2g_sem_-wait().

A.3.2 *Mutex*

A.3.2.1 *int r2g_mutex_destroy (***r2g_mutex_t** ∗ mutex*)*

Destroy a mutex.

Parameters:

mutex Mutex object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function destroys the mutex object referenced by *mutex*.

Warning:

Destroying a mutex that other threads are currently blocked on (in r2g_-mutex_lock()) produces undefined behavior.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutex_destroy).

A.3.2.2 *int r2g_mutex_init (***r2g_mutex_t** ∗ mutex*, const* **r2g_mutexattr_t** ∗ attr*)*

Initialize a mutex.

Parameters:

mutex Mutex object

attr Attribute of mutex

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

ENOMEM Insufficient memory to allocate internal data structures

EINVAL Invalid attributes

EFAULT Mutex object is not in shared memory, but shared mutex requested

This function initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutex_init).

A.3.2.3 *int r2g_mutex_lock (***r2g_mutex_t** ∗ mutex*)*

Lock a mutex.

Parameters:

mutex Mutex object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EFAULT Error in RTEMS subsystem

EAGAIN Maximum recursion deep reached

EDEADLK The current thread already owns the mutex

This function locks the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread will be blocked until the mutex becomes available.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutex_lock).

A.3.2.4 *int r2g_mutex_trylock (***r2g_mutex_t** ∗ mutex*)*

Try to lock a mutex.

Parameters:

mutex Mutex object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EFAULT Error in RTEMS subsystem

EAGAIN Maximum recursion deep reached

EBUSY The mutex could not be acquired because it was already locked

EDEADLK The current thread already owns the mutex

This function trys to lock the mutex object referenced by *mutex*. If the mutex is already locked, the function returns EBUSY.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutex_trylock).

A.3.2.5 *int r2g_mutex_unlock (***r2g_mutex_t** ∗ mutex*)*

Unlock a mutex.

Parameters:

mutex Mutex object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EFAULT Error in RTEMS subsystem

EPERM The current thread does not own the mutex

This function unlocks the mutex object referenced by *mutex*.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutex_unlock).

A.3.2.6 *int r2g_mutexattr_destroy (***r2g_mutexattr_t** ∗ attr*)*

Destroy mutex attribute object.

Parameters:

attr mutex attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

0 Success

EINVAL Invalid attribute pointer

This function destroys the mutex attributes object pointed to by *attr*. Destroying a mutex attributes object has no effect on mutex that were created using that object.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_destroy).

A.3.2.7 *int r2g_mutexattr_getpshared (const* **r2g_mutexattr_t** ∗ attr, *int* ∗ pshared*)*

Get pshared attribute of mutex attribute object.

Parameters:

attr mutex attribute object

pshared buffer for the mutex pshared attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the pshared attribute of the mutex attribute object *attr* in the buffer pointed to by *pshared*.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_getpshared).

A.3.2.8 *int r2g_mutexattr_gettype (const* **r2g_mutexattr_t** ∗ attr*, int* ∗ type*)*

Get mutex type attribute of mutex attribute object.

Parameters:

attr mutex attribute object

type buffer for the mutex type attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the mutex type attribute of the mutex attribute object *attr* in the buffer pointed to by *type*.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_gettype).

A.3.2.9 *int r2g_mutexattr_init (***r2g_mutexattr_t** ∗ attr*)*

Initialize a mutex attribute object.

Parameters:

attr mutex attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function initializes the mutex attributes object pointed to by *attr* with default attribute values. After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more r2g_mutex_init() calls that create mutexes.

When a mutex attributes object is no longer required, it should be destroyed using the r2g_mutexattr_destroy() function.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_init).

A.3.2.10 *int r2g_mutexattr_setpshared (***r2g_mutexattr_t** ∗ attr, *int* pshared*)*

Set pshared attribute of mutex attribute object.

Parameters:

attr mutex attribute object

pshared the mutex pshared attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid *pshared* argument

EINVAL Invalid attribute pointer

This function sets the pshared attribute in the mutex attribute object pointed to by *attr*. Valid settings for *pshared* include:

- R2G_PROCESS_SHARED

  This type of mutex is shared between threads on different CPU's

- R2G_PROCESS_PRIVATE

  This type of mutex is not guaranteed to be shared between threads on different CPU's

R^{2}G Default: R2G_PROCESS_SHARED

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_setpshared).

A.3.2.11 *int r2g_mutexattr_settype (***r2g_mutexattr_t** ∗ attr, *int* type*)*

Set mutex type attribute of mutex attribute object.

Parameters:

attr mutex attribute object

type the mutex type attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid *type* argument

EINVAL Invalid attribute pointer

This function sets the mutex type attribute in the specified mutex attribute object pointed to by *attr*. Valid settings for *type* include:

- R2G_MUTEX_NORMAL

  This type of mutex does not detect deadlock. An attempt to relock this mutex without first unlocking it deadlocks. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

- R2G_MUTEX_ERRORCHECK

  This type of mutex provides error checking. An attempt to relock this mutex without first unlocking it returns with an error. An attempt to unlock a mutex that another thread has locked returns with an error. An attempt to unlock an unlocked mutex returns with an error.

- R2G_MUTEX_RECURSIVE

  A thread attempting to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type R2G_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. An attempt to unlock a mutex that another thread has locked returns with an error. An attempt to unlock an unlocked mutex returns with an error.

- R2G_MUTEX_DEFAULT

  Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type that was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type that is not locked results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1c (pthread_mutexattr_settype).

A.3.3 *Condition Variable*

A.3.3.1 *int r2g_cond_broadcast (***r2g_cond_t** $*$ cond*)*

Wakeup all waiting processes.

Parameters:

cond Condition object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function unblocks all threads currently blocked on the specified condition variable *cond*.

It can be called by a thread whether or not it currently owns the mutex that threads calling r2g_cond_wait() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling r2g_cond_broadcast() or r2g_-cond_signal().

This function has no effect if there are no threads currently blocked on *cond*.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cond_broadcast).

A.3.3.2 *int r2g_cond_destroy (***r2g_cond_t** $*$ cond*)*

Destroy a condition variable.

Parameters:

cond Condition object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

0 Success

Destroys the given condition variable *cond*.

Warning:

Destroying a condition variable that other threads are currently blocked on (in r2g_cond_wait()) produces undefined behavior.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cond_destroy).

A.3.3.3 *int r2g_cond_init (***r2g_cond_t** $*$ cond, *const* **r2g_condattr_t** $*$ attr*)*

Initialize a condition variable.

Parameters:

cond Condition object

attr Attribute of conditional

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

0 Success

ENOMEM Insufficient memory to allocate internal data structures

EINVAL Invalid attributes

EFAULT Condition variable object is not in shared memory, but shared condition variable requested

Initialize the condition variable *cond* with the attributes *attr*. If *attr* is NULL, the default attributes will be used.

Attempting to initialize an already initialized condition variable results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cond_init).

A.3.3.4 *int r2g_cond_signal (***r2g_cond_t** ∗ cond*)*

Wakeup one process.

Parameters:

cond Condition object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

It can be called by a thread whether or not it currently owns the mutex that threads calling r2g_cond_wait() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling r2g_cond_broadcast() or r2g_-cond_signal().

This function has no effect if there are no threads currently blocked on *cond*.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cond_signal).

A.3.3.5 *int r2g_cond_wait (***r2g_cond_t** ∗ cond*,* **r2g_mutex_t** ∗ mutex*)*

Wait on a condition variable.

Parameters:

cond Condition object

mutex Mutex reference

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

0 Success

The r2g_thread_cond_wait() function block on a condition variable. It has to be called with mutex locked by the calling thread or undefined behavior results.

This functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to r2g_cond_broadcast() or r2g_cond_-signal() in that thread behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_cond_wait).

A.3.3.6 *int r2g_condattr_destroy (***r2g_condattr_t** ∗ attr*)*

Destroy condition variable attribute object.

Parameters:

attr condition variable attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function destroys the condition variable attributes object pointed to by *attr*. Destroying a condition variable attributes object has no effect on condition variables that were created using that object.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_condattr_destroy).

A.3.3.7 *int r2g_condattr_getpshared (const* **r2g_condattr_t** ∗ attr*, int* ∗ pshared*)*

Get pshared attribute of condition variable attribute object.

Parameters:

attr condition variable attribute object

pshared buffer for the condition variable pshared attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid attribute pointer

This function returns the pshared attribute of the condition variable attribute object *attr* in the buffer pointed to by *pshared*.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_condattr_getpshared).

A.3.3.8 *int r2g_condattr_init (***r2g_condattr_t** ∗ attr*)*

Initialize a condition variable attribute object.

Parameters:

attr cond attribute object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function initializes the condition variable attribute object pointed to by *attr* with default attribute values. After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more r2g_cond_init() calls that create condition variables.

When a condition variable attribute object is no longer required, it should be destroyed using the r2g_mutexattr_destroy() function.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_condattr_init).

A.3.3.9 *int r2g_condattr_setpshared (***r2g_condattr_t** ∗ attr*, int* pshared*)*

Set pshared attribute of condition variable attribute object.

Parameters:

attr condition variable attribute object

pshared the condition variable pshared attribute value

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

EINVAL Invalid *pshared* argument

EINVAL Invalid attribute pointer

This function sets the pshared attribute in the condition variable attribute object pointed to by *attr*. Valid settings for *pshared* include:

- R2G_PROCESS_SHARED

  This type of condition variable is shared between threads on different CPU's

- R2G_PROCESS_PRIVATE

  This type of condition variable is not guaranteed to be shared between threads on different CPU's

R²G Default: R2G_PROCESS_SHARED

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_condattr_setpshared).

A.3.4 *Barrier*

Barriers can be used to synchronize N threads. Until the N'th thread has called r2g_barrier_wait() the other N - 1 threads are blocked.

A.3.4.1 *int r2g_barrier_destroy (***r2g_bar_t** ∗ bar*)*

Free barrier resources.

Parameters:

bar Barrier object

Returns:

On success, 0 is returned; on error, an error number is returned.

This function destroys the barrier referenced by *bar* and releases any resources used by the barrier.

Warning:

Destroying a barrier that other threads are currently blocked on (in r2g_-barrier_wait()) results in undefined behavior.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_barrier_destroy).

A.3.4.2 *int r2g_barrier_init (***r2g_bar_t** ∗ bar*, uint32_t* n*)*

Create a barrier synchronization object.

Parameters:

bar Barrier object

n Number of threads to synchronize

Returns:

On success, 0 is returned; on error, an error number is returned.

Return values:

ENOMEM Insufficient memory to allocate internal data structures

This functions initializes the barrier *bar*. Thereafter, exactly *n* - 1 threads will block when calling r2g_barrier_wait(). When the *n* 'th thread calls r2g_barrier_-wait(), all threads are unblocked.

Attempting to initialize an already initialized barrier results in undefined behavior.

Remarks:

This function **not** directly conforms to POSIX 1003.1-2001. However, it is working in the same way as pthread_barrier_init(), but without attributes.

A.3.4.3 *int r2g_barrier_wait (***r2g_bar_t** ∗ bar*)*

Synchronize on a barrier.

Parameters:

bar Barrier object

Returns:

On success, 0 is returned; on error, an error number is returned.

Synchronize the calling thread on the barrier *bar*. The thread gets automatically unblocked if the number of threads defined by r2g_barrier_init() have reached this function.

Remarks:

Conforming to POSIX 1003.1-2001 (pthread_barrier_wait).

A.4 BOARD SPECIFIC R^{2}G SUPPORT ROUTINES

Most parts of R^{2}G are platform independent and thus are only based on the services provided by the RTEMS operating system. However, some parts are very specific to the platform and the memory address space. Hence, such parts are directly integrated into to board support package (BSP) of RTEMS.

This includes:

- DMA Support,
- Spinlocks,
- Idle routine, and
- Performance measurement

All those parts have to be adapted to support a new platform.

Please also note that those functions are not POSIX conform at all.

A.4.1 *System Support*

A.4.1.1 *void∗ r2gsupp_idle (uintptr_t* ignored*)*

Idle function.

Parameters:

ignored ignored parameter

Returns:

This function never returns.

This function implements the board specific idle routine. In case of CoMET the CPU enters a energy safe execution mode.

A.4.2 *Basic Synchronization*

Every system with concurrently running threads or processes needs a way to synchronize. To implement such a synchronization at least one atomic operation is required. Those operations can be directly implemented in the instruction set architecture of the processor, like test-and-set, or implemented trough special devices.

In case of CoMET the instructions 'ldrex' and 'strex' are used. They implement atomic load and store to a memory cell.

The MPARM implements a special semaphore device which is used for this purpose.

A.4.2.1 *void r2g_spin_lock (***r2g_spinlock_t** $*$ lock*)*

Close a spinlock.

Parameters:

spinlock reference

This function locks the spinlock represented by *lock*. If the lock is available the function returns, immediately. If the lock currently is closed the call spins until the lock gets available

A.4.2.2 *int r2g_spin_trylock (***r2g_spinlock_t** $*$ lock*)*

Try to close a spinlock.

Parameters:

spinlock reference

Returns:

Returns 0 on success; on error, -1 is returned and **errno** is set to indicate the error.

Return values:

EAGAIN The operation could not be performed without blocking

This function is the same as r2g_spin_lock(), except that if the locking cannot be immediately performed, then call returns an error (**errno** set to EAGAIN) instead of spinning.

A.4.2.3 *void r2g_spin_unlock (***r2g_spinlock_t** $*$ lock*)*

Open the spinlock.

Parameters:

spinlock reference

This function unlocks the spinlock represented by *lock*.

A.4.3 *Performance Measurement*

It is very important to measure the performance of an embedded system. Hence, the designer can compare different implementations regarding energy efficiency and timing.

The start/stop functions in this module trigger the measurement of the simulator.

A.4.3.1 *void multi_start_metric (void)*

This function starts the measurement of the executed instructions on the calling CPU. When called multiple times, only the first call will start the measurement.

A.4.3.2 *void multi_stop_metric (void)*

This function stops the measurement of the executed instructions on the calling CPU. The values are printed with special simulator commands.

The measurement is only stopped, when calling this function exact the same amount as multi_start_metric(). This behavior enables measurement in recursive functions and in loops.

A.4.3.3 *unsigned int start_energy_metric (void)*

This function starts the measurement of the energy consummation of the system.

Note:

> Only implemented on CoMET. For MPARM this is equal to multi_start_-metric()

A.4.3.4 *unsigned int stop_energy_metric (void)*

This function stops the measurement of the energy consummation of the system. To retrieve the energy values, you have to look at the simulator log file respectively the simulator output.

Note:

> Only implemented on CoMET. For MPARM this is equal to multi_stop_-metric()

BIBLIOGRAPHY

[1] MNEMEE – Memory maNagEMEnt technology for adaptive and efficient design of Embedded systems. `http://www.mnemee.org/`.

[2] Daniel Cordes, Peter Marwedel, and Arindam Mallik. Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming. In *Proceedings of CODES+ISSS, 2010)*, Scottsdale / US, October 2010.

[3] Yiannis Iosifidis, Arindam Mallik, Stylianos Mamagkakis, Eddy De Greef, Alexandros Bartzas, Dimitrios Soudris, and Francky Catthoor. A framework for automatic parallelization, static and dynamic memory optimization in MPSoC platforms. In *Proceedings of the 47th ACM/IEEE Design Automation Conference – DAC*, pages 549–554, Anaheim (California) / USA, June 2010.

[4] Christos Baloukas, Lazaros Papadopoulos, Dimitrios Soudris, Sander Stuijk, Olivera Jovanovic, Florian Schmoll, Daniel Cordes, Robert Pyka, Arindam Mallik, Stylianos Mamagkakis, François Capman, Séverin Collet, Nikolaos Mitas, and Dimitrios Kritharidis. Mapping embedded applications on MPSoCs: the MNEMEE approach. In *Proceedings of the Annual Symposium on VLSI*, Lixouri, Kefalonia, Greece, July 2010.

[5] Christos Baloukas, Lazaros Papadopoulos, Robert Pyka, Dimitrios Soudris, and Peter Marwedel. An Automatic Framework for Dynamic Data Structures Optimization in C. In *Proceedings of the 18th international conference on Very Large Scale Integration (VLSI) System-on-Chip (SoC), VLSI-SOC 2010*, Madrid, Spain, September 2010.

[6] ArtistDesign Network of Excellence on Embedded Systems Design. `http://www.artist-embedded.org/`.

[7] Arindam Mallik, Maryse Wouters, Peter Lemmens, Eddy De Greef, and Thomas J. Ashby. Source-to-source optimizations of statically allocated data mapping on MPSoC platforms. In *Poster at MPSoC Workshop Rheinfels*, Rheinfels / Germany, 2010.

[8] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *The Journal of VLSI Signal Processing*, 41:169–182, 2005.

[9] Comet simulator. `http://www.synopsys.com/Community/Interoperability/SystemLevelCatalyst/Pages/MVaST.aspx`.

[10] Michael Dales. Swarm 0.44 documentation, 2000. `http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html`.

[11] Jennifer Eyre and Jeff Bier. Infineon's TriCore Tackels DSP. *Microprocessor Report*, 13.

[12] Eclipse ide. `http://www.eclipse.org/`.

[13] ARM 1176 CPU. `http://arm.com/products/processors/classic/arm11/`.

[14] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[15] ICD-C Compiler framework. `http://www.icd.de/es/icd-c/`.

[16] ICD - Informatik Centrum Dortmund. `http://www.icd.de`.

[17] IMEC - Interuniversity Microelectronics Centre. `http://www.imec.be`.

[18] Andreas Heinig. R²G home page. `http://www.andreasheinig.de/permalink/10121/`.