An Experimental Testbed for Embedded Real Time Ada 95

W. M. Walker P. T. Woolley and A Burns
Real-Time Systems Research Group
Department of Computer Science
University of York. UK

Abstract

The modifications made to Ada during the 9X process have resulted in a language that is ideally suited to programming real-time systems. In this paper we investigate the difficulties in realising this potential. In particular, we consider the issues raised when porting the public GNAT system on to a bare processor and producing a predictable and effective run–time system. As we were also concerned with issues of distribution via a CAN broadcast bus, support for the CAN protocol was included in the run-time system. In addition to investigating the performance of real-time Ada 95 applications we were also interested in more general issues associated with embedded kernel support for Ada 95. To facilitate these investigations and obtain the required level of performance, the thread package via which GNAT implement various aspects of the Ada language has been re-implemented. We discuss the architecture selected for this embedded kernel implementation and its relation to the architecture of the GNAT compiler.

1. Introduction

The availability of GNAT has significantly improved access to the Ada 95 language for general purpose programming. However, the public releases of GNAT are designed for target platforms on which application programs are dependent on services provided by an underlying operating system. Since these operating systems do not exhibit predictable behaviour there is no easy way in which GNAT can be used to investigate the real-time behaviour of Ada 95 programs. In the case of distributed real-time systems, these problems are compounded by the non deterministic behaviour of the communication protocols often used to interconnect network nodes. The forthcoming support for the Ravenscar profile addresses one set of topics [1], but the full Ada 95 tasking model also has much to offer for dynamic, autonomous systems. There is thus a need to allow experimentation with the full language. In order to provide a testbed on which this is possible it was decided to port the GNAT compiler on to a new platform comprising a number of bare board computers built around the Motorola MC68376 microcontroller (see Appendix 1). The problems associated with the non deterministic nature of the operating system services required by GNAT were addressed by implementing an embedded kernel which provides these services in a predictable manner. In order to facilitate experimentation with distributed real-time Ada 95 applications, the embedded kernel also provides predictable communications services using the Controller Area Network (CAN) protocol. The implementation of the communications protocol used by the embedded kernel is discussed in a related paper [6]. This paper focuses on the issues arising from porting the GNAT compiler onto an embedded platform.

2. The GNAT Ada 95 Compiler

In order to generate executable Ada 95 code for the processor boards in the experimental platform it was necessary to build a cross compiling version of GNAT and implement an embedded kernel which provides the run time services required by GNAT. Before discussing the problems associated with building the cross compiler it is useful to examine the architecture of the GNAT compiler and the issues which this architecture raises when porting the compiler on to an embedded platform.

2.1. The GNAT Architecture

To facilitate platform portability GNAT implements many of the key features of the Ada 95 language (Figure 1) using standard POSIX Application Program Interface (API) calls for thread services [3] [4]. Not all of the target platforms for GNAT support the full range of standard POSIX API's. Consequently, the compiler provides a method for mapping standard POSIX Pthread API service calls to those thread services provided on the target platform. This is achieved using the architecture shown in Figure 2. The Gnu Ada Run time Layer (GNARL) comprises the bulk of the

Multi tasking
Protected Objects
Rendezvous
Asynchronous Transfer of Control
Delay Function
Interrupt and Signal Handling

Figure 1. GNAT Ada 95 features implemented using Posix Pthread services

Application
Gnu Ada Run Time Layer
Gnu Lower Level Layer
Pthread Layer
Operating System
Machine

Figure 2. Architecture of the GNAT Ada 95 compiler

compiler code and is intended to be 100% portable across all target platforms. Those aspects of the compiler that are dependent upon the thread services provided by an underlying operating system are included in the Gnu Lower Level Layer (GNULL). In the current implementation of the compiler the thread services provided by GNULL are incorporated into a single Ada package, system.os_interface. The specification of this package includes :

- A set of Ada 95 data type declarations which correspond to those used by the standard POSIX Pthread functions.
- A set of Ada 95 function headers which correspond to the standard POSIX Pthread API's.

In cases where an underlying operating system fully supports a standard POSIX Pthread API, the os_interface package specification simply contains Ada 95 code to import the function call from the operating system's run time library. If the operating system provides a nonstandard implementation of a thread service required by the compiler then the package body of os_interface contains the declaration of a wrapper function which emulates the required Pthread service using calls to the operating system's available thread and system services. This wrapper function has a header which corresponds to the standard POSIX API being emulated by the function. Consequently, the nonstandard nature of the implementation of the thread service is not visible to the GNARL.

2.2. Porting the GNAT Compiler

The source code distribution of the GNAT compiler includes os_interface (GNULL) packages for a range of common platforms. When porting the compiler to an unsupported platform it is necessary to implement an appropriate version of os_interface which maps the thread services of the target operating system to those required by GNARL. In the case of an embedded system an alternative approach is to select one of the supported os_interface packages and implement an embedded kernel which provides the thread services specified in the os_interface package. This approach has the advantage that it is not necessary to make any modifications to the GNAT source code. Consequently, it should be relatively easy to upgrade the embedded version of the compiler as future upgrades of GNAT are released. For the purposes of this project the FSU version of GNARL was chosen since the source code for the Pthread services used by this implementation is freely available which provided some guidance when implementing the embedded kernel.

The procedures associated with porting GNAT to a new platform involve building a cross compiling version of the GNAT compiler and the GNAT linker and then using this cross compiler to build the libraries which constitute the GNARL and GNULL layers of the compiler (these layers are written in Ada 95 and are provided in the source code distribution of GNAT). Finally (in the case of an embedded system), it is necessary to implement the embedded kernel that provides the runtime services required by GNARL. The source code for the embedded kernel must then be cross compiled to produce a run time library. Once these tasks have been performed GNAT can be used to compile user applications targeted at the new platform. The GNAT linker is then used to link the application code with the code from the GNARL, GNULL and embedded kernel libraries to produce an executable file.

2.3. Building a GNAT Cross Compiler

Before carrying out the cross compilation of GNAT it is necessary to build an appropriate cross compiling version of the gcc C compiler (parts of the GNAT compiler are written in C). Building the C cross compiler is a relatively straight forward process since both Linux host and Motorola target architectures are supported by the Makefile supplied with the gcc C source distribution. The only problem associated with the build process relates to the fact that the MC68332 microprocessor boards do not support the bitfield instructions found on more common Motorola CPU's. Additionally, the target boards do not have floating point processors and hence floating point arithmetic must be achieved using software emulation. However, these issues do not present problems when using the C compiler directly since compiler switches are available to prevent the use of bitfield instructions and force the use of software emulation for floating point calculations. Having built the gcc C cross compiler it is then possible to proceed with building the GNAT cross compiler.

The source distribution for GNAT includes a generic Makefile and a configure script which produces a customised Makefile that allows the crosscompilation process to be performed automatically for supported platforms. When using the configure script one simply specifies the host machine operating system (in this case Linux) and the target architecture. This presented some problems since the target architecture required was not supported by the configure script. One aspect of these problems related to the selection of the appropriate os_interface packages to include in the compilation process. The Make process selects the appropriate package from the range available by examining the target specified in the call to the configure script. As the target required was not supported, manual analysis and editing of the Makefile was required to ensure that the appropriate components of os_interface were selected. A further problem related to the lack of a floating point hardware and support for bitfield instructions in the target processors. Although gcc C allows these features to be made unavailable when compiling C programs, the GNAT Makefile did not apply the appropriate switches when compiling those components of the compiler which are written in C. These problems were resolved by manually adding the appropriate switches to the calls to gcc in the Makefile and manipulation of the machine descriptor files which specify the registers available on the target architectures. Once these problems had been resolved it was possible to build the GNAT cross compiler and GNARL and GNULL libraries as required. It was then possible to develop the embedded kernel.

3. Implementation of the embedded Pthread kernel

As discussed in the previous section, the operating system services normally required by the GNULL layer of GNAT are provided by an embedded kernel. When deciding how to implement this kernel there were a number of issues to consider. Firstly, the efficiency of the implementation was important in terms of the kernel overhead imposed during the process of context switching and the blocking and unblocking of threads at mutexes, condition variables and in relation to timed delay and interrupt pending events. It is obviously desirable that these overheads be minimised in any kernel. In the case of the experimental platform there were explicit constraints on the amount of time taken to perform a context switch which arose as a result of the need to ensure that all messages transmitted using the CAN protocol were received at their destination nodes. It was required that a task awaiting the reception of a CAN message should be able to be suspended pending the arrival of the message and then awoken when the message arrives. In order that messages are not lost it was necessary to ensure that the kernel could repeatedly suspend and awaken a task when the task was receiving a sequence of messages at the maximum transmission rate allowed by the CAN communications protocol. A second issue influencing the implementation of the embedded kernel relates to platform portability. Although the main target platform for the embedded system is the MC68376 based processor boards described in Appendix 1, the experimental platform also includes a number of control systems built around Motorola 68030 and MC68332 boards. There are some differences in the instruction sets supported by these boards and the methods whereby realtime clock and interrupt services are implemented. In order to enable the embedded kernel to be easily ported between the various boards in the system it was desirable that the embedded kernel architecture isolate those aspects of the kernel related to hardware specific issues. A further factor which needed to be considered when implementing the kernel related to issues of flexibility and maintainability. In addition to providing an experimental platform for investigating the behaviour of realtime Ada 95 applications we are also interested in more general issues relating to kernel support for Ada 95. For example, the influence on performance and memory requirements of the queuing algorithms used by the kernel and the potential benefits of providing kernel level support for alternative scheduling algorithms such as that used in the Dual Priority protocol [2]. To facilitate investigation of these issues it was desirable that those aspects of the kernel which directly relate to the implementation of individual Pthread API's should be separated from those aspects which relate to the interaction between threads and the maintenance of the various thread queues in the system. The choice of language(s) in which to implement the embedded kernel required consideration of the kernel overheads associated with an Ada 95 implementation versus an implementation using the C language. For realtime applications Ada 95 was judged to be the more appropriate. However, initial experimentation showed that the kernel overheads associated with an Ada 95 implementation were significantly greater than those incurred when the kernel functions were implemented in a combination of C and assembler (see section 4). In order to optimise the performance of the embedded kernel the C/assembler implementation route was chosen. However, future work will involve developing an Ada 95 implementation of the kernel with the CAN bus operating at a reduced data rate if necessary.

GNARL
GNULL
Pthread API Layer
Thread Management Layer
Real Time Clock and Interrupt Service Layer
Single Board Computer

Figure 3. Architecture of the embedded system

3.1. Architecture of the Embedded GNAT Compiler

To meet the objectives discussed above the embedded kernel uses the architecture illustrated in Figure 3. The GNARL and GNULL layers are identical to those of the GNAT architecture shown in Figure 2. The Pthread API layer consists of a set of functions which implement the Pthread services required by GNULL (note GNULL only requires a subset of the POSIX thread services and hence the embedded kernel only supports this subset). When implementing the Pthread functions it is necessary to provide lower level services which are invoked at schedule decision points (i.e. points at which the status of a thread or threads in the system is changed). For example :

- The creation of a new thread or a change in the priority of an existing thread.
- The suspension of a thread when it becomes blocked on a mutex or condition variable.
- The suspension of a thread for a specified period of time or until a hardware interrupt occurs.
- Returning a thread to the run queue on unblocking at a mutex or condition variable, after the expiry of a timed delay or on the occurrence of a hardware interrupt.

These services are provided by the thread manager layer shown in Figure 3. The thread manager has two main responsibilities :

- Priority based arbitration between threads when the status of the threads in the system changes.
- The management of the thread queues associated with threads that are delayed for a given period of time or pending a hardware interrupt.

When maintaining the timed delay and interrupt pending thread queues the thread manager utilises services provided by the Real Time clock and interrupt Service Layer shown in Figure 3. This layer acts as an interface to the board hardware and provides services which allow the thread manager to request that :

- a thread should be suspended for a given duration or until a specified interrupt occurs.
- the thread manager should be invoked after the duration has expired or the interrupt event has occurred.

When the thread manager is invoked the suspended thread will be transferred to the queue of runnable threads and priority based arbitration will take place to determine whether this thread should pre-empt any currently running thread.

3.2. Overview of the behaviour of the embedded kernel

The interaction between the the Pthread API and thread management layers of Figure 3 take place via a call to a single function:

kernel_suspend_and_call(thread_manager)

Functions in the Pthread API layer invoke this kernel function at all schedule decision points. The following events then occur:

- The context of the current thread is saved in a thread descriptor maintained by the kernel.
- The system stack pointer is switched to a dedicated thread manager stack
- The thread manager function is invoked.

For example, if a thread becomes blocked on a mutex the mutex_lock function enqueues the thread on the associated mutex queue. The function then invokes the kernel_suspend_and_call(thread_manager) function which causes the context of the running thread to be saved in an associated thread descriptor. The thread manager function is then invoked. The thread manager examines the queue of runnable threads and selects the thread with the highest priority. The context of this thread is then obtained from the relevant thread descriptor. This context is then restored and the thread resumes execution. Eventually the blocked thread will become unblocked when another thread calls the mutex_unlock function. At this point mutex_unlock will transfer the blocked thread from the mutex queue onto the kernel's queue of runnable threads. If the unblocked thread has a higher priority than the currently running thread then mutex_unlock will enqueue the running thread on the kernels queue of runnable threads and then invoke the kernel_suspend_and_call(thread_manager) function. As in the case of mutex_lock this causes the context of the thread to be saved in the relevant thread descriptor and the thread manager function to be invoked. The context of the thread with the highest priority is then restored by the thread manager (in this case the thread which became runnable due to the call to mutex_unlock) and thread execution is resumed.

The majority of the schedule decision points which occur within functions in the Pthread API layer can be dealt with via a single invocation of the thread manager. However, in the case of threads that are suspended for a given duration or pending an interrupt, two schedule decision points occur. Firstly, at the point of thread suspension and secondly when the time duration has expired or the interrupt occurrs. In the case of the schedule decision point which occurs at the initial thread suspension, this can be dealt with by the relevant function in the Pthread API layer invoking the thread manager via a call to the kernel_suspend_and_call(thread_manager) function. The thread manager must also be invoked at the schedule decision point at which the thread again becomes runnable as a result of a duration expiring or an interrupt occurring. In this case the thread manger is activated by the services in the real time clock and interrupt services layer. For example, in the case of a thread that is suspended due to a call to the nanosleep function the time in the future at which the thread should be awoken is calculated and the thread is enqueued on the kernel's time ordered delayed-thread queue. Nanosleep then invokes the kernel_suspend_and_call(thread_manager) function which causes the thread context to be saved and the thread manager function to be entered. The thread manager then examines the head of the delayedthread queue to determine the earliest time T_wakeup at which a delayed thread must be awoken. This time is then passed to the realtime clock service function. This function uses the MC68376 MCU's programmable timer interrupt (PIT) capability to schedule an interrupt to occur at T_wakeup. The interrupt service routine for the MCU's PIT interrupt is designed to save the context of the thread that is running when the PIT interrupt occurs and then invoke the thread manager. The thread manager will then transfer the thread at the head of the delayedthread queue onto the kernel's queue of runnable threads. If the delayed thread queue is not empty a further PIT interrupt is scheduled at the time required by the thread at the head of the delayed thread queue. After scheduling this PIT interrupt the thread manager examines the queue of runnable threads and restores the context of the thread with the highest priority.

Within the GNAT Ada 95 compiler, dynamically installed interrupt handlers are mapped to Posix signals and each signal handler is mapped to a Posix thread [5]. The thread is suspended pending the interrupt/signal by GNAT via a call to the sigwait function. Within the embedded system the sigwait function enqueues the calling thread on an interrupt-pending queue and invokes kernel_suspend_and_call(thread_manager) which causes the context of the thread to be stored in an appropriate thread descriptor and the thread manager to be invoked. The thread manager then restores the context of the highest priority runnable thread. In order for the suspended thread to be awoken when the interrupt occurs a small interrupt service routine (ISR) must be written to interface to the hardware. This ISR must :

- Perform any low level handling of interrupt configuration registers.
- Invoke the thread manager when an interrupt occurs so that the thread can be transferred from the interrupt-pending queue to the kernel's queue of runnable threads.

The process of invoking the thread manager is implemented using the real time clock services. The ISR first sets a flag in the kernel's data space which indicates the source of the interrupt. The wake up time of the PIT ISR is then set so that a timer interrupt will occur on the next clock tick. As in the case of the timer delay queue this causes the running thread to be suspended and the thread manager to be invoked when the PIT interrupt occurs. In this case the thread manager will examine the set of flags associated with possible interrupt sources and transfer the thread(s) associated with that interrupt to the kernel's queue of runnable threads. The PIT interrupt timer is then reconfigured so that the next scheduled PIT interrupt will occur at the time required by the thread at the head of the delayed-thread queue. Finally the thread manager restores the context of the thread with the highest priority on the kernel's runnable thread queue.

4. Timing Behaviour of Application Programs

To facilitate the experimental measurement of program execution times a high resolution timer has been implemented using the facilities of the configurable timer module of the MC68376 microprocessor. This timer operates at a frequency of 5.25Mhz giving a clock resolution of 190.5 nanoseconds.

The timing behaviour of application programs is determined by the execution time of the application code plus any overheads imposed when the GNULL layer of GNAT makes calls to the embedded kernel Pthread functions. Since these kernel functions behave in a predictable manner the overall timing behaviour of application programs is deterministic. To illustrate the predictability of the kernel behaviour we consider a simple example consisting of 5 periodic real–time tasks, 4 of which operate at equal priority and a single task that operates at a lower priority. The low priority task is included here simply to illustrate the kernel overheads associated with task pre-emption. The tasks execute a section of code and are then suspended via a call to the delay function. In this simple example we are interested in the worst case context switching times which result from task suspension (i.e. when a running task executes a delay function) and task pre-emption (i.e. when a high priority thread becomes runnable whilst the low priority task is currently executing).

In the case of task suspension, the worst case context switching time occurs when a high priority task is currently running, the low priority task is on the ready queue and the remaining high priority tasks are on the delayed–thread queue. If the high priority running thread executes a delay function which requires it to be placed at the end of the delayed–thread queue then this results in the maximum kernel activity during a context switch. This worst case context switching time is measured to be 130 micro seconds.

In the case of task pre–emption, the worst case context switching time occurs when the low priority task is currently running and all of the high priority tasks must be simultaneously removed from the delayed–thread queue and placed on the ready queue. The worst case context switching time in this case was measured to be 287 micro seconds.

5. Conclusions

Although it proved possible to port the GNAT system to a new platform the process was not entirely straightforward and required considerable detailed knowledge of both the GNAT and Pthread architectures. Nevertheless, the retargeting of the Pthreads facility proved to be less effort than doing a bare machine implementation of Ada. As we were not concerned with Pthreads itself support for a reduced set of Pthread API's has been implemented. We were also able to produce a time-predictable implementation of Ada that facilitated the use of Ada in a timecritical distributed testbed. To support the communications requirements of distributed applications we have also included support for the CAN protocol in our kernel. For pragmatic reasons the reimplementation of Pthreads has been undertaken in C. Our current aim is to reimplement this in Ada.

References

[1] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, pages 263 – 275. Springer Verlag, 1998.

[2] A. Burns and A. Wellings. Dual priority scheduling in ada 95 and real-time posix. In *Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming, WRTP'96*, pages 45–50, 1996.

[3] E. Giering, W. Mueller, and T. P. Baker. Implementing ada 9x features using posix threads:design issues. *TRI-Ada*, pages 93–103, Nov 1994.

[4] E. Giering, W. Mueller, and T. P. Baker. Features of the gnu ada runtime library. *TRI-Ada*, pages 214–228, Sep 1993.

[5] D. Oh, T. P. Baker, and S. Moon. The gnarl implementation of posix/ada signal services. *AdaEuro*, 1997.

[6] P. T. Woolley, W. M. Walker, and A. Burns. Developing a testbed for distributed real-time applications. In *Proceedings of the 24th IFAC/IFIP Workshop on Real-Time Programming, Saarland, Germany*, 1999.

Appendix 1

The experimental test bed includes 5 single board computers (SBCs) built around the Motorola MC68376 micro controller chip operating at a clock frequency of approximately 21MHz. This micro controller incorporates the following modules :

- CPU32 Module a 32 bit architecture central processor unit.
- CAN 2.0B Controller Module (TouCAN) a full implementation of the Controller Area Network (CAN) Protocol.
- Configurable Timer Module (used for the generation and monitoring of time related events).
- Queued Serial Module (used for serial communications with a local PC).
- Background Debugging Mode (BDM) Port (for use in conjunction with incircuit debugger software running on a local PC).

Each SBC is equipped with 1M bytes of user memory, 256K bytes of fast SRAM and 64K bytes of EPROM memory. The boards are connect (via the TouCAN module of the MC68376) to a CAN bus operating at 1 Mbps. The EPROM memory on each board holds a monitor program which allows software to be down loaded to each board over the CAN network. Software for the SBC's is developed on a PC which hosts a Linux MC68376 cross compiling version of the GNAT compiler. The development PC is also connected to the CAN network via 2 PCI mounted Intel 82527 CAN controller chips. One of these CAN controllers is used in conjunction with the monitor program running on each of the 5 SBC's for down loading software from the PC to the SBC's. The other CAN controller may be used either for monitoring traffic on the CAN network or as an interface to the CAN network for PC based user programs.