Development Of A Real-Time Simulation System

Standard off-the-shelf tools targeted for a wide variety of applications may not always meet the requirements of a particular type of application. Instead of simply following industry trends, embedded system developers should make a detailed analysis of the requirements of their application and buy and develop tools accordingly, as this case study points out.

Product designers are finding ever more things that can be done with embedded controllers to improve product performance, add features, reduce system cost, lower product liability risks, and aid compliance with government regulations.

For a complex system, such as an automobile or aircraft, the embedded control system is complicated by such factors as the need to incorporate more than just “control law” code in the controller’s application software. Another complication is the need to use a variety of intercoupled controllers so the response to a control command or an unusual operating condition may affect two or more controllers.

Product testing is a significant aspect of the development costs for complex systems. Detailed, real-time, closed-loop testing of the controller software needs to be performed over the entire system operating range to ensure both product performance and product safety. This testing needs to include fringe areas of operation and emergency situations. In the case of multiple, interconnected controllers, all interactions between or among the controllers need to be understood as unintended interactions that can adversely affect system performance.

How can this real-time, closed-loop testing of an embedded control system best be carried out? The approach used for over 40 years in the aerospace industry involves simulating, in hard real-time, the plant to be controlled, and interfacing this simulation to the controller. This type of approach is referred to as hardware-in-the-loop (HIL) simulation. The use of HIL simulation for testing embedded controllers is just coming to the forefront in other industries, particularly the automotive industry.

Reasons for using real-time, HIL simulation include:

- There may be no other feasible approach. An example of this situation is that, because a satellite is designed to work in the weightlessness of outer space, there is no practical way to test a satellite’s attitude control system on Earth other than to simulate the satellite dynamics and interface this simulation to the controller hardware.
- Control of the testing environment may be very difficult.
- Safety considerations may preclude performing tests on the system.
- Straightforward economic considerations may dictate simulation use.

continued on next page
Closely related to HIL simulation testing is rapid prototyping. For both rapid prototyping applications and testing embedded controllers properly, the HIL simulation studies must be performed in hard real-time. These studies often have a very demanding numerical integration frame time requirement of 0.5 to 1.0 ms if adequate dynamic accuracy is to be achieved. In 1990, Applied Dynamics International (ADI) began the development of a new system to replace an older, special purpose, proprietary product which was designed in the early-1980s specifically for real-time simulation.

This new system was named the Applied Dynamics Real-Time Station or AD RTS. We were closely involved in the design of the new system, particularly its software. The AD RTS has evolved through a number of versions to its present form. All major design goals set initially for the AD RTS have been achieved, but not without a lot of learning.

The AD RTS is a multiprocessor system with multi-language (Fortran and C) support that is capable of microsecond-level determinism. An open, standards-based system architecture is used. The system software provides real-time bridges to various popular modeling languages like EASY5, SIMULINK, and SystemBuild.

Although most of the requirements of the AD RTS were known at the start, all major design goals set initially for the AD RTS have been achieved, but not without a lot of learning. The AD RTS was to be a tool for developing real-time, HIL simulations. Ease of use and programming were important, and could only be achieved if the software tools were very tightly integrated. The software development and experimentation environment needed to include both non-intrusive debugging (the capability to monitor the execution of the programs without stopping their execution) and intrusive debugging (such as setting breakpoints...
Simulation System Development

... and single stepping). Being a multi-
processor system, a framework had to be provided for integrated multiproces-
sor debugging. A software environment that allowed the automation of tests comprised of multiple runs with parameter perturbations was required.

The AD RTS was to be capable of operation over a LAN so engineers could access the system remotely. High performance D/A, A/D, and Digital I/O devices had to be supported. Interfaces to popular industry standard devices and buses such as MIL-STD-1553B, IEEE-488, ARINC-429, RS 422, HSD and DRII-W were essential for the success of the AD RTS. For hardware-in-the-loop simulations, it was important to provide arbitrary function generation modules, and hardware emulators for transducers such as LVDTs, thermocouples, and RTDs.

Finally, it was essential to be able to produce a family of real-time products ranging from a uniprocessor system supporting simple I/O devices to a multiprocessor system supporting complex I/O requirements.

**HARDWARE ARCHITECTURE SELECTION**

Based primarily on the requirements that the AD RTS be open, standards-based, and have high performance, the VMEbus was selected as the backplane for the system. The VMEbus was already widely accepted and there was a broad range of third-party products available for it.

The system developed has a Unix host, one or more real-time processor boards, and both intelligent and simple I/O boards plugged into a VMEbus backplane. The mathematical models or real-time programs are executed on the real-time processor boards. The Unix host is used for program development and interaction, but is isolated from time-critical activities. ADI purchases many off-the-shelf processor and I/O cards and manufactures some high-performance I/O and communication cards in-house.

**BASIC SOFTWARE ARCHITECTURE SELECTION**

The first software architecture question considered was whether to go with a self-hosted approach or a host/target approach.

Eventually, a host/target approach was chosen. In a host/target architecture, the code that executes on the real-time processors is developed in a non-real-time host and then downloaded to the target processors for execution. Host services, such as access to a sophisticated file system and network services, are available to target processors via service requests to servers on the host.

With a host/target architecture it is much simpler to implement deterministic behavior. Real-time programs running on a target processor have precise timing requirements, but they do not require most of the services offered by a general-purpose operating system (OS). Hence, in a host/target system the code executing on the target board can execute on top of a relatively simple real-time kernel. In such a system, the user has fine grain control over the code executing on the target processor and does not have to overcome the background actions of a complex OS. Even the real-time Unix vendors have started using a host/target approach in which a full Unix environment is provided for the host processor and a stripped-down version of Unix is provided as the RTOS for the target processors.

With a host/target architecture it is also simpler to develop a heterogeneous system. The kernel needed for a target processor is fairly small and can easily be ported to any processor without much effort. Further, the small size of the kernel allows it to be supported on boards with minimal resources. Currently, ADI's real-time processor boards are based on MC68040, MC88110, AM29000, and PowerPC 604.

**REAL-TIME KERNEL CONSIDERATIONS**

Having decided on the basic architecture, the next task was to investigate the real-time kernels available in 1990-91. Criteria to be satisfied included performance, availability for the processors to be used, support for Fortran, and ease of use in a multiprocessor environment where simplicity of interprocessor communication is important.

While many real-time kernels were available for Motorola’s 68K series and Intel’s 80X86, they were not available for the higher-performance processors under consideration for the AD RTS. Also, no real-time kernel was found that was available for all the processors to be supported in the AD RTS. Another problem was that most of the real-time kernels did not support Fortran, at least for the processors under consideration. Fortran support was very important because many ADI customers have validated simulation math models developed in Fortran.

In experiments performed with a commercially available real-time kernel, features offered by the kernel were often more in the way than useful for typical applications to be run on the AD RTS. The main reason for this complication was that in high-performance, hard real-time applications, scheduling is inherently a part of the application program. A priority-based kernel is not an appropriate base on which to layer a user-supplied scheduler efficiently. For example, one experiment had three tasks, T1, T2, and T3, which were to be scheduled cyclically such that T1 always ran to completion, T2 was to be allotted 200 μsecs, and T3 was to be allocated 100 μsecs. The experiment was performed on a MC68030-based board running a standard real-time kernel. The context switch overhead was documented to be about 70 μsecs. However, to force the required context switches with the available system calls, the overhead was closer to 140 μsecs per context switch.

*continued on next page*
Simulation System Development

The method of handling interprocessor communications (IPCS) in the standard real-time kernels considered was also of concern. The IPC area was one that had to be kept as simple as possible. After all, the main concern of ADI’s users is with their applications, not with solving software implementation problems. The use of name servers and global objects for interprocessor communications was considered more complicated than something a typical customer would want to confront. Software could have been developed to simplify this aspect of programming, but this would have involved a significant development effort, and would likely have added run-time overhead.

Some of the popular control system development packages, such as SIMULINK and SystemBuild, generate code with a scheduler or a partial scheduler (generally a rate monotonic scheduler) embedded in it. In these cases the real-time kernel’s scheduler is of little use except to slow down the application. Primitive functions that simplify the support of code generated by these packages are required.

In summary, off-the-shelf real-time kernels would have been of little use, except possibly for their code development and program loading tools. So ADI did what the experts advise you not to do—roll your own tools. However, this procedure was not as difficult as it would appear, because ADI had developed similar products and had in-house expertise for building these tools. The authors believe that the overall integrated system that was developed is much better and easier to use than any system that could have been patched together by integrating tools from various vendors. Developing these tools in-house has provided another important benefit-time to market. When a new microprocessor is announced that ADI wants to incorporate in the AD RTS, it’s not necessary to wait for the kernel vendors to support it before the AD RTS can be upgraded.

COMPONENTS OF THE SOFTWARE SYSTEM

Having selected the VMEbus as the basis for the hardware architecture, the host/target approach to the software architecture, and having decided against the use of third-party real-time OS packages, software development began. The software system developed has the following components:

- An environment for program development
- An integrated program monitor/control/debug environment
- Target processor run-time system
- Graphical data visualization and run-time control tools

Before describing these various components it is useful to discuss briefly one aspect of software development that cut across these components and relates to the host/target nature of the software architecture.

Given that a host/target model had been selected, a major issue was keeping non-time-critical activities occurring in the Unix environment from interfering with the time-critical activities of the target part of the system at run-time. This issue was handled by not allowing the Unix host to access the VMEbus after the target program started executing. One of the real-time processors, designated as an interact processor, interacts with the Unix host during run-time. The interact processor deposits data requested by the host system in a FIFO in the memory of the host processor. In the other direction, commands issued interactively by users are deposited in buffers in the host. These buffers are accessed under program control by the interact processor. This communication scheme between the host and the interact processor ensures that the VMEbus can be used deterministically by the target processors.

PROGRAM DEVELOPMENT ENVIRONMENT

The program development environment provides an overall framework for developing software that executes in multiple processors. The primary components of the program development environment are a configuration system, an applications development shell, and cross compilers. Typical data about a board includes the VMEbus addresses of the board, the local address of the memory on the board, interrupt level and interrupt vector for interrupts raised by I/O boards, and the device driver name for an I/O device.

The applications development shell provides an environment for building a multiprocessor application on the system. While the configuration system is used to describe the configuration of the system hardware, the applications development shell is used to describe the configuration of an application running on the system. Its specific uses are the following:

- Specifies the programs that execute on the different processors
- Specifies the devices controlled by each processor
- Declares interprocessor communication devices
- Provides initialization code

A typical applications development shell script is shown in Listing 1. The processor declaration specifies the processors from the hardware configuration used in this application. After the processors and devices are declared, the schedule section provides the call to the root thread in each processor. The shmem and pipe declarations specify the size and location of shared memories and pipes, such as the shmem declaration in the following example:

shmem shi(100): (pi,p2,p3) in p1

continued on next page
Simulation System Development

This code declares shl as a shared memory of size 100 bytes that can be accessed from processors p1, p2, and p3 and forces the physical location of shmem to be the processor board, pl. This specification of the physical location of the interprocessor communication devices gives the user precise control over the VMEbus usage.

Any device or interprocessor communication device declared in the applications development shell can be opened and accessed from the processors that have been specified in the declaration of the device. The access functions are based on Unix device driver calls.

The applications development shell in Listing 1 generates initialization code, a makefile, device driver tables, tables for interprocessor communication devices, and a global symbol table. Users fill in macros and rules to build their application C and Fortran programs for the target processors in the makefile. The makefile is then invoked to generate code for each target processor and to link the global symbol table. The global symbol table is used by the interactive utilities.

The cross compilers that form part of the program development environment are Fortran and C compilers that run on the Unix host and generate compiled code that executes on the target processors.

PROGRAM MONITOR/CONTROL/DEBUG ENVIRONMENT

Our primary tool for downloading the generated code to the target processors and interacting with the target processors is called Interact. This tool is multipurpose and allows users to:

- Download the nanokemel and application code to each of the target processors
- Set up tables in each target processor for non-intrusive monitoring of program execution
- Control program execution in the target processors by allowing target programs to be started, suspended, and halted
- Symbolically access all variables in each target processor
- Have multiprocessor debugging capability, with support for standard debugging features such as trace-back, breakpoints, and single stepping
- Employ a powerful script language for automating test sequences and also for developing custom user interfaces with menus and buttons
- Act as the server for the target nanokemels, so the target processors can access the services of the host operating system, particularly the host’s file system. Typically, user applications initialize themselves by reading files from the host.

TARGET PROCESSOR

RUN-TIME SYSTEM

The run-time system for each processor includes standard libraries, device driver libraries, thread libraries, and the nanokemel. The run-time system was specifically designed to achieve a very high level of determinism and to be extremely simple to use. An important feature of the run-time system is that it provides the basic services required for developing different schedulers.

Broadly speaking, in the software architecture, each processor in the AD RTS executes a process with multiple threads of execution. The threads running on a given processor board communicate with each other via global memory. Threads running on different boards communicate with each other via shared memory and pipe devices. The interprocessor communication devices (shared memory and pipes) are declared and created statically during program development on the host. With the AD RTS, in contrast to most real-time kernels, name servers and interrupt processing are not involved in creating and using the interprocessor communication devices.

LISTING 1

Example of applications development shell script.

```c
/* Declare processors pl, p2 and p3 from real-time station AD RTS1 for use in the application */
processors pl = "AD RTS1.pl",
p2 = "AD RTS1.p2",
p3 = "AD RTS1.p3"

/* Declare the devices to be used in the application */
device dl = "AD RTS1.d1",
d2 = "AD RTS1.d2"

/* Declare shared memory and pipes for interprocessor communication. */
shmem shl(100) : (pl,p2,p3) in pl
shmem shl2(100) : (pl,p3)
pipe pipel(IODO) : (pi,p2) in pl
pipe pipe2(1000) : (pl,p3) in pl

schedule sample_program
<<pl,p2,p3>>
section
/* Call the main of the root thread in each processor */
prg_main ()
end section
end schedule sample_program

/* If no processor location is specified, a default location is provided. */
```

More specifically, the structure of the run-time system is illustrated in Figure 1. The run-time system for each board is layered on a nanokemel. The nanokemel provides the following basic services:

- Creates a thread
- Starts the execution of a thread
- Maintains the register context of threads
- Performs context switches
- Allows the application threads to access the host
- Provides all communication with the host
- Provides basic interrupt handling
- Maintains breakpoint tables for the threads

continued on next page
Layered on top of the nanokernel is a thread library. The thread library allows an application to create higher level threads, start the execution of another thread from a thread, and start the execution of a thread from any user supplied interrupt handler. Whenever a thread is started by another thread or an interrupt handler, the execution of the executing thread is suspended, and its thread identifier is pushed onto a thread stack. When a thread completes its execution, the execution of the thread at the top of the thread stack is resumed. Functions are provided to manipulate the thread identifier stack (rearranging the stack if necessary).

The thread library makes it extremely simple to build schedulers. A rate-monotonic scheduler is layered on top of the basic thread library. The code generated by two popular control system development packages, SIMULINK and BEACON, executes on top of the thread library. The run-time system for the code generated from SIMULINK, the control system package from MathWorks, was layered on the thread library.

The code generated by BEACON, a graphical tool that generates production quality code from control system block diagrams and control flow diagrams, is also layered on top of the thread library.

The functions of each layer of the run-time system are documented and available to the user program. Application programs can interact with any layer. If an application is single threaded, it interfaces directly with the nanokernel via the application and device driver library. In this case the program running on a target board is never interrupted. The code generated by Boeing’s modeling package, EASY5, is supported using this approach. ADI’s modeling package, ADSIM, is supported in a similar fashion.

It is also possible to support multi-threaded systems directly on top of the nanokernel. For example, Integrated System Inc.’s SystemBuild/AutoCode package generates code with a rate monotonic scheduler and thread abstraction embedded in the code. The C code generated by AutoCode is supported directly on top of the nanokernel by providing a timer interrupt handler and some support code.

**GRAPHICAL DATA VISUALIZATION AND RUN-TIME CONTROL TOOLS**

Two graphical tools were developed for use with the AD RTS. One of these tools is a high speed, real-time, graphical plotting package with many features including both scrollable strip-chart displays with dynamic autoscaling and X-Y plot capability. It is useful for real-time simulation. The other tool is a graphical control and monitoring package, based on DataViews, that allows real-time collection and display of data in terms of graphs and dynamic objects such as dials, gauges, and so on, and provides run-time control via graphical objects such as buttons and sliders.

ADI’s philosophy has been to provide the primitives required for developing a real-time application. One of the most interesting things that has come out of our development activities is that the software tools and run-time package developed for the AD RTS appear to be directly applicable to the related real-time area of embedded microprocessor controller application development.

John Boyd manages the software development group at ADI and was one of the primary software architects of the AD RTS software. Boyd holds an MBA from the University of Michigan, an MS in computer science from Purdue, and a BS in computer science from Michigan State University.

Roger Theyyunni is staff engineer at ADI, where he designs and develops compilers, real-time kernels, device drivers, and other tools for real-time programming. Theyyunni holds MS degrees in computer science and chemical engineering.