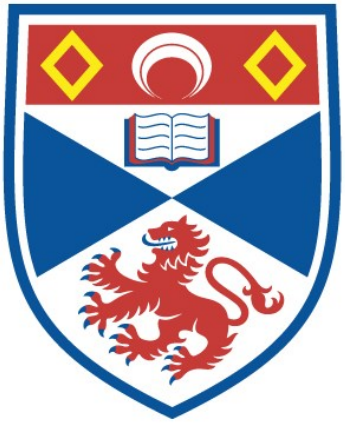
A TESTBED FOR EMBEDDED SYSTEMS

Peter Burgess

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1994

A Testbed for Embedded Systems

Peter Burgess
PhD Thesis
University of St Andrews
Division of Computer Science
Department of Mathematical and Computational Sciences
University of St Andrews
St Andrews, Fife, KY16 9SS

June 17, 1994

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

Testing and Debugging are often the most difficult phase of software development. This is especially true of embedded systems which are usually concurrent, have real-time performance and correctness constraints and which execute in the field in an environment which may not permit internal scrutiny of the software behaviour. Although good software engineering practices help, they will never eliminate the need for testing and debugging. This is because failings in the specification and design are often only discovered through testing and understanding these failings and how to correct them comes from debugging. These observations suggest that embedded software should be designed in a way which makes testing and debugging easier and that tools which support these activities are required. Due to the often hostile environment in which the finished embedded system will function, it is necessary to have a platform which allows the software to be developed and tested "in vitro".

The Testbed system achieves these goals by providing dynamic modification and process migration facilities for use during development as well as powerful monitoring and background debugging support. These facilities are built on a basic run-time harness supporting an event-driven programming model with a global communication mechanism. This programming model is well suited to the reactive nature of embedded systems. The main research contributions of this work are in the areas of finding deadlock-free, path-optimal routings for networks and of dynamic modification with automated conversion of data which may include pointers.

(i) I, Peter Burgess, hereby certify that this thesis, which is approximately 60,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

date 6/5/94 signature of candidate

(ii) I was admitted as a research student under Ordinance No. 12 in November, 1990 and as a candidate for the degree of PhD in November, 1990; the higher study for which this is a record was carried out in the University of St Andrews between 1990 and 1994.

date 6/5/94 signature of candidate

(iii) I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 6/5/94 signature of supervisor

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date 6/5/94 signature of candidate

Acknowledgements

I would like to thank, primarily my supervisor Mike Livesey for suggesting I come to St Andrews to do a PhD in the first place, for his initial motivation and numerous suggestions and guidance along the way and for sharing the invaluable insights he gained from his work with the original ROV project. I would also like to give special mention to the following people:

Colin Allison, who often provided practical comments during the many Testbed design sessions based on his experience in the systems field and who read the first draft of this thesis at extremely short notice and gave many helpful suggestions for improvement.

Gerald Ostheimer, who has had to share an office with me for three years and whose revolutionary work on abstract architectures for parallel processing provided much inspiration for the Testbed programming model. I am also indebted to him for sharing his hypertext bibliographic database system, for reading and providing very useful criticism of an early version of the dynamic modification chapter and for broadening my knowledge of Computer Science considerably through numerous discussions.

Duncan Matthew, for providing me with source code and very thorough documentation of his hexapod walking robot, enabling me to implement it using the Testbed and so gain confidence in the programming model.

Tony Reynolds, for supervising a short term research fellowship at BT which provided early insight into the use of real-time clocks in distributed testing.

Finally, I would like to thank my brother, Richard, for getting me into computing in the first place.

Contents

Chapter 1

Introduction

Embedded systems are computer systems which are embedded in larger systems and running a single custom application program. Examples range in complexity from systems with a single processor such as video recorders, and disk controllers through more complex systems requiring more powerful processors such as data communications boxes to very complex distributed multiprocessor systems such as chemical plant controllers, robots and space craft control systems. An embedded system can generally be decomposed into a number of semi-independent state machines or processes each of which may have the following common characteristics:

Reactive The process spends most of its time idle, awaiting some event, which may be external (from the outside world) or internal (a message or request from another process). It must then perform some action, possibly within some deadline and return to the idle state.

Periodic The process repeatedly performs the same task at set intervals of time. Each instance of the task may have a deadline relative to its scheduled start. This can be regarded (and will be in the system described) as a special case of a reactive process in which the triggering event is a time.

Tightly coupled to hardware and environment Each embedded system has special requirements which depend on the hardware which is to be controlled (actuators) and the interface to the external environment (valuators). The presence of the external environment in the specification of the system results in complex state spaces which are difficult to predict and model.

Concurrent In all but the simplest systems there will be more than one distinct subsystem and hence process. These processes will need to communicate, with the resulting increase in complexity. In addition each process may well

have to accept new external events while previous ones are still being dealt with.

1.1 Example of an Embedded System: The ROV

As a motivating example consider a submersible Remotely Operated Vehicle (ROV), typically used for inspection of off-shore oil platforms. The target system consists of a submersible robot attached by an umbilical cord to a surface controller as illustrated in Figure 1.1. The vehicle has four motors; two at the rear and two at the side each of which can be driven individually in either direction. The rear motors provide forward or backward thrust and heading control while the side motors which are inclined at 45 degrees, provide both vertical and sideways thrust. Buoyancy tanks at the top of the vehicle prevent significant tilting when the side motors are driving at different speeds.

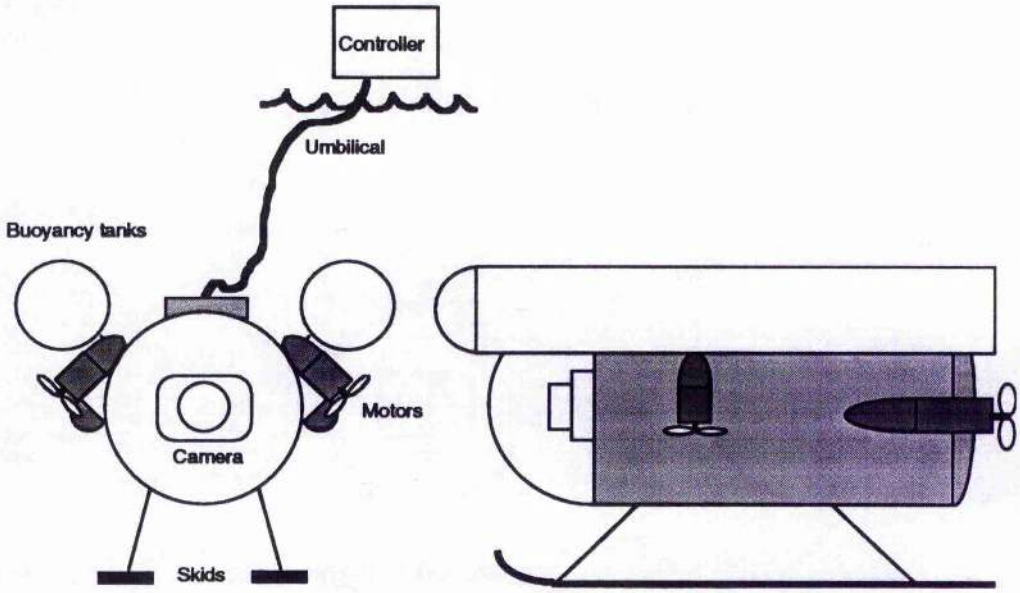


Figure 1.1: The submersible ROV.

The ROV is controlled by an operator at the surface who uses visual feedback from a closed circuit video camera mounted on the front of the vehicle and a 20 character LCD display which displays depth and heading reported back by the software from gauges in the vehicle. The control panel includes a joystick for controlling the horizontal motion (speed and direction) of the ROV relative to the current heading, by the position of the joystick, and the heading itself, by twisting the grip of the joystick. The heading control returns to a neutral position when released. There are buttons for driving up or down at a fixed speed and for selecting autodepth and autoheading control. There is also a joystick for pan and

tilt of the camera and buttons for focusing. Figure 1.2 shows the operators view of the ROV.

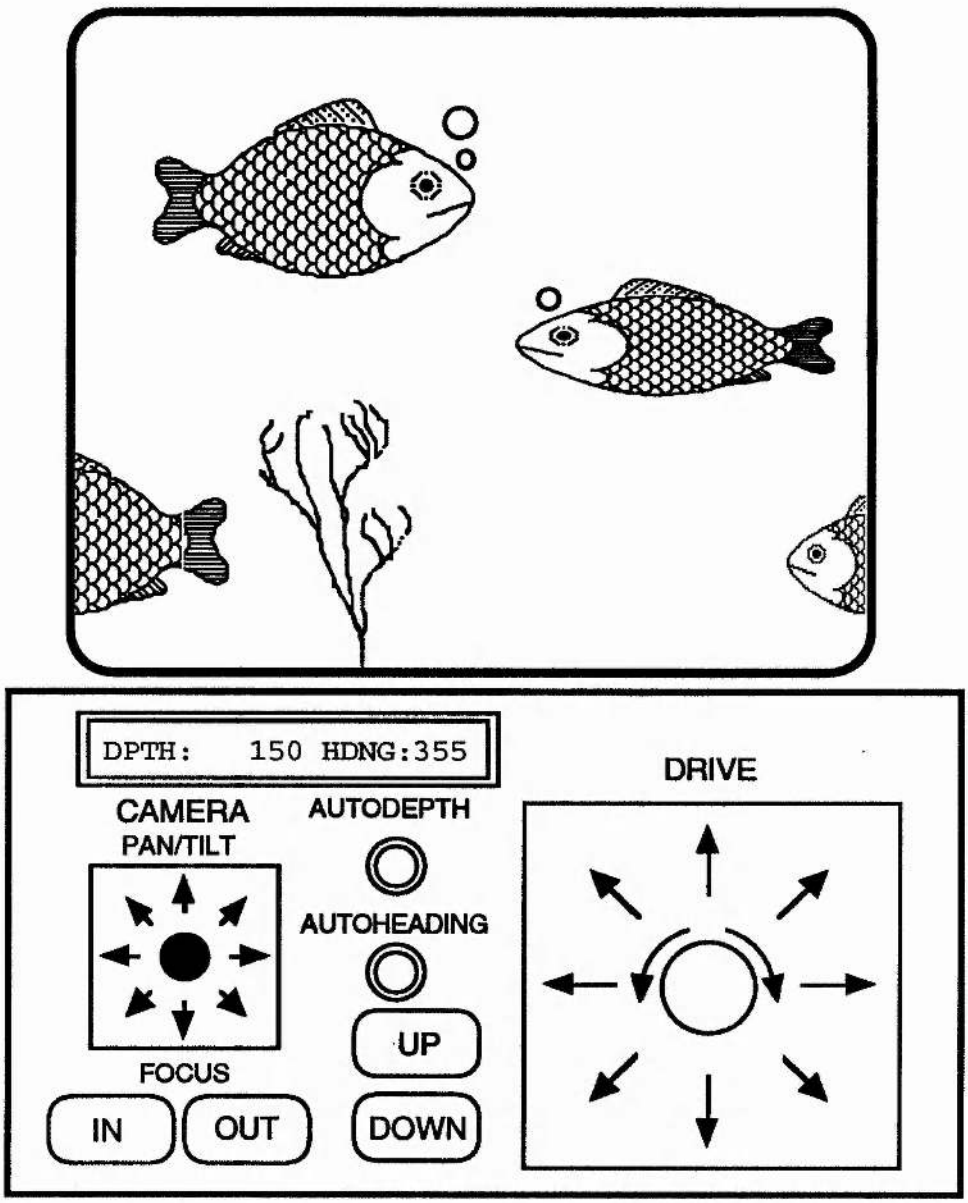


Figure 1.2: Operators view of the ROV.

While autodepth (autoheading) is selected the control software attempts to maintain depth (heading) at the value reported by the depth gauge (compass) when this mode was entered. Autodepth control is temporarily overridden while one of the up or down buttons is depressed and restored when released with the target depth reset to the current depth. Similarly autoheading is temporarily overridden when the drive joystick is twisted to turn left or right and restored when it returns to its normal position with the target heading reset to the current compass value.

The software is required to ensure the following safety conditions to avoid

damage to the hardware:

1. There is an upper bound on the rate of change of speed of each motor.
2. Each motor must be brought to a stop for a short period before changing direction.
3. There is an upper bound on the sum of speeds of the two motors on each side of the vehicle.

The software is divided into two layers. The low level control performed onboard the vehicle consists of switching each motor (including those which control the camera) on or off for a percentage of a duty cycle specified by the high level control software which resides in the surface controller. The onboard software also polls the depth and compass gauges periodically and reports the values back to the surface controller. The high level controller is responsible for converting operator commands into appropriate motor values and for the autodepth and autoheading control while maintaining the safety conditions, as well as displaying the reported depth and compass values.

The original version of the ROV control software was implemented in Z80 assembly language with the only software testing done on the vehicle. Any behavioural faults meant hauling the robot up to the surface, trying to debug the assembly language program, re-assembling the new control code, burning new eproms, and trying again. In Section 2.10 an implementation of the ROV in the Testbed programming model described in Chapter 2 is presented.

1.2 Approaches To Embedded System Construction

It might at first seem that as embedded systems are concurrent, they are a special case of general purpose distributed operating systems or perhaps should be studied along with supercomputers. However the first three characteristics on the list on page 1 conflict with the goals of these fields. General purpose computers tend to run multiple competing tasks and are tuned to provide good average throughput. They often have features which embedded systems do not require such as virtual memory and disk/file systems. The maximum memory requirement can usually be determined during an embedded system's development and the extra cost in overhead, loss of predictability and extra power and space requirements often rule out virtual memory and disks. The applications run on general purpose computers and supercomputer applications tend to differ from embedded applications in that they run to completion with as few pauses as possible. If the system fails due to shortage

of resources, it may be acceptable simply to reconfigure and restart the application or even reboot the system if the fault lies there (e.g., memory fragmentation). This is generally not acceptable in an embedded application/system. Most work on embedded systems appears in the real-time systems literature, however there are other important issues which deserve study such as the general development and debugging problems which this work addresses.

Two approaches to embedded system construction are to program the hardware directly or to use an operating system. The advantages of the former are that it is theoretically possible to minimize hardware requirements such as memory and processor performance. This is probably appropriate for the simplest embedded applications where these resources are most limited. The main disadvantage of the direct approach is high software costs; the code is likely to be hard to understand, debug and modify. These become rapidly worse as the complexity of the application increases. Operating systems have the advantage of code reuse, the application programmers task should be easier and the resulting code should be simpler, as much detail is hidden in system calls. The operating system may well provide facilities and tools which make the system easier to develop, debug and modify, such as the ability to test outside the target environment using simulations of the real system. The disadvantages of an operating system include the excess baggage in terms of memory and processor performance requirements, possible loss of control of resource management and the fact that the applications programmer may not be able to obtain detailed information about the behaviour of the system and the consequences/side-effects of system operations. This last difficulty makes verification of the system difficult and also hinders debugging as the programmer may be unsure of whether a bug is in the system or application code. Often such cases arise from a lack of understanding of the operating system's behaviour due to poor documentation.

The difficulties encountered with operating systems often result from the use of a general purpose operating system (such as a Unix™ variant or MSDOS™). These systems have not been designed with embedded applications in mind and consequently do not have the required properties such as determinism or level of documentation. Often compromises have had to be made due to the lack of advance information about the application, in order to improve average performance at the cost of occasional unpredictable delays or even potential failure. Even the basic process model supported by such systems is often inappropriate in that it requires the application programmer to provide an event loop which is common to all processes and which might well have been included in the system. It is also likely that the system is not customizable to a fine enough level, requiring

the programmer to work around inappropriate low level mechanisms which add unnecessary overhead. For example it is often not possible for the application to control low level scheduling or inform the system of its timing constraints. The microkernel approach currently in vogue in operating systems design (Amoeba, Mach 3.0 [97]) helps to some extent, however most of these systems have been developed with different objectives to embedded systems operating systems such as implementing a better (e.g., distributed, object oriented) Unix.

An operating system designed specifically for embedded applications should overcome most of these difficulties. Obviously due to the highly disparate nature of embedded applications, it is hard to produce an operating system which is ideal for all and there will always be resource penalties to pay for using one. However for complex embedded applications the benefits should outweigh the costs.

In addition to the attributes of predictability, simplicity and configurability of the kernel, an operating system for embedded systems can also provide higher level features which aid in the development process. These are outlined in the next section which overviews the goals of the Testbed project.

1.3 Goals of the Testbed

Monitoring It is essential that both the whole system and individual running processes can be monitored during their operation. Monitoring inevitably causes perturbation of the target system (called the "probe effect" in [28, 40]). It is important to minimize this effect, and allow the test bed to eavesdrop on the target system as silently as possible. Embedded systems are almost always time-critical. Timing is one of the most delicate aspects of the target system behaviour, making silence a difficult criterion to meet and effectively ruling out interpretive debugging.

Dynamic Modification If a malfunction or undesirable behaviour is detected during monitoring, subsequent modification of the system incurs a cost. For any large system, such as a telephone exchange, airline reservation system or operating system, this cost may be unacceptable. Minimizing the potential cost requires the system to support some degree of dynamic modification. An embedded system will normally have a large range of internal states, reflecting the system's close coupling to its external environment. The state path from startup to the state where the malfunction is detected may therefore be long. The cost of re-creating this path, the "warm up" cost, may be prohibitive; in some cases it may be effectively infinite. It is therefore highly desirable to have dynamic modification at the level of individual

processes.

Process Interconnection In order that a client application interconnection topology can be designed independently of the physical network topology, it is necessary to provide a communication harness that supports a single system wide communication model. This model must provide, as simply and efficiently as possible, deadlock-free, versatile and reliable routing of messages.

Process Migration Time criticality makes the allocation of processes to processors an important issue, and the test bed must also provide for dynamic control over this allocation by allowing processes to be migrated. Migration is another reason for having the system wide communication model. Migration can be viewed as a form of dynamic modification.

Background Debugging Interactive debugging creates a bottleneck, which causes excessive intrusion and contributes to the probe effect mentioned above. One way to minimize the interaction bottleneck is by means of background debugging in which debugging decisions and actions are devolved as far as possible to code which runs alongside the application software.

1.4 System Overview

Testbed is currently implemented on a Meiko Computing Surface connected to a Sun workstation. The reconfigurability of the Computing Surface makes it easy to experiment with different interconnection topologies for multiprocessor embedded applications and also provides extra processors which can be used for simulating devices and off-loading monitoring tasks (as suggested in [1] and [33]), reducing interference which might otherwise change the behaviour of the application.

The user interface to the system is a Sun workstation (hosting the Computing Surface) through which the user is able to interact with the application under development through a root node for purposes such as monitoring, debugging and controlling simulation (see Figure 1.3). All communication with the host system uses Meiko I/O functions from the root node. Communication from elsewhere in the system is performed strictly using Testbed functions which communicate over transputer channels with system components which route messages back to the root node.

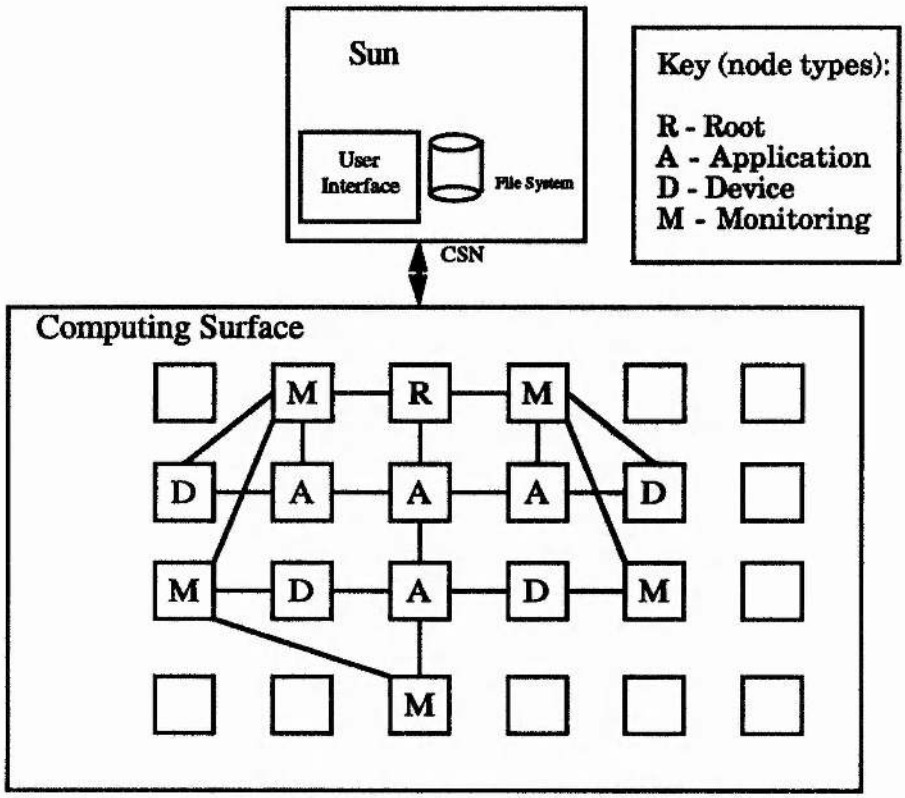


Figure 1.3: The Testbed development platform supports "in vitro" development with extra nodes used for monitoring and simulating external devices.

1.5 Related Work

1.5.1 Embedded Systems Development Environments

A number of test beds and development environments for embedded systems have appeared in the literature recently, most with some features in common with Testbed. However none have all the features described in this thesis. In particular dynamic modification facilities are either absent or much more primitive than those described in Chapter 7.

[83] describes testing distributed real-time systems in the context of the MARS architecture and MARDS design environment. The MARS system consists of a set of clusters of *components* which are stand-alone computers. Each component in a cluster communicates with the others using a real-time bus while some components can be connected to an *interface bus* or *field bus* for communicating with other clusters or the external environment. Each component runs a copy of the MARS operating system. Apart from the architectural differences, MARS is fundamentally different from the Testbed system and programming model in that MARS is a *time-triggered* system in which each action is executed at a predeter-

mined point in time. By contrast, Testbed is an *event-triggered* system in which the scheduled start time for a time-dependent action is an event.

Similarities in the work described in [83] include the use of passive monitoring rather than interactive debugging to avoid the probe effect and the use of extra processes and processors for environment and device simulation.

[33] describes testing in a "host" environment with device handlers replaced by device simulators for testing the logical function of the system and also in the target system with an environment simulator for testing performance. The environment simulator runs on an external system to avoid perturbing the software under test.

1.5.2 Concurrent Debugging

The same techniques which are used for debugging sequential programs may be used to debug concurrent ones, both to detect the same types of bugs and also new ones introduced by communication, such as sequencing errors. In the latter case the efforts of the debugger are often thwarted by the intrusiveness of the debugging process—the probe effect, in which the very presence of the debugger alters the timing properties of the program to mask possible errors. Detecting the special errors which are introduced due to concurrency, and avoiding the probe effect, have led to approaches which concentrate on specific aspects of debugging, or one technique (such as message monitoring).

[60] surveys techniques for debugging concurrent programs and identifies four classes:

1. Traditional debugging techniques.
2. Event-based debuggers.
3. (Graphical) tracing techniques.
4. Static analysis.

Here debugging techniques will be divided into the following (non-mutually exclusive) categories:

Interactive Debugging In which the user may view and interact with execution of the program in real-time.

Monitoring In which the events/states in the system are captured, and viewed/used either in real-time or post-mortem.

Replay In which sufficient information is recorded to enable a program to be replayed from a particular point, with more detailed examination.

Static Debugging Where a specification of the program is checked against certain assertions about its valid behaviour.

Background Debugging In which monitoring techniques are used in conjunction with the assertion idea from static debugging to automate the process of interactive debugging.

1.5.3 Interactive Debuggers

Conventional source level debuggers are a very useful tool in software development. Unfortunately many systems which support and even encourage concurrent programming, fail to provide interactive source-level debuggers which support multi-threaded programs. For example the dbx and dbxtool debuggers in SunOS and the tdb and tdbtool debuggers for the Meiko Computing Surface cannot cope when a process forks (or PARs), although there have been interactive windows-based debuggers for some time [27, 40, 48, 80, 89] which do support multithreaded programs.

1.5.4 Replay

In replay systems [16, 17, 24, 31, 54, 69], the communications which the process under inspection makes with the outside world are recorded. When some incorrect behaviour is noticed, the system is stopped and then the process is "replayed" with the monitored communications performed in the same sequence. This allows full source level debugging with breakpoints and state inspection to be performed without the danger of changing the temporal ordering of events. Replay systems have two major disadvantages for embedded systems. The first is that embedded systems run for an indefinite amount of time. Thus if all communication events are logged, the log may grow indefinitely. Many replay systems cope with this by including periodic checkpointing, allowing the log to be discarded prior to the checkpoint. However any behaviour which the monitoring system (or human observer) is trying to detect must be guaranteed not to have occurred before the checkpoint or the subsequent replay will not be able to identify the root cause. Another problem is that checkpointing is a time consuming activity and may itself interfere with the temporal or timing behaviour of the system. The other major problem with replay is that it only preserves the temporal ordering of the events in the system. Timing behaviour will still be altered by any source level debugging during the replay.

1.5.5 Static Debugging

This approach generally involves converting a high level specification into a graph, such as in the Petri net approach [21] which may be analyzed to detect potential invalid communication patterns such as deadlock. Unfortunately for realistic systems the graphs quickly become unmanageable.

[41] describes a method for testing a set of communicating sequential processes, by attempting to construct a feasible sequence of interactions between them from a given set of inputs by an iterative algorithm which initially assumes random inputs to each process. This approach may be suitable for short processes of finite duration, however the artificial time ordering of the communications used may not reflect what happens in the real system.

The research described in this thesis has focused on providing mechanisms to facilitate non-intrusive debugging which is not possible with source level, breakpoint techniques. Consequently Testbed does not provide these facilities. A survey of work on non-intrusive monitoring and background debugging can be found in Chapter 6.

1.6 Thesis Structure

The rest of this thesis is structured as follows: Chapter 2 describes the programming model around which the Testbed is built. Examples are presented to illustrate the use of this model.

Chapter 3 gives details of the higher levels of system software which provide the debugging support and implement the programming model.

Chapter 4 describes the design of the BED (Basically Event Driven) multithreaded kernel which supports the Testbed programming model.

Chapter 5 presents work done in support of Testbed on deadlock-free routing in networks.

Chapter 6 describes the monitoring features of Testbed and discusses how they may be used to debug and tune an embedded application without interfering with the application's behaviour.

Chapter 7 contains a formal treatment of the solution to the pointer updating problem as well as a general description of the mechanisms which support dynamic modification in Testbed. A simple example is provided which illustrates how dynamic modification may be used in practice.

Chapter 8 describes the process migration mechanism of Testbed and illustrates with an example how it can be used to improve load balancing.

Chapter 9 summarizes the project and its contribution.

Appendix A gives a detailed description of the user's view of the current Testbed implementation, Appendix B gives details of system port assignments and Appendix C gives source code for the examples described in the main body of the thesis.

1.7 Original Contribution

The original aspects of this work include:

1. The programming model (Chapter 2), which differs from the CSP style commonly used for distributed memory systems and is better suited to the reactive nature of embedded systems.

2. The integration of time into scheduling (Chapter 4).

3. Work on optimizing message routing while preserving deadlock-freedom and extending networks with such routings in a regular way (Chapter 5).

4. Monitoring and background debugging (Chapter 6). The structure of Testbed programs provides natural breakpoints which enable non-intrusive monitoring and provide support for background debugging.

5. Dynamic modification (Chapter 7) and Migration (Chapter 8). The structure of the Testbed system and applications also allows some of the most difficult problems in these two fields to be solved. Much more flexibility in data structuring and a greater degree of automation of data conversion are provided than with previous work in the field of dynamic modification.

Some of the work contained in this thesis has been previously published by the author [12, 13, 14, 15, 57].

Chapter 2

The Programming Model

2.1 Introduction

Testbed is based around a distributed reactive processing model in which events are synonymous with messages. The processing entities are known as *slots* which are active objects consisting of encapsulated state data and code for processing and responding or reacting to messages. Slots may reside on the same processor or different ones separated by a network. In both cases communication is identical from the slot's viewpoint and any slot may send messages to any other without the need to establish a connection. Events may originate outside the system via a device interface or inside when one slot sends a message to another. Similarly a slot sends messages to a device as if it were another slot. When a slot receives a message it is processed by a unit of code called an *action*. This is a procedure which may update the slot's state in response to the message and any data it may contain and respond by sending new messages. Actions always terminate and do not perform input. Instead new messages are processed by new action instances. Only one action can be in progress at a time in any slot. The slot's state includes a table mapping *ports* to actions. Each message contains a port identifier and the system uses this to index the port table and determine which action to invoke to process the message. This port assignment may be altered by actions. Time is intrinsic to the model in that each message carries a timestamp which, if in the future, can be used to delay a message. Also a slot can have a deadline for processing each message which the system uses to schedule multiple slots on a single processor.

It will be argued that this simple programming approach is more appropriate to embedded applications than a conventional sequential process-based model such as CSP [43], though the latter is more general. Testbed provides the embedded systems software implementor not only with a debugging and testing environment,

but with a framework for the software. Specifying embedded software using the Testbed model involves breaking the system down into components which form the slots, determining the types of messages which these components need to send and respond to, and designing the ports and associated actions required to receive and process these messages. This resembles the state machine network approach often used in the higher levels of embedded systems design [86]. In fact state machines map naturally onto slots with events corresponding to messages, states to the combination of state data and port settings and state transitions to the operation of actions on the state triggered by the events. The event-action model also greatly simplifies and modularizes the implementation of the monitoring and debugging activities of Testbed. If the user were to provide a complete process then Testbed would be forced to probe at its inner workings from the outside. This would require a great deal of work at the compilation stage and would be especially difficult if the process had multiple threads.

The Testbed programming model is supported by a kernel described in Chapter 4 which transports messages between slots and performs scheduling operations. The rest of the Testbed's application and debugging support is implemented using slots and actions. Each processor has a *centre* slot which performs housekeeping activities and a special version of this slot residing on the root processor provides the user interface via the Unix host using the X Window System and Athena Widgets [71]. This is referred to as the *host server* slot.

2.1.1 Platform and Language Issues

Although Transputers were chosen as the platform for Testbed, the programming model is not Transputer specific and in fact differs greatly from the CSP/occam model [43, 103] normally used to program these systems[1]. Transputers are an appropriate platform for Testbed as they were designed for both embedded systems and multiprocessing applications. Testbed is aimed at multiprocessor embedded systems and the availability of Unix workstation hosted Transputer arrays such as the Meiko Computing Surface allow the software to be developed and tested on the same processor as the target system.

Given the event-action model, the choice of programming language had to be made. The obvious choice for Transputers is occam, however the fact that actions are sequential and are not permitted to perform blocking input would mean that the programmer would be restricted to a sequential subset of occam without the input operations. It was decided that it would be better to use a purely sequential

[1] See Section 2.11.

language for the actions and since C is also in common use for writing embedded systems software both on Transputers and other platforms, it was chosen.

2.2 Action Semantics

Actions are simple functions which are called by a special harness process. All actions are invoked with the same parameters. A C prototype for an action would have the form:

```
void action(void *data);
```

The parameter is a pointer to the data contained in a the message (if any). System functions are provided for obtaining information from or even a complete copy of the message header. This is useful for reasons such as identifying the sender, checking for expiration of the deadline or rescheduling the message for a later time.

Actions may send messages to other slots, to the same slot or to an external device, but may not perform input. This property reflects the view of action functions as interrupt handlers, which should execute for a short bounded period (without interruption). If blocking input needs to be performed from an external device, it will be performed by a special input handler process which is considered to be part of the system rather than application code. When a new message arrives it is placed by the system into a FIFO queue for the destination slot. When space for messages becomes exhausted, messages are discarded. It is up to the application designer to ensure that there is always enough space for the maximum number of messages which may be waiting at each slot.

2.3 Message Semantics

When sending a message, the sender provides the following parameters:

Destination The destination slot id.

Size The size of the message data.

Port An indication of the type of message to the destination slot.

Timestamp A time before which the message must not be processed at its destination.

Data Information to be processed.

The application does not need to know on which processor the destination slot is, though this location is specified by the user during initial loading and may change during development as slots are migrated to tune performance. The message is automatically routed to the correct node as described in Chapter 5.

The queue of messages for a slot will be referred to as its schedule. Once a message has reached the head of this queue it will eventually be removed and an action will be dispatched to process it. At this point it is no longer part of the schedule and cannot be overwritten by newly arriving messages.

When a message arrives with a timestamp which is greater than the processor's local clock value, its arrival is delayed until this time has past. Then the message is added to the back of the queue for that slot as if it had just arrived. Each slot also has a post time which is added to the timestamp to give a deadline before which the system should action the message if possible. In cases where no particular deadline is appropriate, a priority may be given. The deadlines and priorities are used by the system to choose between different slots on the same processor. This scheduling is described in Section 4.6. The destination slot has a port table which contains a mapping from the port contained in the message to an appropriate action. This mapping may be changed by the application.

It is possible that an overload situation may arise in which messages need to be discarded. To avoid losing crucial messages, the application may specify a minimum number of buffers which are reserved for use by messages destined for a particular port. On the other hand it may be known by the system designer that certain types of messages are liable to flood the system in exceptional circumstances and so it is possible to specify a maximum number of such messages which may be waiting to be processed at any one time, so protecting other more important messages from being lost. When a message arrives and the slot currently has the maximum number of messages for that port its data field is used to overwrite the data of the most recently arriving message. This mechanism is also useful as an optimization for update messages when only the most recent message is of interest. For example a device may periodically report the current value of some valuator. There is often no need to store several such messages so the maximum value can be set to 1.

The application can specify that messages for a particular port are *critical* in which case they may not be interrupted by the system as long as it performs no output. Also if a slot is waiting to process a critical message whose deadline is earlier than any other slot, the delay before the slot is scheduled will be short and bounded. The critical attribute is a useful way of reducing interference with time critical activities by other slots and by message traffic which would normally take

precedence. This attribute also provides a mechanism for responding to device or timer interrupts deterministically.

2.4 Devices

In keeping with the rule that actions cannot block awaiting input, synchronous devices, such as those attached to Transputer links or which indicate readiness via the event pin, are handled by kernel threads which convert incoming data into Testbed messages. There are also kernel threads which convert Testbed messages into raw data sent to the device. Since the input data does not arrive in the form of Testbed messages the destination slot and size of the data need to be specified by the application in advance. Also the application needs some way of addressing the device. Device links, as well as *raw* links[2], are specified as part of the network description[3] at which stage they are given slot identifiers. Any slot can send a message to a device as if it were an ordinary Testbed slot. Two function calls are provided to configure the kernel thread which inputs from the device, telling it the size of the incoming messages and which slot and port to deliver them to. The `configure_device` function causes messages of the given size to be read repeatedly and passed on to the given slot and port, while the `read_device` function results in a single message being delivered. In this way slots anywhere in the network can access any device transparently and input from the device can be routed to a receiving slot on any processor.

Asynchronous devices, such as memory mapped, polled devices do not need to be part of the kernel. Application actions can read and write them directly as this does not involve blocking. If access from multiple slots or from remote nodes is required to such a device then a dedicated slot can act as a device driver, but this is part of the application, not part of the system.

2.5 System Library

A set of library functions is provided to be called by applications. These include a subset of the standard C library functions as well as many Testbed specific calls which provide features such as communication, memory management and port management. A full explanation of these functions along with some useful macros which are provided in header files, can be found in Appendix A.

[2]useful during development, so that a device may be simulated by a Testbed slot on a separate processor

[3]See Appendix A.

2.6 Initializing Applications for Testing

When the application is ready to be installed on the target system, it will usually be linked with those parts of the Testbed system which are required to support its execution into a set of executable modules which can be loaded from disk or stored in ROM. However during development on the Computing Surface the Testbed system, including its user interface is loaded first using the Meiko system configuration and loading facilities (CSBuild [61]) and then the application is loaded by Testbed.

At system creation time, the physical network of processors, together with routing tables are supplied by the user. The target application is composed of a collection of slots which are assigned to physical processors (not necessarily one-to-one). There is no special compiler for Testbed. Instead the one which comes with the system (e.g., the Meiko C compiler) is used. The source code is divided into modules which are loaded as units. There are three types of modules, declaration modules, initialization modules and application modules. Each declaration or initialization module contains a declaration or initialization function which is called after the module is loaded. Declaration functions define user defined types, state variables and the initial port assignment for the slot. Variables can be given initial values when first defined. Initialization functions contain extra initialization which needs to be performed at system startup. Application modules do not contain a function which is invoked when they are loaded. Any of the three types of module can contain actions and other functions called by them, however the declaration and initialization modules may not call functions in application modules or declaration and initialization modules which are loaded after them. Nor may they refer to variables declared in these later modules. This is because declaration and initialization modules are linked immediately, and invoked, while the application modules are not linked until the end of the load process. The system is bootstrapped by means of a special initialization function executed on the host server. This function makes system calls to create and initialize all other slots in the system. The initialization of these slots is performed by loading the declaration, initialization and application modules which define them.

Actions are linked with the slot's state data. These data may be shared between actions, but there is no danger of conflict, as only one action may execute at a time. The state variables are maintained in a symbol table each entry of which contains the type of the variable and its address. Static and dynamic variables are created in the same way but are distinguished by associating a name with each static variable and allowing it to be linked into actions. Dynamic variables may only be referred to by indirection through pointers contained in other variables.

Type definitions and static variables are generally created by declaration functions which are invoked once when the application starts up or during a reload operation. Such reloads are the means by which the incremental modification occurs.

2.7 Programming Interface to Data and Types

Both types and state variables are currently defined using procedures and macros from declaration functions[4], in the case of static variables, or from action code in the case of dynamic variables. A set of standard types is predefined and these form the basis for the user defined types. With the exception of arrays and pointers, each state object must have a type corresponding to a definition in the type table. This means that all structures must be defined explicitly as types in the type table before being used (equivalent to having to typedef each struct used in a C program). Variables may be declared as pointers to, or arrays of, existing types (or pointers, arrays etc.). This results in all intermediate pointer and array types being added to the type table. The naming convention for these types is based on C syntax. A pointer to base_type is named base_type*, while an array of n base_types is named base_type[n]. For multiple dimensional arrays the dimensions are ordered from right to left. So base_type[n][m] is an array of n base_type[m]. Pointers and arrays may be combined. For example if a variable is declared of type int*[3]**[4][5], (with int already declared) then this results in the following types:

```
int*
int*[3]
int*[3]*
int*[3]**
int*[3]**[5]
int*[3]**[4][5]
```

In words the final type is: array of 4 array of 5 pointer to pointer to array of 3 pointer to int. It is not clear whether this is the best convention intuitively since to completely dereference a variable of this type called x, you might say: *(**x[3][4])[2]. Note that these declaration conventions do not exactly implement the C declaration conventions which are more general and involve brackets. They are used because such expressions are easier to parse than general C type specifiers.

Unions are not currently supported.

[4] detailed in Appendix A

2.8 Dynamic Memory Allocation

Dynamic variables may be created at any time by using the same functions as are used for defining static variables in declaration modules. They are distinguished by having a NULL name value. Both dynamic and static variables are stored in the symbol table and are converted during dynamic modification and copied during a migration.

The standard C library malloc and free operations take non-deterministic times, which make them unacceptable for a time critical embedded application. It is also possible that the heap may become fragmented so that requests for storage may fail after the system has been running for some time which is also unacceptable. For these reasons a free storage management scheme based on memory pools is used. The free store consists of a set of pools of free blocks of the same size, with one pool per block size. Allocating a new block consists of searching for the first pool (the pools are kept in a size ordered list) which has a free block large enough and deleting the first element (constant time). Since there is a small fixed number of pools, there is a small upper bound on the time for an allocation. Storage for messages also needs to be allocated dynamically and so a similar mechanism is used as for state variables. As the management of the schedule is performed by the kernel concurrently with the action, there needs to be a separate set of pools for this, known as the system store. Although the system heap allocation/deallocation is only performed by the kernel, the user can specify the heap configuration at slot creation time.

When a slot is created, arrays of pool specifiers are provided for the system and user stores. Each pool specifier consists of the block size and the number of blocks of that size. The system uses this information to initialize the two free stores. A default store configuration is provided so that the user can omit the specification initially. Section 3.3.5 describes the implementation of the heap.

2.9 Development Support Environment

In order that the behaviour and performance of the application in vitro be as close as possible to its in vivo state, as much as possible of the development system which will not be present in the live system should be off-loaded onto extra processors available on the development platform. These may be used for simulating external hardware devices and the environment, for off-loading some of the work of monitoring and for providing routing paths back to the root processor.

2.9.1 Simulating Hardware and Environment

The concept of simulating external devices using separate processors is suggested in [33] as a way of avoiding perturbing the software under test. This works particularly well with transputers as the external devices are likely to be attached through the same link interface which is used to communicate with other processors. Fortunately these simulation problems have much in common with embedded systems and Testbed is well suited to implementing them. Simulating devices and environment on separate processors from the application also provides an opportunity to perform monitoring on these processors, which does not interfere with the performance of the application system, although care is still needed to ensure that it does not interfere with the performance of the simulation.

2.9.2 Using Separate Processors to Reduce Monitoring Overhead

The presence or absence of monitoring software can cause differences in behaviour of the system being monitored, known as the probe effect. Off-loading some of the work involved in monitoring onto separate processors can alleviate the problem. This is the approach taken in [1], where an efficient breakpoint mechanism is used to capture information and pass it to a separate processor where analysis, filtering and logging or reporting back to the user are carried out, while the application is allowed to proceed with minimal interference.

2.9.3 Using Extra Routing Nodes to Avoid Interference

Due to the limited number of communication links per transputer, it may be necessary for messages to be routed through intermediate processors to get to their final destination. If these processors are involved in the application then interference may occur, which may be difficult to predict and may make deadlines impossible to guarantee. It is especially undesirable for the messages between processors which will not be present in the target system, such as those used for simulation and monitoring and the root processor, to interfere. This is another example of the probe effect. The problem can be avoided if there are extra processors available which can be used entirely for routing purposes.

2.10 Example: Implementing the ROV

The Testbed is particularly suited to the development of reactive embedded systems. The programming model described in this chapter is illustrated with the control software for the submersible Remotely Operated Vehicle (ROV) described in Section 1.1. More than one possibility exists for mapping the control system onto slots. One possibility is to have three slots on two processors, one for the high level control functions which are performed by the surface control module, which has its own processor and two for the low level onboard control functions and status reporting on the other processor. These will be referred to as HIGH_CONTROL, LOW_CONTROL and FEEDBACK. Initially it will be assumed that there is a HOST slot which converts operator commands into messages to the HIGH_CONTROL slot and to which the depth and compass values are reported and an ROV device which accepts motor on/off and direction values from the LOW_CONTROL slot, without acknowledging and requests for the current depth or compass value from the FEEDBACK slot which result in a reply. Later the method of implementing a user interface and a simulated ROV for testing will be explained. The camera control involves exactly the same sorts of messages and actions as the driving of the vehicle, except that there is no feedback within the system (the video signal is carried back to the monitor by a separate cable). It has been omitted from the implementation described here for simplicity.

2.10.1 Notation

Testbed programs will be described in an Algol-like pseudocode form. For each slot the actions, ports, outgoing messages and pseudocode are presented followed by initialization code. For the ports and messages only the fields which take non-default values are specified. Full C versions of the examples can be found in Appendix C. When a value such as the action corresponding to a port is constant it is indicated using an =, whereas if it may change := is used.

2.10.2 HIGH_CONTROL

Actions and Ports

This slot receives separate heading, horizontal and vertical velocity commands from the HOST slot (coming from the joystick and up/down buttons respectively). These are handled by the `Heading`, `H_Velocity` and `V_Velocity` actions. Messages which change the autodepth and autoheading status are handled by the `AutoDepth` and `AutoHeading` actions. Only the most recent of each of these

message types is of interest, so the corresponding port entry has the max field set to 1 in the following port definitions:

```
h_velocity   [action = H_Velocity; max = 1;
                 data = horizontal_velocity]
v_velocity   [action = V_Velocity; max = 1;
                 data = rate]
heading      [action = Heading; max = 1;
                 data = differential]
autodepth    [action = AutoDepth; max = 1;
                 data = status]
autoheading  [action = AutoHeading; max = 1;
                 data = status]
```

The slot also receives messages carrying depth and compass values from the on-board processor. These are normally handled by the `Hold_Depth` and `Hold_Heading` actions except when entering or resuming autodepth or autoheading mode when the target depth or heading is reset using the `Zero_Depth` or `Zero_Heading` actions. As with the messages from the HOST slot only the most recent is relevant, so max is set to 1:

```
depth_report    [action := Hold_Depth; max = 1;
                    data = depth]
heading_report  [action := Hold_Heading; max = 1;
                    data = compass]
```

Note that the HIGH_CONTROL slot is entirely reactive. It only performs actions in response to messages from other slots with no periodic activity scheduled from within the slot. However since the LOW_CONTROL slot described below periodically sends the depth and compass values, this imposes a periodic behaviour on the HIGH_CONTROL slot.

Outgoing Messages

Messages are sent to the LOW_CONTROL slot of the form:

```
UPDATE_MOTOR [dest = LOW_CONTROL; port = update_motors;
                  data = new_motor_settings]
```

and status messages containing the latest depth and compass readings are sent to the HOST slot:

```
STATUS_REPORT [dest = HOST; port = display_status;
                  data = status]
```

Actions

The actions in this slot make use of the utility procedure `adjust_motors` which computes new target motor speeds, changes the value of each motor in the direction of the target speed within the safety constraints[5] and sends `UPDATE_MOTOR` messages for motors which have changed. If condition 3 would be violated by the new motor settings, then the speed of each motor is reduced to 75% repeatedly until the sum of each of the two motor speeds on each side is within the required limit. The autodepth and autoheading control are performed by the `auto_vertical` and `auto_turn` functions. These functions may be found in Appendix C.

Heading messages contain a differential value in the range $[-100, 100]$, derived from the position of the drive joystick twist grip, which is the difference between the right and left motor values. A positive differential will thus produce a left (anticlockwise) turn which is consistent with compass values. The `rear_differential` value is used by `adjust_motors`, while `manual_diff` is set to indicate that manual heading control is in operation. If the grip position has returned to the neutral zero position and autoheading control is engaged, the action for the `heading_report` port is set to `Zero_Heading` so that the next reported compass value will be taken as the new target heading for autoheading control.

```
Heading(differential):

manual_diff := differential;
rear_differential := manual_diff;
adjust_motors;
if manual_diff = 0 and autoheading_mode then
    heading_report.action := Zero_Heading.
```

The horizontal velocity has both sideways (positive to the right) and forward components each in the range $[-100, 100]$. These values may be changed independently of autodepth or autoheading control.

```
H_Velocity(horizontal_velocity):

velocity.right := horizontal_velocity.right;
velocity.forward := horizontal_velocity.forward;
adjust_motors.
```

The vertical velocity can be -100, 0, or 100 corresponding to up down or stopped respectively. The `manual_v` variable is set to indicate that manual depth control

[5] See Section 1.1.

is in operation. If the rate received is zero this indicates that the button has been released and if autodepth control is engaged the action for the depth_report port is set to Zero_Depth so that the next reported depth value will be taken as the new target for autodepth control.

```
V_Velocity(rate):

manual_v := rate;
velocity.down := manual_v;
adjust_motors;
if manual_v = 0 and autodepth_mode then
    depth_report.action := Zero_Depth.
```

The autodepth and autoheading messages contain on or off status values. When one of these modes is selected, the appropriate port is set to the Zero action to set the target to be maintained.

```
AutoDepth(status):

autotdepth_mode = status;
if autodepth_mode then
    depth_report.action := Zero_Depth.

AutoHeading(status):

autoheading_mode := status;
if autoheading_mode then
    heading_report.action := Zero_Heading.
```

The Zero actions set the new target to the reported data and revert to the Hold action. If auto mode is set and manual override is not in effect the Hold actions compute new target velocities. Otherwise they just use the targets set by the operator. Only the Hold_Heading action adjusts the motors and reports the status to the HOST slot as there is always a heading report following each depth report from LOW_CONTROL in each duty cycle.

```
Zero_Depth(depth):

target_depth := depth;
depth_report.action := Hold_Depth;
status.depth := depth.

Hold_Depth(depth):
```

```
if manual_v = 0 and autodepth_mode then
    velocity.down := auto_vertical(target_depth - depth);
status.depth := depth.

Zero_Heading(compass):

target_heading := compass;
heading_report.action := Hold_Heading;
status.compass := compass;
Send(STATUS_REPORT).

Hold_Heading(compass):

if manual_diff = 0 and autoheading_mode then
    rear_differential := auto_turn(target_heading - compass);
adjust_motors;
status.compass := compass;
Send(STATUS_REPORT).
```

No initial messages need to be sent from this slot so initialization consists of setting the state variables to their default values.

```
Initialisation:

manual_v := 0;
velocity := {0,0,0};
manual_diff := 0;
rear_differential := 0;
autodepth_mode := FALSE;
autoheading_mode := FALSE;
zero_target_speeds.
```

2.10.3 LOW_CONTROL

This slot has a duty cycle during which it switches each motor on for a percentage of the cycle given by the speed.

Actions and Ports

The slot receives messages from the HIGH_CONTROL slot containing motor updates, handled by the `Update_Motors` action. Motors whose speeds are

non-zero are switched on at the start of the duty cycle by the Switch_On action which also schedules SWITCH_OFF messages. The port definitions are:

```
update_motors [action = Update_Motors; max = 1;
                  data = new_motor_settings]
switch_on     [action = Switch_On]
switch_off    [action = Switch_Off; data = motors_off;
                  critical]
```

Note that the switch_off port is critical as if it is delayed, the motor will stay on for too long.

Outgoing Messages

```
MOTORS_ON     [dest = ROV; data = motors_on]
MOTORS_OFF    [dest = ROV; data = motors_off]
MOTORS_PLUS   [dest = ROV; data = motors_plus]
MOTORS_MINUS  [dest = ROV; data = motors_minus]
SWITCH_ON     [dest = LOW_CONTROL; port = switch_on]
SWITCH_OFF    [dest = LOW_CONTROL; port = switch_off;
                  data = motors_off]
```

Actions

The motor settings from the HIGH_CONTROL slot are stored to be used by the Switch_On action.

```
Update_Motor(new_motor_settings):

motor_settings := new_motor_settings.
```

All motors whose speed is non-zero are switched on by the Switch_On action. This also sets the direction and schedules SWITCH_OFF messages. If several motors have the same speed then only one message is scheduled to switch them all off. If a motor speed is 100% then it is left on until the next duty cycle. The last Switch_Off action in each duty cycle reschedules the next SWITCH_ON message unless there are none, in which case it is scheduled by Switch_On. The Switch_On action is also responsible for advancing the next_duty_cycle variable. The while loop ensures that if there is an unexpected delay of some kind which causes one or more cycles to be missed, the missing cycles are simply skipped.

```
Switch_On:
```

```
while next_duty_cycle < Now do
    next_duty_cycle := next_duty_cycle + duty_cycle;
motors_on.set := motors_plus.set
    := motors_minus.set := EMPTY_SET;
for i := 0 to nmotors-1 do
  begin
    speed = motor_settings[i].speed;
    if different_speed(i) then
      begin
        motors_off.set := EMPTY_SET;
        add_motors_with(speed,motors_off);
        if speed < 100 then
          begin
            delay SWITCH_OFF by duty_cycle * speed/100;
            on_count := on_count+1
          end
      end;
    if speed > 0 then
        add(i,motors_on);
    if motor_settings[i].direction = 1 then
        add(i,motors_plus)
    else
        add(i,motors_minus)
  end;
Send(MOTORS_PLUS);
Send(MOTORS_MINUS);
Send(MOTORS_ON);
if on_count = 0 then
    Schedule SWITCH_ON for next_duty_cycle.

Switch_Off(motors_off):

Send(MOTORS_OFF);
on_count := on_count-1;
if on_count = 0 then
    Schedule(SWITCH_ON,next_duty_cycle).
```

The initialization of this slot involves scheduling the first SWITCH_ON message.

```
Initialisaton:

duty_cycle := DEFAULT_DUTY_CYCLE;
```

```
next_duty_cycle := Now + duty_cycle;
initialise_motor_settings;
on_count := 0;
set_priority(0);
Schedule(SWITCH_ON,next_duty_cycle).
```

Note that the priority for this slot is set to the maximum (0) which means that it will always be scheduled in preference to the FEEDBACK slot.

2.10.4 FEEDBACK

This slot polls the ROV device for the current depth and compass values which are reported to the HIGH_CONTROL slot.

Actions and Ports

Periodic polling is initiated by the `Poll_Depth` action (which requires no data). The depth and compass messages returned are handled by the `Report_Depth` and `Report_Compass` actions.

```
poll_depth    [action = Poll_Depth]
status        [action := Report_Depth; data = status_value]
```

Outgoing Messages

```
DEPTH_REQUEST    [dest = ROV; data = depth_request]
COMPASS_REQUEST  [dest = ROV; data = compass_request]
POLL_DEPTH       [dest = FEEDBACK; port = poll_depth]
DEPTH_REPORT     [dest = HIGH_CONTROL; port = depth_report;
                      data = depth]
COMPASS_REPORT   [dest = HIGH_CONTROL; port = compass_report;
                      data = compass]
```

Actions

The `Poll_Depth` action is scheduled initially for the start of the duty cycle for this slot. As in the LOW_CONTROL slot the while loop ensures that if there is an unexpected delay of some kind which causes one or more cycles to be missed, the missing cycles are simply skipped. After the `DEPTH_REQUEST` message is sent to the ROV, the action for the status port is set up to process the depth value returned. The `Report_Depth` action which processes this message reports the depth back to the HIGH_CONTROL slot and sends the `COMPASS_REQUEST` message to the ROV, setting the action to `Report_Compass` to process the

reply. This action reports the compass value to HIGH_CONTROL and schedules the next poll for the start of the next duty cycle.

```
Poll_Depth:

Send(DEPTH_REQUEST);
status.action := Report_Depth.

Report_Depth(status_value):

depth := status_value;
Send(DEPTH_REPORT);
Send(COMPASS_REQUEST);
status.action := Report_Compass.

Report_Compass(status_value):

compass := status_value;
Send(COMPASS_REPORT);
Schedule(POLL_DEPTH,next_duty_cycle).
```

The initialization of this slot involves scheduling the first POLL_DEPTH message and configuring the ROV device to pass messages of the correct size to the FEEDBACK slot, status port.

```
Initialisaton:

duty_cycle := DEFAULT_DUTY_CYCLE;
next_duty_cycle := Now + duty_cycle;
configure_device(ROV,status_value_size,FEEDBACK,status);
Send(POLL_DEPTH).
```

2.10.5 Notes

The FEEDBACK slot is given the default priority which is lower than that given to the LOW_CONTROL slot. This is because it is assumed that there will be sufficient slack time during the duty cycle of the LOW_CONTROL slot for the polling and reporting actions of the FEEDBACK slot and it is less important that these be delayed than that the SWITCH_OFF event is processed as near as possible to the scheduled time, avoiding the motors staying on too long. If the duty cycle of the LOW_CONTROL slot were so tight that too many of the status reports were missed, then both the operator's feedback and the autodepth and autoheading

control would suffer, so it might become necessary to sacrifice small variations in the motor speed. This effect can be achieved through the use of deadlines rather than fixed priorities for both the FEEDBACK and LOW_CONTROL slots.

2.10.6 Testing the ROV

So far only an implementation of the application part of the ROV control software has been described. In order to test the software in vitro, both the ROV control console (the HOST slot) and the robot itself need to be simulated. The serial cable which connects the surface controller to the on-board controller may also be simulated. The console, the robot and the serial link are each simulated on a separate processor by a Testbed slot. Figure 2.1 shows the configuration of slots and processors in the development architecture.

Note that the introduction of the CABLE slot requires modification to the implementation presented above to convert between the raw links and the slots at each end of the cable. Details of the customization of the Testbed user interface are presented in Chapter 3.

Device and Environment Simulation

The ROV device and environment simulation are provided by the ROV slot which resides on a separate processor attached to the processor containing the FEEDBACK and LOW_CONTROL slots by a raw link. To the application it appears exactly as if this link is connected to the real ROV hardware interface via a link adaptor. At a later stage in the development it would be possible to replace the raw link and the processor performing the simulation, with the actual hardware, without the application software needing to change in any way as a device link is indistinguishable from one end of a raw link. All that would be required would be a change in the network configuration file.

A raw link is accessible to the application via two device ids, one for each end, which are equivalent to slot ids, but must be specified in the configuration file. In this example the application end of the link has id ROV_DEV, and the simulated ROV end has id CONTROL_DEV.

The ROV simulation slot communicates with the application by sending and receiving messages from the CONTROL_DEV device but may also be accessed from the HOST slot or other surrogate slots involved in background debugging through the normal Testbed communication harness. Since messages sent through a raw link lose their header information, a single port is used for both types of status request and for motor updates. Each of these device messages contains a

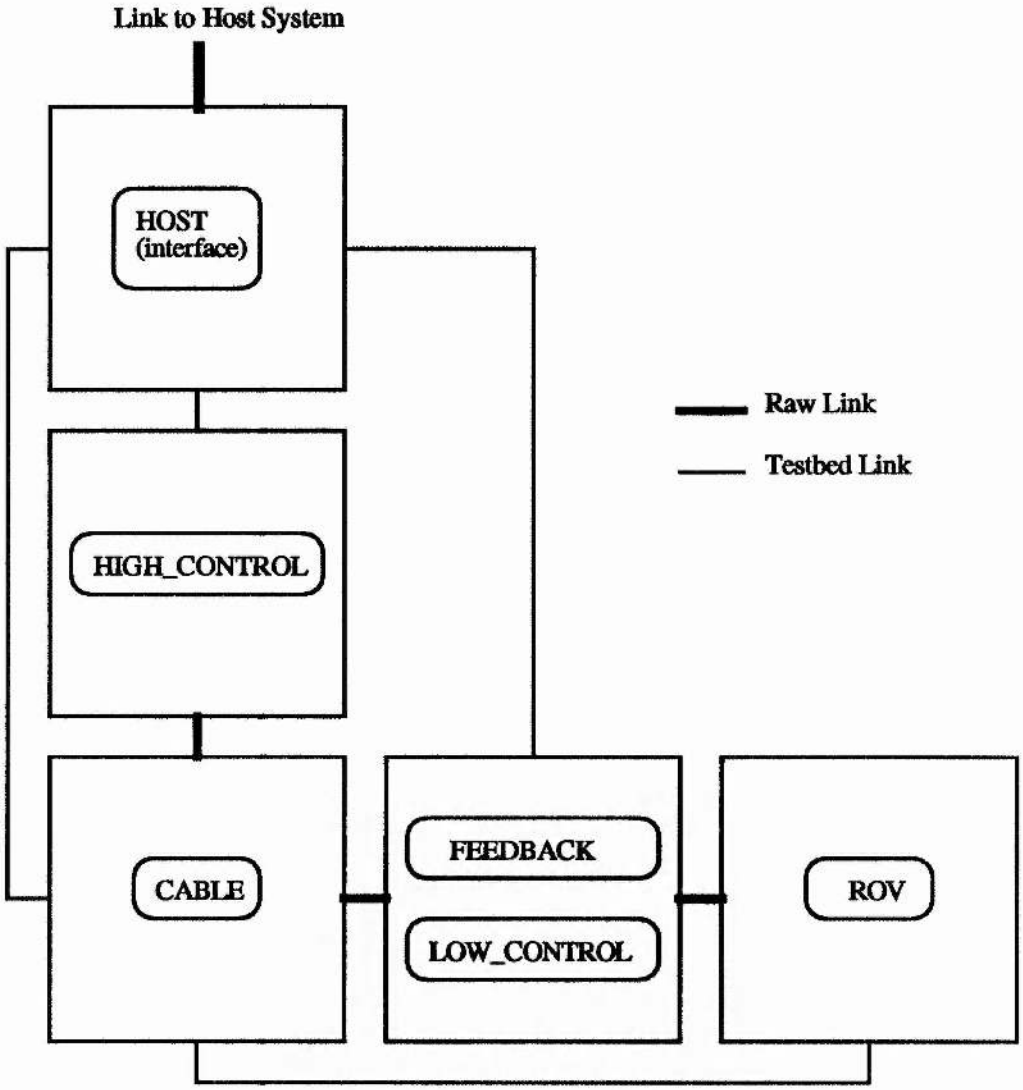


Figure 2.1: The ROV development architecture.

key which identifies its type. Each time a message arrives the current values of the depth and compass are updated by calculating the vertical velocity[6] based on the current states (on or off) and directions of the side motors then multiplying this by the time which has elapsed since the last update. An environmental component is added to the velocity to simulate the effects of currents, buoyancy of the vehicle etc. The compass is updated in a similar way except that the difference between the directions and states of the rear motors is used. Full C source code for the ROV slot is given in Appendix C.

[6]The damping factor of the water means that the motors drive the vehicle at a low constant speed, with acceleration effectively instantaneous.

2.11 Related Work

Most programming systems oriented towards development of real-time systems are based on conventional sequential processes, for which a scheduler [74, 79, 93] or communication model is provided [58]. Often in fact these systems are implemented in a Unix or similar environment, in the form of a light-weight threads package [36, 84].

Spring [92, 93] is implemented on a special architecture consisting of a number of SpringNet nodes connected by an ethernet for non-real-time traffic and a fibre optic ring connecting 2 Mbyte memory boards on each node providing replicated memory with predictable performance for real-time communication. Each node consists of a number of 68020 processors divided into system processors which perform scheduling operations and application processors which execute previously guaranteed application tasks. Although such a separation between time-critical and other work onto separate processors is not intrinsic in Testbed, it is supported by the global communication model. The Spring programmer sees a set of conventional processes containing critical sections and both synchronous and asynchronous communications which may be grouped to form a process group with a single timing constraint. These processes are transformed by the compiler into a set of execution units called tasks. The Spring kernel contains support for a wide variety of different task attributes. Testbed messages and their associated actions form the equivalent of tasks, but have much simpler attributes. This is not to say that a more complex mechanism may not be built on top of the simple Testbed scheduler in a layered fashion. The problem with a model such as the Spring one which hides the underlying scheduling units from the programmer/developer is that debugging and performance tuning become more difficult. The Testbed programmer is fully aware of the units of scheduling and yet does not lose the benefits of programming in a high-level language. In fact embedded systems are likely to be specified as a collection of state machines which map naturally to an implementation as Testbed slots, whereas in the Spring approach the programmer would need to map the state machine to a sequential process which is then converted by the compiler back down into a set of tasks. Like Testbed, Spring uses memory pools for preallocating a number of chunks of memory of a given size to avoid nondeterministic delays when a time-critical task requires memory. Also like Testbed, Spring's communication is based on message passing, however communication in Spring is more complex, with both synchronous and asynchronous messages supported, whereas Testbed only supports asynchronous messages.

In iRMX [82], memory set aside for message buffers is divided into 16 free lists of buffers each with a different size in order to save time when allocating a

buffer for an incoming message. Testbed uses a method similar to this[7] and in addition allows buffers to be reserved for particular types of message[8].

DRAGON SLAYER/MELODY [102] is similar to Testbed in that they use an earliest deadline first scheduling strategy[9] and assume that tasks arrive aperiodically at each node. In addition to a deadline, tasks may have an earliest release time corresponding to the timestamp in Testbed messages.

[46] presents a specification scheme which bears some resemblance to Testbed's programming approach and uses a formal logic called RTL to verify systems specified in this way.

A number of authors [33, 83, 88] note the need to use extra processors for testing and device and environment simulation. As with Testbed the same programming model, methodology and support system is often used for developing these test and simulation components.

The message driven programming model in the Testbed resembles the hardware message driven processing models found in the J-Machine [20] and in STAR:DUST [72] and the software approach of [101], with the distinction that it is not hidden from the application programmer.

Transputer-based embedded systems are often programmed in occam [103] which is derived from the CSP specification language [43]. Current Transputer implementations of occam only support direct channel communication between processes which are on the same processor or on adjacent processors (in which case the channel is mapped onto one of the hardware links). Due to this low level nature and the close mapping between occam constructs and Transputer instructions, it is often regarded as a Transputer assembly language. The event-action programming model chosen for Testbed differs markedly from the CSP/occam style. The key to this difference lies in the communication model. In CSP and occam processes communicate along synchronous channels with the destination process required to perform an input operation corresponding to each message sent to it, or a deadlock occurs. When asynchronous communication is required, to allow concurrent computation to be performed or because message order is nondeterministic, it must be built into the application using the CSP || and [] or occam PAR and ALT constructs. The resulting applications tend to be mazes of plumbing and debugging them often involves untangling the pipes and locating the sources of deadlock. Although of course Testbed-style programs could be written in occam, much of the Testbed system software (such as the routing and scheduling harness)

[7] Section 2.8

[8] Section 2.3

[9] See Section 4.6.

would have to be written by the application programmer and the resulting system would not provide the testing debugging and dynamic experimentation facilities. Providing these facilities for an occam programming system would be much more difficult due to the lower level, less structured nature of information flow in occam programs.

2.12 Conclusions

Testbed applications are collections of state-machines called slots which are active objects, receiving events in the form of messages and processing them in FIFO order. This programming model forms the basis of the Testbed run-time environment and debugging and testing support, implemented on a Transputer-based Meiko Computing Surface as described in the following chapters. This chapter concludes with a discussion of the reasons for choosing the programming model over a more conventional one and of trade-offs which were made.

2.12.1 Reasons for Choosing this Model

By contrast with many software development systems commonly used for embedded software the Testbed has a programming model which imposes greater structure on applications. The Testbed model was designed in recognition of the fact that most embedded systems are composed of a set of processes each of which has a main loop which first waits for an event and then dispatches an appropriate subroutine to deal with it. In Testbed the main loop is moved from the application into the system and the application designer/programmer is left with the task of writing the subroutines (actions). The rest of this section outlines the additional benefits of the model.

No Plumbing Required

Any component in an application can send a message to any other without requiring an explicit channel to be shared between them and if they are not on adjacent processors, without the need for extra routing code—Testbed has this built in. For any given topology any routing function can be verified deadlock free as shown in Chapter 5 which also gives methods of automatically producing suitable routings. This removes many potential deadlocks which occur in occam programs due to poor plumbing.

Deadlocks Easier to Analyse

The fact that an action may not receive messages (apart from the one which triggers it) combined with the fact that slots are always prepared to accept messages from the network allows the network layer to be freed from the possibility of deadlock, by choosing an appropriate routing, without the need for a complex protocol. Of course it is still possible for the application to become deadlocked. However when this happens, all parts of the system remain alive and a developer is able to examine any part of the application and determine the cause of the deadlock. Applications may still fail due to a backlog of messages using up all the message storage at a slot. However this will be reported by the system and again does not result in catastrophic failure. Often new messages will merely be updates which supersede earlier ones. In this case the max field of the port table is used and no additional storage is required. Processing time is also saved in this case as the superseded message is not actioned.

Natural Break Points

It was decided that the Testbed should not require special compiler support and that a standard compiler[10] should be used. This made the state monitoring, migration and dynamic modification design goals of the Testbed difficult to achieve if a conventional process model were used. Both these activities require a break point to be inserted in the code at which it is known to be safe to perform the operation. This is difficult to do without compiler support[11]. For dynamic modification it would be very difficult to match up the old and new versions especially if the process had multiple threads created with an occam style PAR.

The breakdown of each process into short-lived action functions provides natural break-points at which state monitoring, migration and code and data replacement can safely occur.

Separate State and Type Information

The separation of the definition of the global state variables and type definitions from the main application code into separate modules which make function calls allows the system to maintain the type and symbol tables which allow monitoring requests to be satisfied independently of the application without the need to parse the code. It also allows data to be converted automatically between the old and

[10] in the current implementation the Meiko C compiler [62]

[11] Sophisticated preprocessing would be required.

new versions of the code and for pointers to be updated during both dynamic modification and migration.

2.12.2 Trade-offs

As with all programming models Testbed's has trade-offs between such attributes as ease of programming, ease of understanding, predictability, implementability and performance. In addition Testbed introduces the ease of development and debugging as factors in this equation. For example, to enhance predictability, it was decided that the overhead introduced by the system (i.e., the kernel) for any single operation, such as scheduling a message, should have an upper bound which is proportional to the number of slots on a node and not on the number of messages waiting to be processed. This limits the degradation in performance during periods of high message traffic. The consequence of this decision is that the kernel is able to perform deadline scheduling on slots (requiring that the active slots be kept in an ordered queue), but cannot sort the messages waiting for individual slots. This lead to the FIFO queueing of messages within each slot. The use of memory pools rather than the more general malloc mechanism for dynamic storage is another trade-off of flexibility against predictability.

Chapter 3

The Testbed System

3.1 Introduction

Testbed is structured into three identifiable layers as shown in Figure 3.1. Code entities in the figure are shown in rounded rectangles while the main data entities are shown in ordinary rectangles. Data is largely isolated within each layer with access through function calls between layers, though sometimes lower layers may access data in higher layers via pointers. For example the kernel accesses the sys_q and user_q message queues and the port table for a slot when a message arrives. Each processor has a single copy of the kernel layer code and data to provide message passing and scheduling services. There is a separate copy of the system layer data for each slot, but a single copy of the code is shared across all slots. This layer provides all debugging and development support through system actions and the symbol and type tables. The application layer is different for each slot and consists of the application actions and related code and the state data which is linked with the code.

This chapter describes the structure and implementation of the system layer. The kernel layer is covered by Chapter 4.

3.2 The Slot

Each slot consists of a set of state data and an action_harness process. The state data is divided into system and application (user) parts. The system data are either inaccessible or are accessible only by function call from the application, while the application data are linked directly with the application code. Except for a small amount of information which is kept in a *frame* array which is accessible to both the system and kernel layers, the state data is also accessed by the kernel threads through function calls. In this sense the slot is a passive object.

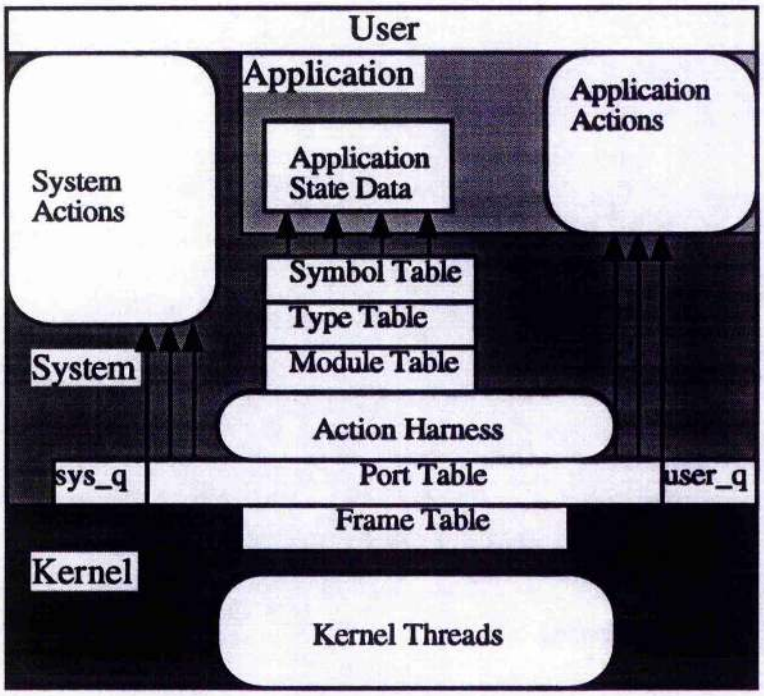


Figure 3.1: Software layers in Testbed.

The system data includes the port table, the user and system message queues (user_q and sys_q respectively in Figures 3.1 and 4.1), and a pointer to the slot's current message. The port table maps the port id contained in the message header to an appropriate action function. It also contains scheduling and debugging information as described in Section 3.3.1. There are two FIFO message queues to allow system messages to overtake application ones. When the kernel executive thread chooses a new message to be actioned for the slot the system queue is checked first, then the user one.

Each slot has an action harness, which is a thread, similar to kernel threads except that it has its own data area and only shares a few system variables with the kernel via pointers. These shared system variables can only be updated by the kernel or by critical system actions. The action harness calls an action function also specified in the port table to process each new message. After this function returns the harness performs any event reporting requested by monitoring clients and then signals action termination to the kernel executive thread. This releases the storage allocated for the message and selects the next message to be actioned from one of the slot's queues, if there are any. Figure 3.2 shows the operation of the action harness process[1] which has three states: wait_for_message in which the message queues are empty and there is no current message, wait_for_kernel in which there is a current message, but the slot is currently inactive as some other

[1]The state of this process shall be referred to as the state of the slot.

slot is running, and action in which it processes the action. The slot may alternate between the action and wait_for_kernel states several times as it is preempted or as it relinquishes the processor in order to send a message, then has to wait until it is scheduled again by the kernel.

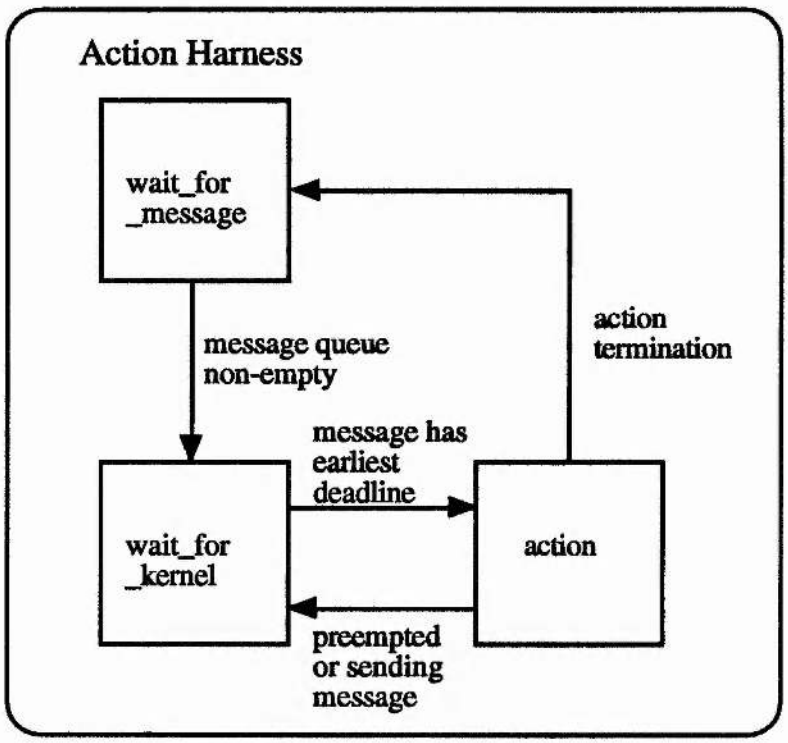


Figure 3.2: The action_harness process within the slot.

3.2.1 Accessing the Slot's System Data

Each slot has its own private system data. The system library functions need to be able to access these data when called from an action or from a system thread. One possibility would have been to require the action library and system functions to have an extra parameter which is an index into a table whose entries are structures containing all the globals for a slot. However a better method is to make use of the fact that the Meiko C compiler already passes a hidden parameter bssp, which is a pointer to the base of the calling process' global data area. All references to global data, apart from pointer dereferences use an indirection through this pointer. This is a common practice with Transputer C compilers described in [37]. Whenever a new slot is added an entry is created in the frame table. A field of the frame table entry is made to point to a copy of the system's global data area. Kernel threads which need to access the slot's global data do so through function calls which index into the frame table to get this pointer and then use it to overwrite the

hidden bssp parameter at the bottom of their call stack. This is the first thing the action harness does, so subsequent library calls by actions see the slot's private copy of the global data and do not need to be passed an extra parameter.

3.3 Slot Structure

The slot contains tables: module_table, user_q, sys_q, port, symbol_table and type_table.

3.3.1 Slot Tables

module_table A linked list of module descriptors containing the name of the module and a pointer to the data structure received when the action was loaded which is unchanged except that the code segment has been patched to link in the required external functions and global variables. The entries also contain a pointer to the previous version of the module which is saved during a reload operation in case the reload is aborted. There is also a list of local variables, used to store data for initializers, and local labels and a list of global labels. These are also inserted in the symbol table, but the list here is used to keep track of which module the symbols came from so that they can be removed if the module is removed. Each of these lists has a corresponding old version which is used if a reload operation has to be aborted.

sys_q,user_q Lists of schedule entries stored in arrival order. System messages are distinguished from user ones by the magnitude of the port id. Messages are taken from the sys_q first by the kernel.

port A table indexed by the port identifier contained in each message. Each port descriptor contains the following fields which can be specified by the application:

action The current action function.

min The number of reserved buffers for messages of this type.

max The maximum number of messages which will be stored for this port (not counting one which is being processed).

critical Flag which specifies that any message for this port is critical. The effect of setting this flag is to provide a high priority interrupt action. The implementation of this is described in Chapter 4.

type The type of the data contained in messages for this port. Used for debugging purposes.

In addition the port table contains the following fields which are used by the system:

count The number of messages currently stored for this port.

old_type Used during dynamic modification so that data in messages can be converted.

box Pointer to the most recent message waiting to be processed.

free Pointer to the list of reserved buffers.

monitor Event based monitoring information.

symbol_table A table of symbol entries. Used for linking, monitoring, state conversion and slot migration operations.

type_table A table of type definitions used for monitoring, state conversion and slot migration operations.

Most of these tables are only used by the system layer, however the user_q and sys_q are used by the kernel's executive thread which selects the next message to be actioned by the slot and the count, max, critical, box and free fields of the port descriptor are used by the kernel for scheduling messages. The slot's system state also includes the **post** variable which is set by the application and used by the kernel to determine the deadline or priority for each schedule entry.

3.3.2 Access to System Data from Applications

Application code may not access the system data directly, for safety reasons, however library functions provide means of access to values such as the information contained in the header of the message currently being processed. They also allow modification of some of the system tables. For example certain fields of port table entries can be changed, new variables can be created and added to the symbol table and new types can be added to the type table (normally during startup or when reloading the system).

3.3.3 Library Functions and Application State

System library functions are automatically inserted into the slot's symbol table when it is created, so that they may be linked with the application code. These

functions include `def_var` which allocates storage for a new variable[2] and adds an entry to the symbol table. This function is used in declaration modules when the system is loaded to create named variables which are then linked with the code in application modules. The same function is used for creating dynamic variables, except that no name is given for these and they can only be accessed via named variables.

The host slot has some extra library functions not present in other slots which provide access to the host file system, X Windows functions and other useful functions which require host facilities.

3.3.4 System Ports and Actions

The port table on every slot is divided into system and user ports. System ports receive preferential treatment by the scheduler in that they have their own message queue which is ahead of the user queue. Each slot has a set of system actions assigned to these ports which perform functions such as module loading, peeking, poking and migration. The special slots, described in Section 3.5 have a different (overlapping) set of ports defined to ordinary application slots. Appendix B lists the system port definitions.

3.3.5 Heap Implementation Details

Each heap object has a header consisting of an offset from the start of the store to the next free entry (when the object is not in use) or a special code, the bottom bit of which indicates which store the block is from (system or user). The high order part is the index of the memory pool from which the block was allocated. The free operation consists of entering the block at the head of the free list for the appropriate store and pool (constant time). If the object is allocated the body consists of a symbol table entry. When the store is allocated, the system adds the size of the heap header plus the size of a symbol table header to the size given in the store specification to give the actual size of the heap object.

3.4 Loading Modules

Modules are loaded using a special system action. Linking proceeds in two phases. In the first (load) phase the labels and local variables are extracted from the object module, but no linking is done. In the second (link) phase the modules are linked

[2] or redefines an existing one during dynamic modification

using the symbols obtained from the symbol table as well as the local variables. Any of the global functions in a module may be called from other modules, but if name conflicts occur the one chosen is not defined. In the case of initialization and declaration modules, the linking occurs immediately after the load phase and the initialization or declaration function (which has the same name as the module) is called to define variables and initialize ports. Declaration or initialization functions which refer to global functions in other modules or variables created by other declaration modules have to be loaded after those modules. However the linking of application modules is delayed until the end of the load operation so that application modules may access any variable or global function.

The host slot contains a library routine which recompiles the action source if necessary and loads the object file. This function is called either from an interactive action, which allows the user to load individual modules, or from the main initialization and configuration function which is called after startup. Once loaded the action may be sent to the appropriate slot or loaded on the host slot itself.

3.4.1 Construction of Object Module Descriptions

At the host end, loading a module consists of compiling the source file to object form and, if there are no errors, reading in this object file and creating an object module descriptor. This consists of the name of the module followed by the object code. At the destination slot the object module is linked with external names resolved by looking in the symbol table. Definitions of local permanent variables cause storage to be allocated for these and stored in a list associated with the module. These are used for initializing structures and arrays and for statics (though the latter should not generally appear in application modules, as they can not be converted during replacement).

3.5 Special Slots

3.5.1 The Centre

The centre is the first slot to be created when the system is loaded. It is responsible for synchronizing the clock, maintaining the routing tables for the node, loading new slots, handling node suspension during migration and managing synchronous devices. Unlike other slots the centre's global data area is the same as the kernel's so that it can access node-wide data such as the routing tables. The centre recognizes a number of special ports as described in Appendix B. Most of these

ports have the critical attribute set so that the centre actions do not conflict with kernel threads.

3.5.2 The Host Server Slot

The host interface is implemented as a slot on the root node, which is also the centre slot on that node. An extended version of the system software is loaded which contains extra functions performed by the host server. The X window manager is an action which flushes the X output queue and checks for X events (from the X server) in the X input queue, processing these. This is repeated until no events remain, then the action is rescheduled after a short delay. This action is also called as a function by other actions which update the display.

Input and output operations (including X windows protocol messages) are performed using Meiko I/O functions which communicate via the Computing Surface to host interface with a Unix daemon process.

3.6 Host Services

The host slot contains two types of functions:

1. actions which respond to messages from the system, normally to display debugging information, and
2. X toolkit *Callback* functions [71] which respond to user requests via the graphical user interface, such as menu selections and dialog button presses. These normally result in one or more messages being sent to other slots in the system.

3.6.1 User Interface

The operation of Testbed's windows does not quite match the conventional philosophy of an X windows application (although this philosophy has many similarities to Testbed itself) in which output is performed in response to user input via the display server. Since the X library event handling routines do not give access to the low level interprocess communication protocol, it is not possible for the process which responds to display events to also detect input from other processes in the Testbed. Instead output is performed by calling the appropriate library functions (which would normally be called from X widget *Callback* or *Action* functions) from host server slot actions. There is no danger of conflict with the X event

handler, as this is implemented as an action which can not be running at the same time as other actions.

Athena widgets are currently used to create all the windows and subwindows. Each application slot as well as the host slot has its own composite window. Each slot window is divided into three panes: the system command box containing the main command menus, the user control box which may contain subwidgets created by the application, and the scrolling output window for the slot. Each slot has a "Slot Commands" menu and a "Variables" menu, while the host slot also has a "General Commands" menu.

3.7 Booting the Development System

The user supplies Testbed with the name of a file containing the routing tables and device information[3] and the name of a code module containing an initialization function with the same name. This module is loaded onto the host slot after the kernel is booted[4] and the initialization function is called. This function makes system calls to define all other slots and to load all code and initialization modules onto them[5]. It may also make application specific extensions to the user interface and add actions to display values reported by the application within these widgets or in the scrolling text window. Widgets which send messages to or display information from a particular slot are conventionally attached to the window for that slot, however this is not compulsory.

3.8 Customizing the User Interface

Due to the varied nature of embedded systems, it is desirable to be able to add application specific elements to the user interface of a development environment. This capability compliments the extra slots and actions which can be added for development support.

The X window system is initialized and the host slot window is created before the application initialization function is called. This function makes system calls to create the slots which make up the application and any extra slots used for device simulation, monitoring and debugging. A new window is created for each of these slots and initialized with the three standard panes. The application may then add extra widgets to any of these windows or even create new top level

[3] See Appendix A.

[4] See Chapter 4.

[5] Details of these system calls may be found in Appendix A.

widgets. However the additions are normally created in the box subwidget set aside for application specific additions within an appropriate slot.

3.8.1 Example: The ROV Control Panel

Figure 3.3 shows a snapshot of Testbed testing the ROV implementation from Chapter 2. The buttons on the real ROV control panel[6] have been mapped directly onto button widgets. The AUTO/MANUAL DEPTH and AUTO/MANUAL HEADING buttons are toggles while the UP and DOWN buttons each send a vertical velocity of ± 100 on button down and 0 on button up, thus simulating the buttons on the real control panel. The joystick for driving the ROV is simulated by two conventional scroll bars while the twist grip for turning is simulated by a scroll bar whose thumb may be dragged left or right to turn in that direction, but which snaps back to the neutral position on button up. The configuration function contains an action which responds to status messages sent by the HIGH_CONTROL slot. The 20 character LCD display is simulated by a label widget which is updated by this action.

In the scenario shown by Figure 3.3, the customizations to the HIGH_CONTROL slot's window are complimented by the standard variable monitoring dialog which is displaying the value of the `motor_settings` variable from the LOW_CONTROL slot whenever a message with the port `update_motors` has been processed. Chapter 6 describes the Testbed monitoring facilities which enable this.

3.9 Conclusions

The Testbed system layer forms a run-time harness around applications which conform to the model presented in Chapter 2. It provides a system library containing functions useful to embedded applications, as well as debugging utilities and a customizable user interface which enhances the development and testing process.

This chapter has described how the system layer is built from special slots and actions and with the help of various system data structures. These include the host server slot which provides the user interface to the Testbed development environment. Application specific code is run on the host slot to bootstrap the application and load extra development facilities. The application can extend the user interface by creating new widgets and adding X windows Callback and Action functions. Testbed actions can be added to process and display information

[6] Figure 1.2

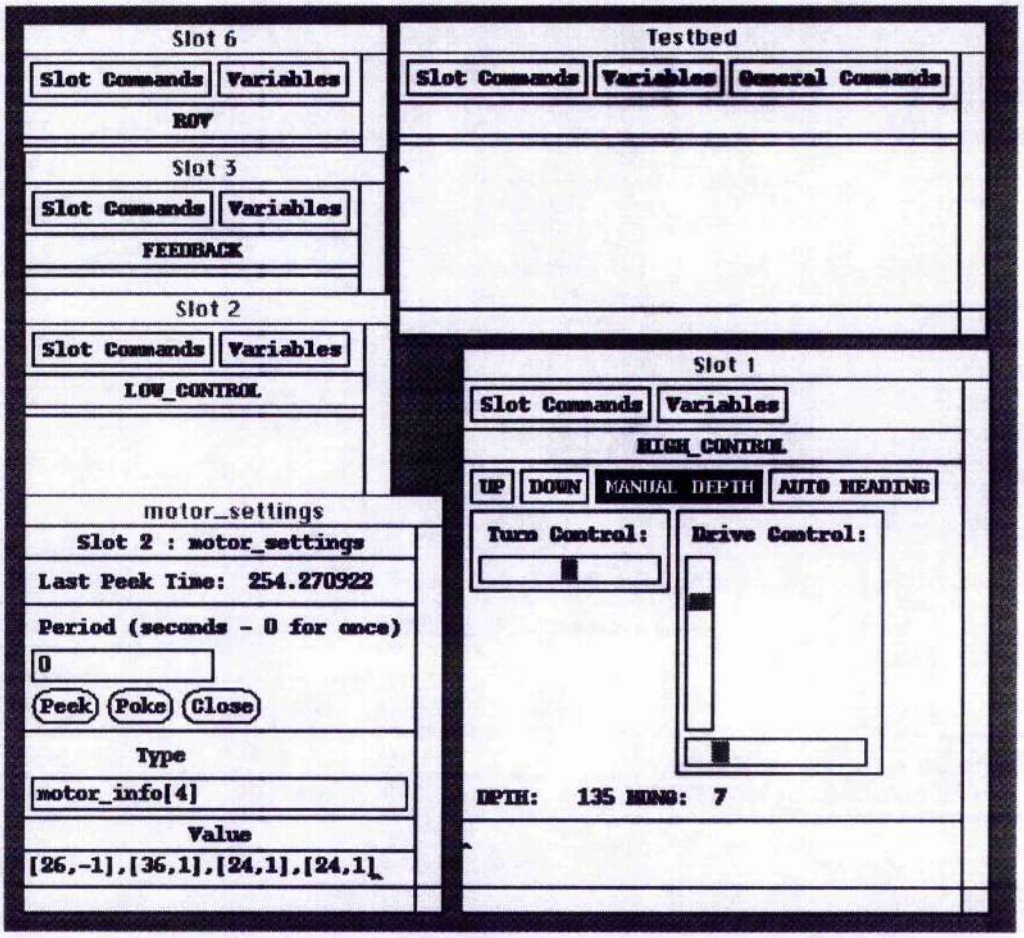


Figure 3.3: Snapshot of the Testbed implementation of the ROV.

reported by the extra application development code.

Chapter 4

The Kernel

4.1 Introduction

One of the difficulties about writing a general-purpose embedded systems operating system is that there are very few features which are common to all embedded systems. For this reason they are often written from scratch on the bare hardware. For very simple systems this approach works well, however as soon as the system complexity rises above that of a simple controller with one or two inputs and outputs, to one where there is a significant degree of concurrency and possibly the requirement or desirability of using more than one processor, it becomes very difficult, time consuming, error-prone and tends to result in a system which is very inflexible. One alternative is to use a traditional operating system such as Unix™, designed for use in a multi-user general purpose computer or workstation. The problem with such systems is that they provide too much. Embedded systems generally do not require virtual memory and often do not need other integral components of such operating systems such as a file system. The scheduler in such a system is often geared towards providing good average response to interactive users, but may have potentially unbounded delays. As mentioned in Chapter 2, the programming model supported by such systems is generally not the most natural one for embedded systems, and consequently the programmer will still have to implement a state machine on top of the operating system's conventional process model. By contrast microkernel operating systems such as Mach 3.0, Amoeba [97] or Chorus [81], provide only the bare minimum of features which are common to nearly all concurrent systems and allow the developer to select only those higher level features which are needed by the application. This makes them a good choice for embedded systems. Another major advantage of microkernel operating systems is that they are much easier to write and maintain. Since the kernel is very simple it is less likely to contain bugs and it is unlikely that bugs will arise

due to unforeseen interaction between higher layer features and the kernel. This is very important as it is extremely difficult to debug the kernel. Perhaps ironically, it tends to be highly susceptible to the probe effect.

The BED (Basically Event Driven) kernel is designed to support the Testbed programming model described in Chapter 2. Each processor has its own copy of this kernel which is composed of a number of threads or active objects as shown in Figure 4.1. Each kernel thread is a simple state machine which spends most of its time dormant and when requested performs one of a small number of alternative actions. In this sense the kernel tasks resemble slots themselves. Most have only one or two dormant (or blocked) states and in between these run without interruption. The kernel performs two basic functions: providing a message delivery service to slots and managing the processor (i.e., scheduling slots). These functions are related in that scheduling is based on individual message deadlines. However they represent the two basic functions of an operating system identified in [97]: providing an extended (or virtual) machine and acting as a resource manager. In addition the slot's action harness contains hooks for monitoring and reporting of events. All other monitoring, debugging and development facilities offered by Testbed (such as slot migration and dynamic replacement) are implemented through special actions and slots (such as the centre which is present on each node and the host server slot on the root node).

4.2 Related Work

4.2.1 Commercial Real-Time Kernels

Many commercial real-time kernels are available (such as HP-RT (Hewlitt-Packard systems), LynxOS (many platforms), OS-9 (Motorola 6809 and 68000), QNX (Intel 80x86) and Helios (transputer-like systems)). Most are designed to be Unix-like or compatible and to support a traditional programming approach. BED is not Unix-like and is designed to support a programming style which is specifically tailored to the reactive nature of embedded systems. Of the systems mentioned BED most resembles Helios in that the latter is a message-based system designed to support multiprocessors. However Helios is still designed to support conventional sequential application tasks which communicate using a Unix streams-like mechanism. In order to achieve the high performance and deterministic behaviour often required for an embedded application the Helios programmer must bypass most of the systems high-level facilities and gain access to the raw hardware.

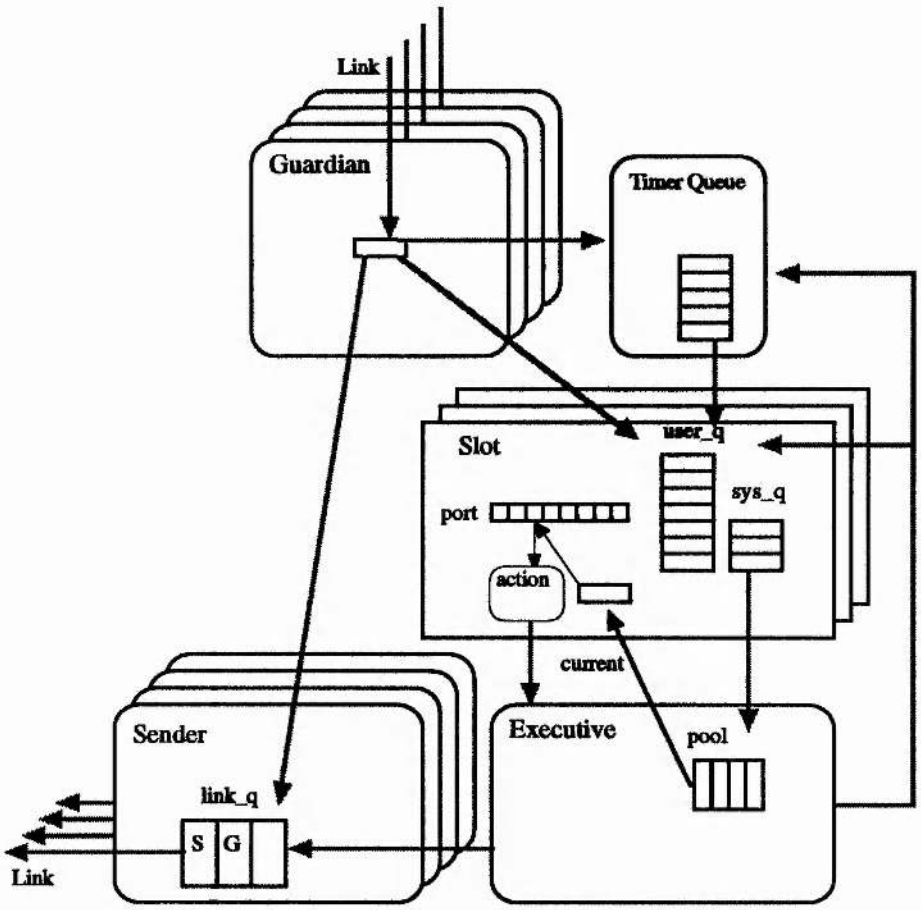


Figure 4.1: Entities making up the Testbed kernel.

4.2.2 Review of Real-Time Scheduling

Terminology

Processes In the real-time scheduling literature the term *process* is used to mean an activity which has a *release* time, which is its earliest allowed start time, a computation time (which may or may not be known in advance) and a deadline, which is normally taken to be relative to the release time. *Periodic* processes are processes which are repeated with release times at regular intervals. *Non-periodic* or *asynchronous* processes are repeated processes whose release times occur at irregular intervals. A distinction is sometimes made between *sporadic* processes defined as non-periodic processes, which have a minimum time between successive releases, and *aperiodic* ones which don't.

Processes may be entirely *preemptable*, meaning that the scheduler can interrupt the currently running process at any time and begin or resume another, or *non-preemptable*, in which the running process may not be interrupted. It is also possible for processes to be non-preemptable for only part of their lifetimes during

critical sections.

Scheduling Scheduling strategies can be divided into two classes; *static*[1] (a.k.a. *pre-run-time*) or *dynamic* (a.k.a. *online*). In static scheduling all process release times are known in advance and a table is generated which the scheduler uses to determine when to perform context switches. In dynamic scheduling the scheduler makes decisions on the fly, without prior knowledge of what processes will arrive.

A schedule is termed *feasible* if all processes are successfully scheduled after their release times and completed before their deadlines. Various optimality criteria are applied to scheduling *strategies* (methods of finding, or attempting to find feasible schedules). The most common one is that the strategy always finds a feasible schedule if one exists. Others include minimizing some measure of the lateness of processes.

Static vs Dynamic Scheduling

Static Scheduling [104] asserts that pre-run-time scheduling is essential in order to guarantee to meet hard deadlines in real systems with large numbers of processes and little free capacity. This is entirely due to the presence of non-preemptable processes. A dynamic scheduler which does not have knowledge of future process releases may schedule a non-preemptable process which blocks another process released after it, which has a deadline before the first process will become preemptable. In this situation the second process misses its deadline, even though it might have been possible to delay scheduling the first and allow both to meet their deadlines. According to [104], *"For satisfying timing constraints in hard-real-time systems, predictability of the system's behaviour is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system."*

Dynamic Scheduling Most common dynamic scheduling strategies are priority based, with the priorities assigned so as to increase the likelihood of meeting deadlines. The most popular dynamic scheduling approach is *earliest deadline first* in which priorities are assigned dynamically according to the order of the deadlines. Others include *rate monotonic*, which assumes periodic processes with deadline equal to period and assigns priorities statically in proportion to frequency, *least laxity*, in which priority is assigned in inverse proportion to the amount of slack time (difference between time to deadline and remaining computation time) and *shortest remaining processing time first.*

[1]Not to be confused with static priority *dynamic* scheduling.

Dynamic scheduling has the advantage of flexibility. Real-time systems generally have to interact with the outside world and this introduces unpredictability which is difficult to take account of in static scheduling.

Periodic vs Non-periodic Processes

The rate monotonic algorithm [56] assumes that all processes are periodic and preemptable. This is clearly not the case with many embedded systems which interact with the real world. It is possible to recover some use from the algorithm for sporadic processes by providing a periodic process with period equal to the minimum time between releases of the sporadic processes, which polls for these processes, as described in [3]. This method also works for static scheduling. The earliest deadline first algorithm is introduced in [56], to improve processor utilization for periodic processes, but [3] and [39] show that it is of much more general use, deriving schedulability tests which cope with sporadic processes without the need for special periodic processes. The algorithm is further shown to optimize lateness and tardiness in the case of aperiodic processes in [94].

Preemptable vs Non-Preemptable Processes

The assumption that all processes are preemptable is often made. This is not very realistic (as pointed out in [104]) since a real system is likely to contain critical sections during which no interruptions are allowed. The schedulability test derived in [39] for the earliest deadline first algorithm takes non-interruptible processes into account, but unfortunately is only an instantaneous test, which limits its usefulness. In general the presence of non-interruptible processes, upsets priority-based scheduling strategies, giving rise to the *priority inversion*, in which a high priority process is blocked by a lower priority one which happens to be in a critical section.

Hardness of the Deadline

In some systems (particularly safety critical systems) deadlines are considered hard, meaning that it is absolutely essential that the deadline be met. Often the execution of a task beyond its deadline, or before its scheduled release time, is considered of no value. Sometimes the consequences of missing a deadline may be more complex, as described in [4] and [47]. There may be some benefit still to be gained by executing the task after its deadline has passed or in beginning its execution early. On the other hand there may be a negative benefit, i.e., it may actually be harmful. The approach taken in [94] suggests that there is some benefit

in performing an operation late, although this lateness should be minimized, while in [4] schedulers which use time valued functions which are constant between the release time and the deadline and curve downwards before and after are described. Hybrid systems are also described which have multiple deadlines, with increasing degrees of hardness.

State of the Art

The most popular algorithm for dynamic scheduling seems to be earliest deadline first, with [56] comparing it favourably with the rate monotonic algorithm in that it improves processor utilization for periodic processes and [39], comparing it favourably with the least laxity algorithm on the grounds that the latter leads to more context switches. [94] proposes a combination of earliest deadline first and shortest remaining processing time first and shows that it minimizes lateness in a vector sense and the number of tasks that miss their deadline at the time the first missed deadline occurs. All of these algorithms require that there be a known minimum time between process release times in order to give any overall guarantee that no deadlines will be missed.

Mathematical scheduling problems and algorithms in the literature are criticised in [104] for making too many unrealistic assumptions about processes in real-time systems, such as that either all processes are preemptable, or that no processes are preemptable. On the other hand they argue in favour of static scheduling as the only way to guarantee to meet deadlines. However for this to work it is necessary to make assumptions such as that all processes are periodic or that there is a known minimum time between releases. They assert that this is acceptable on the grounds that most processes in real-time systems are periodic ones. Current standard practices are criticised for being inadequate to guarantee to meet hard deadlines. Examples of such practices given include:

1. assigning static priorities, which limits the set of possible schedules for a given set of processes, and

2. allowing events to interrupt processes and occupy system resources at any random time, increasing the unpredictability of the system.

Another argument in favour of static priorities put forward in [104] is that it makes it possible to avoid using sophisticated run-time synchronization mechanisms by directly defining precedence relationships and exclusion relations on pairs of process segments. The claim is made that it is easier to verify that all processes will meet their deadlines with a static schedule than when run-time synchronization mechanisms are used.

4.3 Message Flow Through The Node

All components of the kernel are involved in delivery and processing of messages. The grey arrows in Figure 4.1 represent either transfer of control of a particular message or creation of a new message. The rounded rectangles represent threads: the guardians which handle incoming messages, the timer queue thread waiting for timeouts or new messages, the executive which handles task switching and the senders which send messages out to the network. Each of these is essentially an interrupt handler with a number of states. In between these states they run without interruption. The major kernel data structures, as well as those which are shared with the system layer, are depicted according to the convention that the thread which uses or acts on the information contains the data structure. For example the executive thread takes message entries from the pool and processes them. The slots are shown as ordinary rectangles to indicate that they are passive objects each of which contains queues of messages and the port table. Each also contains an action_harness thread shown as a rounded rectangle although this is not a kernel thread as it may be interruptible. Although there can be several slots on a node, at most one may be active at a time. This will be referred to as the *current* slot and the message which this slot is processing as the *current* message.

Messages may enter the node from the outside, via the guardian or from the inside via the executive as a result of an action sending a message. In both cases control of the arriving message then flows to the timer queue, one of the slot queues or a sender thread. From the timer queue, the messages are inserted into a slot queue and from there they pass eventually into the executive's pool of active messages. Once under the control of a sender they will eventually leave the node.

Implementation Notes

Messages are divided into two parts; a header and a body, except in the special case where the message data will fit within the size of a scalar, in which case it resides in a special data field of the header. In this case only the header need be transferred. The message header also contains the source and destination node, the source and destination slot, the destination port and a timestamp, which is a real number of seconds since system initialization and may be in the future. The destination node and slot id are used for routing. For non-scalar messages which are about to be sent or have arrived at their destination the data field doubles as a pointer to the message body. While in transit this field is not used. Once a message has arrived at its destination node, a schedule entry is created which consists of

the message header plus a deadline[2] and a next pointer so that the entries can be linked into the slot queues and the pool.

4.4 Scheduling Data Structures

The schedule entries which encapsulate messages which are waiting to be processed are stored in one of the following queues:

The timer queue Messages whose timestamp is in the future are placed in a time ordered list which is managed by a the Timer Queue thread described in Section 4.5.2.

The sys_q and user_q These FIFO queues are part of the slot but are accessed by the kernel via function calls. They contain messages whose timestamps are in the past.

The pool This contains entries which have reached the head of the slot's queue. They may have been partially processed. There can be at most one entry in the pool per slot.

As pointed out in Section 3.3.1, the kernel uses the count, max, critical, box and free fields of the port table entries. When count reaches max, the kernel uses the data field from subsequent messages to overwrite that of the message in the box. The box pointer may contain a message which is waiting in the timer queue or in one of the slot queues. However it is removed as soon as it enters the pool. The reason is that once a message is in the pool an action may have started processing it and it would be dangerous to update the data. When possible, entries from the free list are used instead of allocating new storage when creating a schedule entry.

4.5 Kernel Threads

The behaviour of each kernel thread is now examined in more detail and also its implementation on the Transputer.

4.5.1 The Guardian

The guardian is responsible for managing an input link. It may sometimes be replaced by a device handler which converts data arriving from an external device into testbed messages, however its behaviour when viewed from the outside is

[2]computed based on the message timestamp and the post time supplied by the slot

unchanged, so for now just the case where the input link comes directly from another Testbed node is considered. The guardian has three states[3] as shown in Figure 4.2. When it has no message to deliver, the guardian waits in the Link? (reading the link) state for a message header to arrive. The guardian is then in the state labelled Wait_for_processor. When the guardian receives the processor, it checks to see if the message has reached its destination. If so then it allocates storage for the message body[4], reads the rest of the message, schedules the message[5] and passes back into the Link? state. Otherwise the *next* routing table[6] is checked to determine which output link (i.e., which sender thread) it should be sent to. An entry is then added to the back of the queue for the sender on that link. If the queue was empty then the sender is reawakened and the guardian blocks until reawakened by the sender after it has forwarded the message.

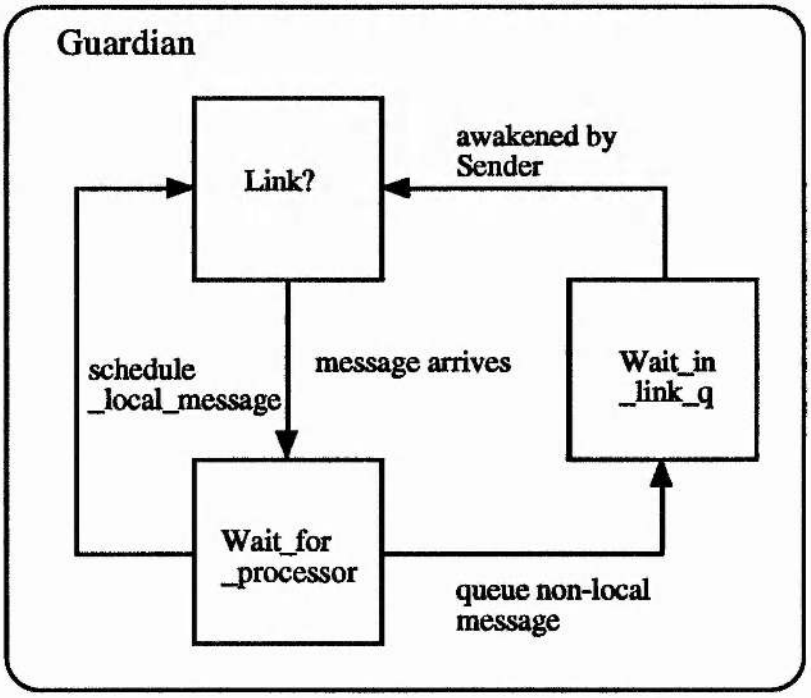


Figure 4.2: The guardian thread.

Implementation Notes

- On the Transputer the link input operates asynchronously with the processor and results in the thread performing the input (the guardian) being inserted

[3] In the state transition diagrams in this section, only states in which the thread may block are shown. The transitions which take it between these states are always executed without interruption.

[4] see Section 2.8

[5] see Section 4.6

[6] see Chapter 5

into one of the processor's two queues. In this case the high priority one. This corresponds to the Wait_for_processor state.

- The sender is awoken by sending it a message on a special channel.
- Each slot and link guardian has an entry in an array of process frames. These frames form the entries in the link_q structure. They contain, among other things, a pointer to the message to be sent and a channel. The guardian blocks by performing an out operation on this channel and is restarted by the sender performing a matching in.

4.5.2 The Timer Queue

Messages whose arriving at the node or being generated by an action sending a message to another local slot (or its own slot), may have a timestamp which is in the future. Such messages are considered not to have arrived yet and their schedule entries are placed in the timer queue until the time has passed. This allows actions to be scheduled (perhaps periodically) at some future time. When this time occurs they are inserted into the schedule as if they had just arrived at the node or just been sent (in the case of internal messages). Entries are added to the timer queue by guardians or the executive. The timer queue is managed by a thread which behaves as illustrated in Figure 4.3. The timer queue manager waits for either a timeout for the entry at the head of the queue (shown as an input from Timer) or a notification from another kernel thread (the guardian or executive) that a new entry has been added to the front of the queue, requiring the current timeout to be cancelled and an earlier one requested. Once a timeout has occurred and the queue manager becomes active it schedules a local message for the appropriate slot[7].

Implementation Notes

In the present Transputer implementation that processor's built in high priority timer is used. The timer queue manager is a high priority process which performs either an alt or a timalt[8], depending on whether there are any entries in the queue, on the start_timer channel. New messages are inserted in order in a linked list of schedule entries. When a new entry is added at the head of the queue by the executive or guardian it outputs on this channel, awakening the timer manager which then starts waiting for the new time.

[7] Section 4.6

[8] Meiko C library calls

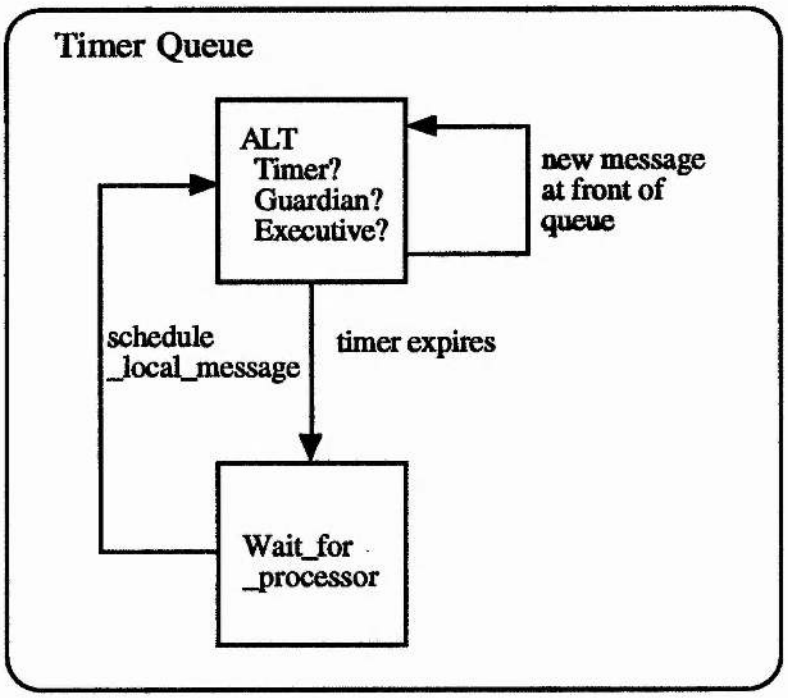


Figure 4.3: The timer queue thread.

4.5.3 The Executive

The scheduling work is actually distributed amongst the kernel threads, however, the executive thread is responsible for descheduling slots and reallocating the processor to the slot whose current message has the earliest deadline, if there are any. During the normal operation of the testbed, the executive has only one blocking state, as shown in Figure 4.4 as sys_chan?. It waits in this state until the current slot (if any) awakens it. This event causes control of the processor to pass atomically to the executive. The signal may have types send, action_end, preemption or terminate. These have the following meanings:

action_end The action which processed the slot's current message is complete (i.e., the function has returned to the action_harness).

send The action wishes to send a message to another (or the same) slot.

preemption In this case a preemption has been scheduled by a guardian, timer or sender thread and the action has reached a point at which it is safe for this to occur. Note: the preemption mechanism is fairly platform specific, see Section 4.6.2 below.

terminate This is a special case in which the testbed is to be shut down, it would normally only happen during development.

Figure 4.5 gives a pseudocode description of what each of these entails. The schedule_local_message subaction is shared with the guardian, timer and sender threads. Note that when called from the executive no_current is true, so the slot with the earliest deadline (if any) is subsequently activated.

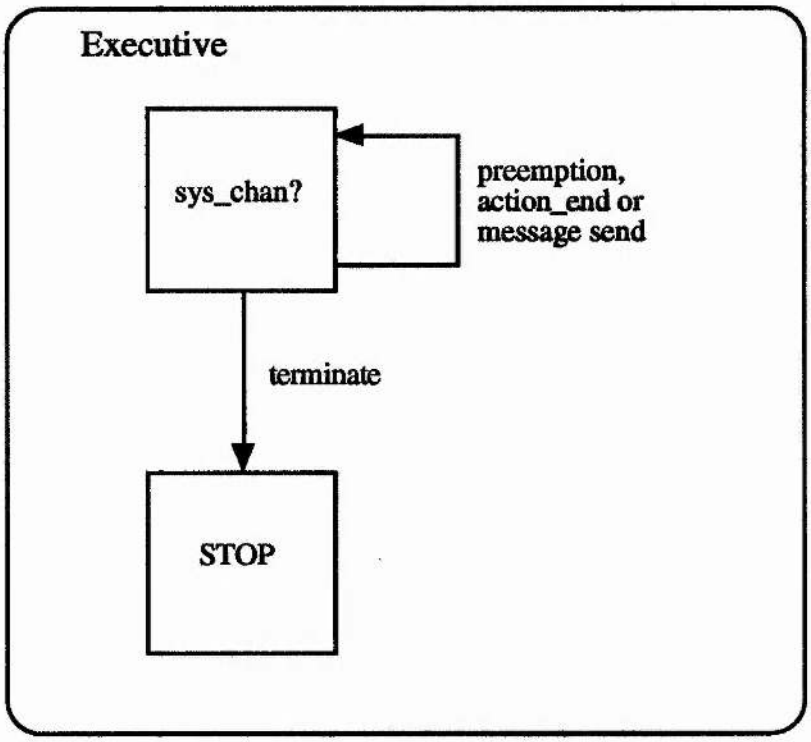


Figure 4.4: The executive thread.

Suspending and Resuming the Slot

The slot may suspend when it sends a message, when the current action returns or when it is preempted. In the first two cases the slot suspends itself by sending a message on a special channel (sys_chan). In this case the executive is awakened, without performing an input (as it performs an alt wait). It then immediately copies the slot's workspace pointer from the channel into the wptr field in its frame entry. In the case of a preemption by the executive, the slot's workspace pointer is removed from the Transputer's low priority process queue and stored in wptr. Slots are resumed by the system performing an input on the wptr field of their frame as if it were a channel. The system behaves differently for slots with new messages, as will be described in section 4.6.2.

```
send:

    set_no_current;
  if local_message then
      schedule_local_message
  else
      add_slot_to_link_q;
    if link_q_length = 1 then
        wakeup_sender;
      activate_slot_with_earliest_deadline

action_end:

    set_no_current;
  if slot_queue_non_empty then
      insert_next_in_pool;
      make_current_in_slot;
    activate_slot_with_earliest_deadline

preemption:

    save_current_slot_context;
    insert_current_in_pool;
    activate_slot_with_earliest_deadline

terminate:

  STOP

schedule_local_message:

  if messages in schedule for this port = max then
      update_data_in_boxed_msg
  else
    if msg.timestamp > now then
        add_msg_to_timer_queue
    else
        insert_msg_in_slot_queue;
      if no_current then
          activate_slot_with_earliest_deadline
      else if deadline < current_deadline then
          schedule_preemption
```

Figure 4.5: Pseudocode for actions of the executive.

4.5.4 The Sender

The sender multiplexes messages onto an output link. As with the guardian, it may be replaced by a device handler which converts Testbed messages into a format understandable by an external device. The sender has two blocking states as shown in Figure 4.6. When the queue is empty it waits to be signalled by either a guardian or the executive which will have added an entry to the queue, causing it to enter the Link! state in which it repeatedly sends the first message from the queue until the queue becomes empty. The messages are sent on behalf of either a slot, in the case of a locally generated message, or a guardian, in the case of a message for a different node. In the former case the slot is rescheduled by placing its current message back into the pool, possibly also scheduling a preemption if the message has the earliest deadline. In the latter case the guardian is signalled and resumes listening for new messages on its link. Through routed messages are wormholed [19], by repeatedly reading short packets (flits) from the input link and writing them to the output link. This has many advantages over store-and-forward routing as discussed in Chapter 5.

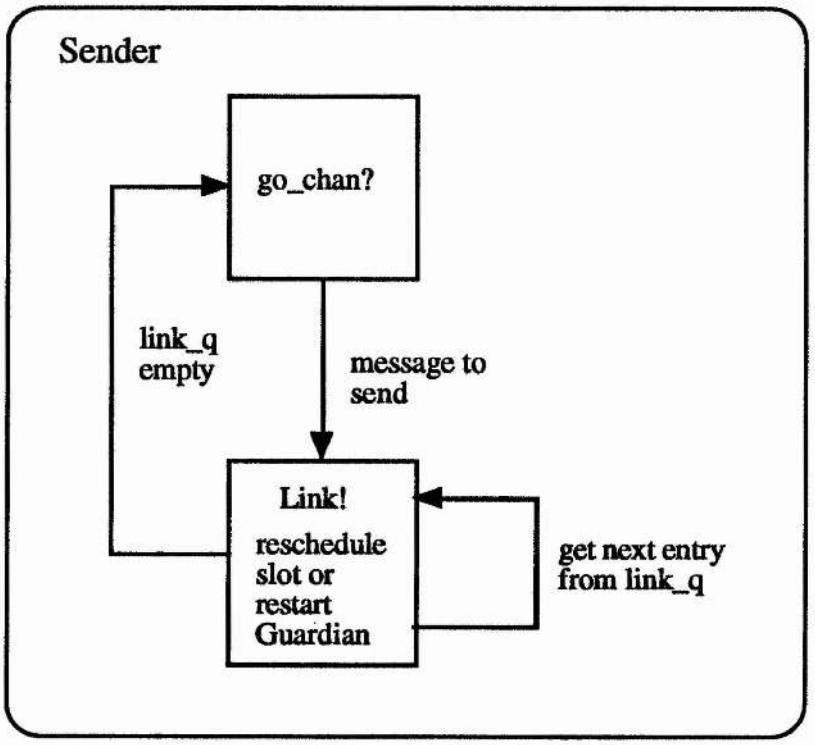


Figure 4.6: The sender thread.

Implementation Notes

The entries in the link queue are actually the elements from the frame array for the slots and guardians which are waiting to send. Each entry has a pointer to the header of the message being sent, a next pointer and in the case of a slot, a pointer to the schedule entry for the message it is currently processing. The latter allows the entry to be returned to the pool. In the case of a guardian, the thread is restarted in the same way that slots are restarted by performing an input on the special channel field of its frame. For slots this also happens if the slot has the earliest deadline and there is no current slot or the current slot is in a preemptable state.

4.5.5 Devices

As described in Section 2.4, a synchronous device looks like a slot to the application. In fact most parts of the system also see the device as a slot on the node to which it is connected. The slot id (known as a *device id*) is provided by the application as part of the network configuration when a link is reserved as either a raw link between two nodes or an external link to a device[9]. In the former case two distinct device ids are provided, one for each end of the raw link, while in the latter only one is required. Although these cases are different during configuration, they appear identical to the system which is oblivious to the connection between the device ids of each end of a raw link. Henceforth the terms "device" and "raw link" shall be used interchangeably.

When a message is sent to a device which is attached to a remote node, the routing system treats it just as any other message destined for that node. Once the message reaches its destination node, the kernel recognizes that the destination is a device and not a slot and queues it for the sender thread corresponding to the device link. The sender for a raw link is the same as for internal links except that it does not send the message header. The device input thread behaves like a normal guardian except that instead of reading the header at the start of each message it uses one supplied by the application, filling in only the data and timestamp fields. In order for the guardian to read and deliver messages, it must be informed of the size of the message to read and the destination slot and port. This is done by the centre slot in response to a message from an application slot containing the header to be used. Both repetitive and one-off message reads are supported. Initially and after a one-off read the guardian waits to receive a message from the centre containing the size of the next read.

[9] See Appendix A.

4.6 Scheduling in Testbed

Scheduling of slots in Testbed is dynamic, using a combined earliest deadline and priority strategy. Actions may be either preemptable or non-preemptable according to a flag in the slot's port table. Unlike most systems which support deadline scheduling, deadlines are not mapped onto a fixed set of priorities. Instead the deadline is stored in the schedule entry and the pool of available messages/slots is kept sorted by this value. Where no particular deadline is appropriate, a priority is assigned to the message instead. This is implemented by using values beyond the range allowed for deadlines. Messages with deadlines are always chosen in preference to those without. This means that at the implementation level, the system does not need to treat messages with priorities differently from those with deadlines, the latter simply come after the former in the pool, with higher priorities having a lower deadline value than the lower ones. The deadline is computed when the schedule entry is created, based on the timestamp contained in the message header and the post time variable of the destination slot. If the post value is greater than the upper bound for timestamps[10] then this value becomes the deadline, otherwise the deadline is the sum of the timestamp and the post value.

Although deadline/priority scheduling is performed among slots on a Testbed node, the messages queued within a particular slot are stored in FIFO order. This may lead to priority inversion, as messages may arrive in a different order to their deadlines. This is compounded by the timeout mechanism which inserts new messages whose timestamp is in the future into the timer queue until after this time has passed, when they are inserted into the slot's schedule as if they had just arrived. Thus messages may be overtaken by others which might have a lower priority or later deadline. Messages with timestamp in the future do not delay other later arriving messages in order to allow a slot to perform work between scheduled activities. Messages are kept in FIFO order in the slot queues in order to reduce overhead and, more importantly, to avoid compounding degradation during overload conditions and to make it simpler to reason about delays as explained below. It should be noted, however that even if the messages were sorted in the slot queue into deadline/priority order, priority inversions could still occur as preemption of actions *within a slot* is not allowed.

4.6.1 Bounded Delays

It is very important that an embedded system application programmer be able to place upper bounds on resource requirements of the application. To support

[10] which is chosen to be a time so far in the future that it is highly unlikely that it will be required

this the kernel must behave in a deterministic way and simple bounds must be obtainable on the resources it requires, such as memory and time. This is one of the main aspects distinguishing an embedded/real-time kernel from a conventional operating system or even microkernels designed to support conventional operating systems. It is often necessary to sacrifice some flexibility and performance in order to achieve these ends.

An example of such a compromise is that messages are not sorted by deadline/priority in the slot queue as this would cause a delay which is dependent on the number of messages queued. Thus the performance of the system would continue to degrade indefinitely as more messages arrive which cannot be processed[11]. Similarly the less flexible memory pools method[12] is used rather than conventional malloc style memory allocation scheme for messages. With the FIFO behaviour and the memory pool allocation strategy there is a constant upper bound on the time required to insert a new message into a slot's schedule while the slot is currently processing an earlier message and a time dependent only on the number of slots on a node to insert a message into the pool[13].

4.6.2 Implementation of Preemption

Preemption can occur under three circumstances:

- When a message reaches the head of a slot's queue having an earlier deadline than the slot which is currently active.
- When a message waiting in the timer queue becomes available and its destination slot has no ready messages.
- When a slot has been suspended while a message was sent outside the node, but is now ready to resume.

If one of these occurs then a preemption is scheduled. If the action which the preempting slot wishes to run is critical, then the current slot's schedule entry is saved and the preemption occurs immediately. The preempting slot's action then runs until completion or until it sends a message at which point the preempted slot is allowed to continue. Note that this type of preemption can only occur if the current slot is executing a preemptable action. During critical actions the system thread[14] which is responsible for introducing or reintroducing the preempting slot

[11] up to the point where storage is exhausted of course

[12] see section 2.8

[13] as each slot may have at most one message in the pool

[14] guardian, timer or sender

is blocked from running. This state holds until the critical action sends a message or terminates, at which point any waiting system threads run in the sequence in which they were triggered.

If the action which the preempting slot wishes to run is itself interruptible then a more elaborate, time consuming and less predictable preemption scheme is required. This is because the T800's high priority processes (used to implement the Testbed kernel threads and critical actions) cannot access all of the state of the current low priority process (used to implement interruptible actions) which they have interrupted. Fortunately there are *descheduling points* in the code at which only that part of the state which is accessible to the high priority processes can be in use. These are also the points at which the low priority processes are timesliced[15] by the Transputer's microcoded scheduler. Using a method described in [87] a dummy process is inserted in the Transputer's low priority queue which simply waits for the current action to be timesliced (ensuring that it is at a descheduling point) and then immediately communicates with the executive thread. This is able to read the head pointer of the low priority queue which points to the workspace of the timesliced action. All the state has been saved in this workspace by the Transputer and the executive thread simply needs to adjust the workspace slightly to produce the effect of the action having executed an OUTBYTE instruction. Consequently it can be resumed using an input instruction just as if it had voluntarily relinquished the processor.

Unfortunately it is difficult to place an upper bound on the time required to perform a preemption in the case where the preempting slot wishes to run an interruptible action. This is because the low priority process is not preempted by the Transputer immediately after its timeslice expires, but rather when it reaches a descheduling point. This is usually not long as most flow control instructions are descheduling points. Nevertheless in a pathological case a program could contain a sequence of non-descheduling instructions limited only by memory. Alternatively it could contain block move instructions which require time proportional to the length of the data to be moved. Although these can be interrupted by high priority processes the low priority process cannot be timesliced while one is in progress. Even without these pathological cases, the timeslice period is 1ms, so this type of preemption can be expected to take more than 0.5ms on average in addition to the normal system overhead. This is no doubt part of the reason that Inmos provided the high priority process as an interrupt mechanism in the first place and also why Testbed allows critical actions to preempt interruptible ones using it.

[15] although timeslicing does not occur in Testbed as at most one low priority process is allowed at a time

Note that the system delay during preemption could be regarded as a short-term priority inversion. If this delay cannot be bounded, then the assumptions on which verification of deadlines [3, 39] is based do not hold. For this reason it may be necessary to use critical actions in critical situations. In this case priority inversions may be of longer average duration but can be bounded, assuming that the durations of critical actions can be. Assuming that there is no other slot performing critical actions and the destination slot has an empty queue the time to invoke a critical action is bounded by the sum of the maximum execution time of each kernel thread, since in the worst case each thread may be waiting to run. Most of these have bounds that are linear in the number of slots on the node since the only variable length computation which they perform is to insert a message into the pool, which in the worst case may have an entry for each slot. The kernel thread may take longer as local message transfers involve making a copy of the message data. Both the number of slots and the sizes of local messages are available to the application designer and can thus be used to calculate an upper bound on the interrupt time for critical messages.

4.7 Initialization

The computing surface is configured and the initial system code is loaded by the CSBuild program `testbed`. This reads a file containing forwarding tables for each node, from which it computes the connection topology actually required. The channels, node id and the next table are passed to the startup process for each node using the import facility.

Upon startup the startup process at each node initializes tables, spawns the link guardians, senders, the clock process (see below) and creates the centre slot. It also sends messages which initiate the clock synchronization and application loading.

4.7.1 Clock Synchronization

Local node time is maintained by a global double precision floating point variable (base_time). The value of this plus the transputer high priority clock (which ticks every microsecond) are used to compute the current time in seconds. The global clock is initialized to zero and is incremented when the transputer timer wraps around to zero. This is done by a high priority process launched at startup which waits on this time. System functions accept and return times which are double precision real values and represent seconds since system initialization, and these are also passed as the timestamp in messages. If a message arrives with a

timestamp in the future, the slot is added to the timer queue. The timestamp is converted to an integer number of ticks (microseconds) of the high priority timer (modulo 2^{32}). The timer thread then waits for this time. If the time is more than the resolution of the high priority timer in the future then the wait is performed several times.

The clock on each node is synchronized with that of the root node by the centres before any other slots are created. This is achieved by sending a message from the centre slot on the root node to the centre on the node to be synchronized. A timestamp is recorded on the root node when this message is sent (timestamp t_0) which is echoed by that slot with data field set to the local time of the receiver (timestamp t_1). When this message arrives back at the root it gets a third timestamp t_2. Based on the assumption that message latency is almost the same in each direction, the approximate difference between the two clocks is then calculated as $(t_0+t_2-2t_1)/2$. This value is sent to the centre on the node to be synchronized and used to adjust the local timer. Experiments suggest that this method synchronizes clocks to within 10μs, which is far less than the minimum message latency between nodes (about 0.4ms), which means that timestamps can be generated which are causally consistent[16]. No significant relative drift has been found between the clocks over time.

4.8 Conclusions

The BED kernel is designed to support the Testbed programming model and its development and debugging goals. It follows the Microkernel philosophy of supporting only a bare minimum of functions required by all applications, with other functionality provided at higher levels. In Testbed this means that they are provided as system actions or slots.

The kernel provides an asynchronous message delivery service in which the application need not know the location of the destination slot. Both deadline order and priority based scheduling are provided for messages and in addition the timer queue allows the action to be delayed. The overheads introduced by message delivery and scheduling are strictly bounded and predictable due to the memory pools storage allocation scheme. As storage for each slot is allocated in advance and the system only makes dynamic allocations from these areas, the slots are independent. Thanks to the FIFO queuing policy at each slot, time for message delivery and scheduling does not increase as the number of messages waiting at the destination increases. The bound on scheduling time is proportional to the

[16] See Section 6.3.1 (Patterns of Events) and [51].

number of slots on a node due to the earliest deadline scheduling among slots.

Chapter 5

Routing

5.1 Introduction

Testbed is intended to run on configurable distributed memory multiprocessors, such as Transputer networks. Although the current system is implemented on a Meiko Computing Surface, for which a message passing library is provided[1], it was decided that Testbed should have its own low-level communication harness. Little information is available on how the Meiko communication mechanism works, making it difficult to predict what impact routing overhead may have on performance. By implementing the communication layer using low level facilities and tailoring it to Testbed's programming model efficiency and control over routing were gained.

Sometimes the paths followed by messages (the routing), will be critical to the application and will be specified explicitly by the designer along with the interconnection topology. In this case verification of correctness properties such as freedom from deadlock and real-time properties of the communication will be part of the application design. In other cases it will be desirable for the system's routing harness to be verified free from deadlock and livelock independently of the application. This chapter describes the Testbed network architecture, discusses criteria for deadlock-freedom, gives a simple algorithm for finding a near-optimal deadlock-free routing for a given topology and explores ways of constructing regular networks from simple building blocks and extending path-optimal and deadlock-free routings to cover them.

Most current work on network routing favours dynamic schemes such as randomized or adaptive routing in which messages may follow more than one route between the same source and destination. These schemes improve the average performance by avoiding bottle-necks, however it becomes difficult to predict

[1] CSN as described in [61]

the impact of communication overheads on a specific application. Embedded systems often have real-time constraints, so it is more important to preserve the predictability of communication delays than to achieve good average performance. This suggests that static routing is more appropriate than dynamic routing. Low communication delays are often a desirable feature in embedded systems, so it would be preferable that messages do not take unnecessarily long routes to their destinations and that the overhead imposed by the routing mechanism is low. Given the constraint of static routing bottle-necks in the system should also be avoided by spreading the load evenly among the communication links.

5.1.1 Finding the Routing

Given an arbitrary connection topology, a message routing method which is deadlock free is required. At the same time it should have the following features:

1. Messages between any two nodes should follow as nearly as possible the shortest path between the nodes given by the connection topology.
2. Minimal overhead is introduced by the deadlock avoidance.
3. Messages should not be lost due to lack of resources within the network, though this may happen if there is insufficient storage at the destination.
4. Load should be as evenly distributed across links as possible, to reduce bottle-necks.

The following approaches to constructing routings in a way which avoids the possibility of deadlock in the network have been considered:

1. Compute a deadlock free (though possibly inefficient) routing function based on a single spanning tree of the network. Then, as discussed below, a set of spanning trees which is near optimal but still deadlock free can be found by improving on this. Section 5.4 discusses this approach.
2. Allow the user to provide a routing function, which may be checked for deadlock freedom. This may be acceptable to the user for small networks or large regular networks, for which a good routing scheme is known (such as the e-cube algorithm for hypercubes) but it would be tedious for a large irregular network. This method may be appropriate when the user has information about the pattern of communication of the application, which is difficult to embody in an algorithm for constructing a routing.

3. Use a deadlock avoidance scheme. This allows any optimal routing to be used at the cost of introducing extra overhead which may be unnecessary and undesirable in an embedded system. Several such schemes are discussed in Section 5.1.2. Protocols that allow two messages for the same destination to take different routes or that allow overtaking within a node introduce the possibility of livelock.

4. Provide a library of deadlock-free optimal (or near optimal) routing functions for certain common types of network topology. If one of these is embedded in the network then it may be used, or the user may construct a routing function by connecting up such components. The result is then checked to make sure it is still deadlock free. Section 5.5 discusses some methods of constructing complex networks from simple building blocks while preserving both path-optimal routing and deadlock freedom.

5.1.2 Related Work

The most common solution is to construct a routing algorithm which is deadlock free for any given routing function [19, 30, 96]. Unfortunately all of these methods impose some extra overhead and some have other drawbacks, such as discarding of messages.

Many such algorithms are based on *structured buffer pools*. Each node has its buffers partitioned into classes upon which a partial ordering is imposed. [96] describes such an algorithm in which each node has $M + 1$ buffers, where M is the maximum number of hops any message should have to make before reaching its destination. The buffers are numbered from 0 to M. Each message enters the network in a buffer numbered 0 and in moving from one node to the next it may only move to a buffer numbered one higher than the one it is in. If a message gets to an M buffer but has not yet reached its destination then it is discarded. This could be thought of as replacing the routing network by one consisting of M layers in which messages enter at the bottom layer and progress up one layer per step. It is easily seen that this network has no cycles! This method has the disadvantage that messages are discarded after M hops, so the routing algorithm must ensure that a message reaches its destination (or is otherwise dealt with) within this limit. Also the larger the network, the larger the number of buffers required at each node.

The structured buffer pools technique is not suitable for the wormhole technique introduced by Dally and Seitz [19] as in wormhole routing the packets (called *flits*) of different messages cannot be interleaved. They present the virtual channel technique which is also based on resource ordering. When cycles are present in

the channel dependency graph, virtual channels are introduced which are multiplexed over existing channels each with a separate buffer. By introducing enough virtual channels the cycles can be broken. Because less than M virtual channels (and hence buffers) may be required and because they are only required where actual cycles exist in the dependency graph and not other parts of the network, the resulting algorithm may be more efficient on resources. Another advantage of this method is that messages need never be discarded. It has the disadvantage that a different algorithm must be worked out by hand for each type of network. Work continues on virtual channel routings for specific classes of network [22, 55].

[30] describes a different type of algorithm in which the underlying undirected network G is partitioned into two directed graphs. This is done by constructing a directed graph G_1 which has the same nodes and arcs as G and for which there is one special *root* node with only outgoing arcs. The other G_2 is the inverse of this (i.e., with the directions of the arcs reversed). Then whenever a packet arrives at a node which has only messages waiting to go out on G_1 edges and which has only one free buffer, either that message or one of the other ones is rerouted on to a G_2 edge. The *root* node is given more buffer space than all other nodes and is forced to accept any incoming message in finite time even if that means overwriting one of its buffered messages. This algorithm has the advantage that when things are running smoothly, there is no extra overhead imposed. Also all buffers are available for any message. It has the disadvantage that messages may be lost.

5.2 The Testbed Network Architecture

Testbed communication networks consist of Transputers connected by bidirectional links. Each link is controlled by sender and guardian kernel threads[2]. Each sender forwards messages on its link on behalf of guardians or local slots, while each guardian receives messages from its link and then routes them to an appropriate output link or local slot[3].

Testbed uses a static routing scheme in which there is a spanning tree centred at each node. Each message contains both the destination node and slot id. When a message is initially sent, the application normally provides the slot id and the system uses a location table to look up the destination node. If the destination is non-local, then the routing harness uses the destination node to forward the message. The destination slot id is not used until the message has arrived at the

[2]described in Chapter 4

[3]See Figure 4.1.

correct node. To route from node x to node y, the message is sent to the parent node of x on the spanning tree centred at y. Each node has a table which gives the next link on the spanning tree for every other node.

5.2.1 Wormhole Routing

Store-and-forward routing (sometimes called packet switching) is the more traditional approach in which each packet is treated by the network as a separate message with its own header. Each node receives the entire packet and stores it in a local buffer. It then uses information in the header to forward the packet towards its destination. In wormhole routing the packets (called *flits*) which make up a message are not interleaved. A routing decision only needs to be made for the first flit, determining which output link is to be used and once that link is available the entire message is sent before any other message may use the same link. While a message is being through routed, an input and an output link are dedicated to that message and flits are alternately read and written until the entire message has passed through. [29] and [70] give useful overviews of wormhole routing and of hybrid techniques such as virtual cut-through in which messages are routed in wormhole fashion until a busy link is encountered at which point the message is removed from the network into a buffer until the link is available.

With store-and-forward routing either the size of messages must be limited by the amount of storage which can be spared at each node for buffers or multiple packet messages must be used. In the latter case there is the problem of reassembling these messages which complicates the receiving process. Also buffers must generally be large to get reasonable efficiency as a routing decision must be made for each. With wormhole routing the buffers can be smaller, creating a pipeline effect which reduces message latency. For these reasons Testbed uses wormhole routing. However the results derived in this chapter hold for both wormhole and store-and-forward routing.

5.2.2 Properties of Testbed Networks

Testbed interconnection networks have the following properties:

1. A message arriving at its destination node is always consumed[4].
2. The route taken by each message is uniquely determined by the source and destination nodes to be a finite path from source to destination.

[4] although it may be discarded if the destination slot has no free storage for messages

3. There is one flit buffer for each input link.

4. Any message waiting for an output link will eventually receive it if no deadlocks occur[5].

Properties 2 and 4 imply that there can be no livelocks in the routing as there is no infinite overtaking within a node and messages cannot wander forever through the network.

It will be seen that these properties of the behaviour of the physical nodes allow us to derive a simple criterion for deadlock freedom of a routing function.

5.2.3 Broadcast

Because Testbed routing functions use the spanning trees for each node strictly in the direction of the root, broadcasts cannot simply use the same trees outwards, since the combination of directions might produce deadlock cycles. Instead a broadcast tree is created for each node by forming the union of the paths from that node y to each other node x produced from x's spanning tree. Then each node z holds for each possible source node y, the children of z on y's broadcast tree. When z receives a broadcast packet from y it sends it on to each child. Figure 5.1 shows the spanning tree for node x, in which the path from y to x, passing through z forms part of y's broadcast tree. z has child x in that tree.

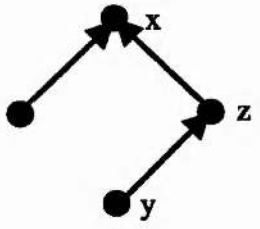


Figure 5.1: Constructing a broadcast tree for y from the spanning tree for x.

5.3 Deadlock-Free Routing Functions

Definition 5.1 Routing deadlock *occurs when there is a cycle of processes each of which is blocked trying to send a message to the next.*

In Testbed the blocked processes are the guardians. Although there are also sender and slot processes involved, ultimately each blocked guardian is waiting

[5]This is because the kernel queues each message being sent on a link and deschedules the sending process (including guardians) until it has been sent.

for a guardian on an adjacent node to receive its message. Guardians do not block waiting to send to slots and each blocked guardian has an associated sender for the link which it wishes to use. For these reasons these other processes are ignored and the deadlocked cycle is considered to be made up of guardians.

Definition 5.2 *A* network *is a directed graph* (N, L) *with nodes* N *and edges* L*, which are called links. Denote the link from* p *to* q *by* $[p, q]$*. Links from a node to itself (*loops*) are disallowed. A* routing *is a function* $R : N \times N \mapsto N$ *which given a node* i *and a destination node* j *gives the next node* $(R(i, j))$ *from* i *on the path to* j*.*

This type of routing function has the effect of defining a set of directed spanning trees rooted at each node, consisting of the paths from all other nodes to that node. The advantage of this type of routing function is that it only requires that each node know the next node on the path to any other node, and that the messages contain their destinations.

Let R be a routing for network (N, L) in the following

Definition 5.3 *A cycle of links in* L*, such that for any three successive nodes* p*,* q *and* r *in the cycle, there is some node* s *for which* $R(p, s) = q$ *and* $R(q, s) = r$*, is known as an* overlap-cycle.

In other words an overlap-cycle is made up entirely of overlapping segments of spanning trees and contains no nodes for which the sets of spanning trees containing the incoming and outgoing arcs are disjoint.

Theorem 5.1 *Deadlock can arise in the routing processes if and only if* R *has an overlap-cycle.*

Proof: ($\Leftarrow$) Suppose such a cycle exists, then because any two adjacent arcs in the cycle must both lie on at least one spanning tree, it is possible that each node in the cycle has the header of a message which came from the previous node and is destined for the next node. When this happens, a deadlock exists since the guardian process at the end of each link in the cycle is blocked waiting for the one at the end of the next link.

($\Rightarrow$) Suppose that there is a routing deadlock, then there must be a cycle of links each of which has a guardian which is blocked waiting to send a message on the next. This means that each guardian has a message whose route includes two hops in the cycle so this must be an overlap cycle. □

Figure 5.2 shows two alternative routings for a four cycle. The shaded arrows indicate the links which may be used by messages destined for the node with the

same shading pattern. The left hand routing has no overlap cycles and thus is free from deadlock by Theorem 5.1. The right hand routing has an overlap cycle and the diagram below shows a deadlock situation in which the guardian at each node has a message which came from its predecessor and is destined for its successor in the cycle. No progress can be made because none of them can pass on its message.

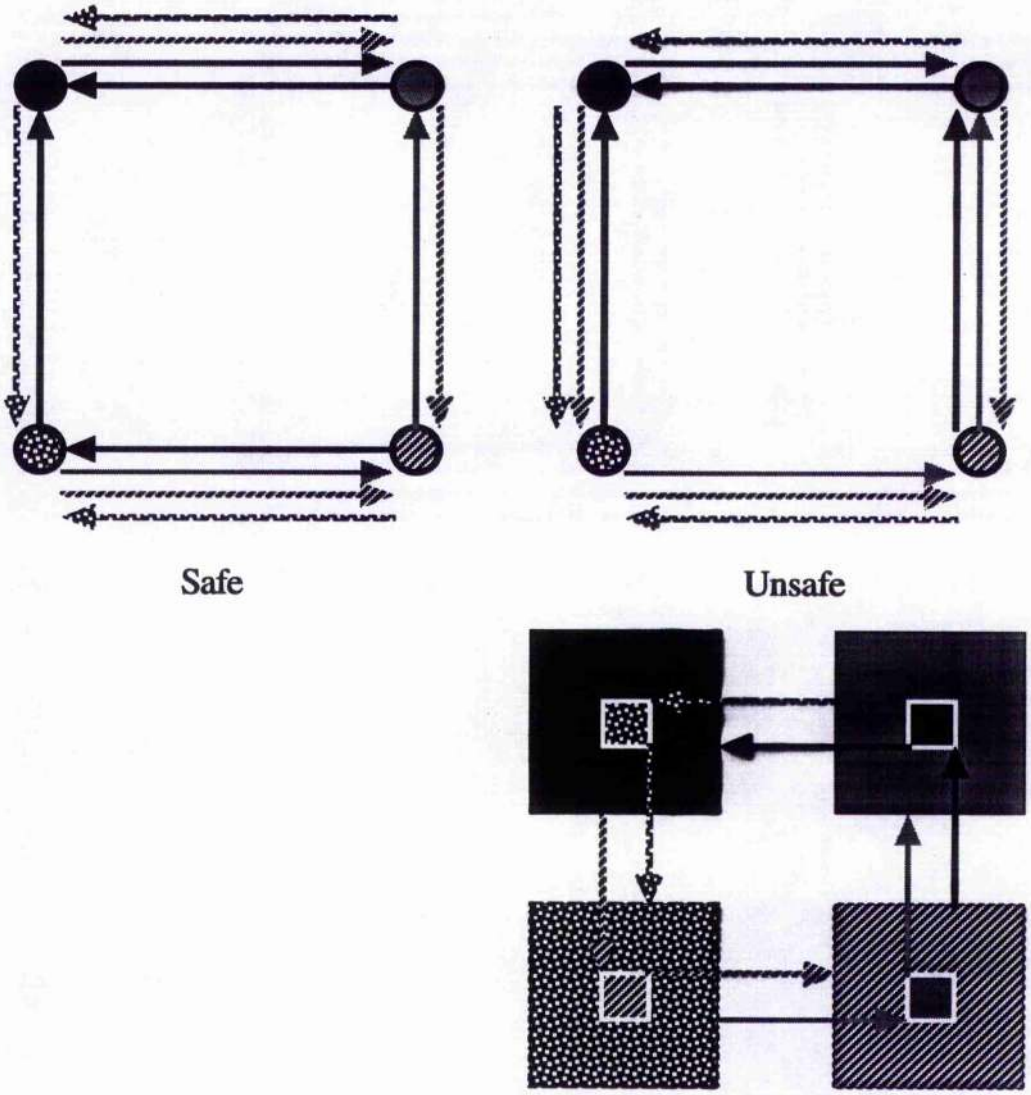


Figure 5.2: Safe and unsafe routings for a four cycle.

5.3.1 Dependency Graph

[19] uses a more general routing function in which the outgoing link which a message follows is determined from the incoming *link* and destination node rather than just the current *node* and destination node. Using this routing function it

is possible to define a *link dependency graph*[6] D as the directed graph whose vertices are the links of the network. Any pair of successive links on a path to some destination node are connected by an edge in D in the direction of the path. Then the routing network is shown to be deadlock free if and only if there are no cycles in D. Testbed's link-independent routing functions are special cases of this more general form and so an acyclic dependency graph provides an alternative criterion for deadlock freedom to the one presented above. It may be possible to construct a routing function of this type which is closer to optimal than any link-independent routing. However considerable extra overhead is required for the more complex routing function.

Since the presence of cycles in the dependency graph is an alternative criterion for potential deadlock, these must be equivalent to overlap cycles in Testbed networks as the following proposition shows.

Proposition 5.2 *A routing has an overlap cycle if and only if there is a cycle in the link dependency graph.*

Proof: ($\Rightarrow$) Given any pair of links in an overlap cycle $[p, q]$ and $[q, r]$ there is some node s for which the route from p to s follows $[p, q]$ then $[q, r]$, so there is a dependency between these links. Thus the links in the overlap cycle form a cycle in the dependency graph.

($\Leftarrow$) Any pair of vertices in a cycle in the dependency graph correspond to links in the network $[p, q]$ and $[q, r]$ say. The fact that there is a dependency between them means that there is some route which follows $[p, q]$ and then $[q, r]$, so there is some s for which $R(p, s) = q$ and $R(q, s) = r$. Thus the cycle in the dependency graph corresponds to an overlap cycle. □

Figure 5.3 shows the link dependency graph for each of the routings from Figure 5.2. The patterns of the edges correspond to the spanning tree on which the pairs of links which make up the dependency lie, illustrating the relationship between cycles in the dependency graph and overlap cycles.

5.4 Optimality

Definition 5.4 *A* path-optimal *routing function is one in which the path from any node to any other node is the shortest possible.*

For some networks it is impossible to find a routing function which is both path-optimal and deadlock-free based on spanning trees. An example is any

[6] actually referred to as a *channel* dependency graph in [19]

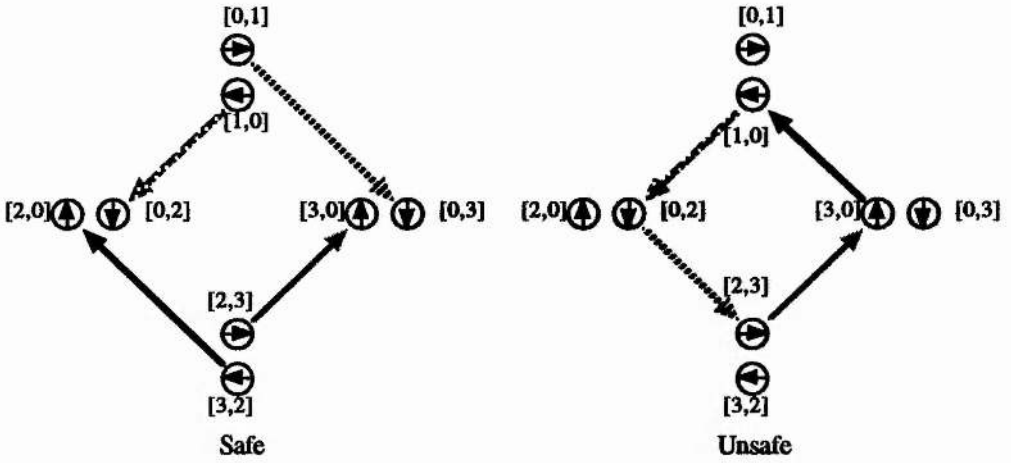


Figure 5.3: Dependency graphs of safe and unsafe routings for a four cycle.

simple cycle with 5 or more nodes[7]. However, assuming that the network consists of bidirectional links, then there is always a (probably non-optimal) deadlock-free routing. This is given by choosing any (undirected) spanning tree and then forcing the directed spanning trees for all nodes to lie on it. This is in effect reducing the network from a general graph to a tree. Because messages do not make U turns, and all the spanning trees lie on the same tree, there can be no overlap-cycles. Once a deadlock free set of spanning trees such as this has been found it can generally be improved, often to the point of optimality or near optimality by making successive adjustments to each spanning tree which do not introduce overlap-cycles.

The following definitions give a measure of the path lengths in a route:

Definition 5.5 *Given any pair of nodes $p, s \in N$. Let $d(p, s)$ be the number of hops from p to s given by R. Then define the* path-compactness *of the spanning tree for s by*

$$\phi_R(s) = \sum_{p \in N} d(p, s) \tag{5.1}$$

and the total path-compactness *of R by*

$$\Phi(R) = \sum_{s \in N} \phi_R(s) \tag{5.2}$$

The algorithm below attempts to minimize the total path-compactness.

The notation for a stepwise change to a routing is as follows:

Definition 5.6 *Denote by $R[p, s \mapsto q]$, where L contains a link $[p, q]$, the routing which is identical to R except that $R[p, s \mapsto q](p, s) = q$.*

[7] See Section 5.6.2.

Proposition 5.3 *If the spanning tree for node s is path-suboptimal then there are nodes p, q, such that $d(p, s)$ can be reduced by changing the next node from p on the path to s to q and the resulting routing is closer to optimal, i.e., $\phi_{R[p,s\mapsto q]}(s) < \phi_R(s)$.*

Proof: Let S be the spanning tree for s and $O \subset S$, the subtree of S consisting of all the nodes whose path to s in S is minimal, then let x be a node outside of O. Let P be a minimal path from x to s and let p be the closest node to s in P outside O. Let D be the length of the sub-path of P from p to s (i.e. the optimal distance). If q is the next node from p on P in the direction of s, then q is in O, so $d(q, s) = D - 1$. Thus by replacing the next node from p in the direction of s in S with q, the length of the path from p to s in S is reduced to D. □

Figure 5.4 shows how a sub-optimal spanning tree for node s can be improved[8] by changing the parent of node p to q, reducing the distance from s of p and each of its descendants. This result means that whenever a routing is path-suboptimal, it can

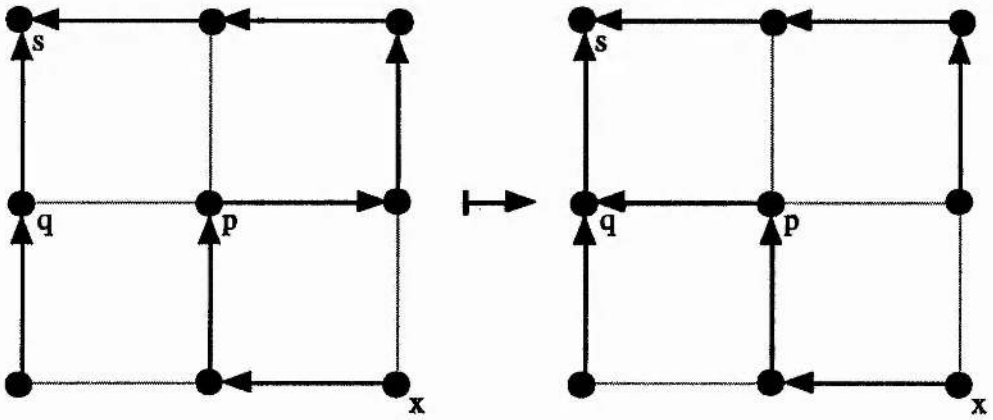


Figure 5.4: Improving path lengths in a spanning tree by a single change.

be improved by changing one link in one spanning tree. If not for the requirement of preserving deadlock-freedom this would clearly lead to a path-optimal routing.

Even though there may be no adjustments which reduce the path from some node to the root of some spanning tree, there may be ones which improve the balance of link load without lengthening the path. This load is defined as follows:

Definition 5.7 *The* load *on a link is the number of paths between pairs of nodes given by the routing function which include that link.*

The balance of link load is measured in terms of the statistical variance:

[8] in fact made optimal in this case

Definition 5.8 *The* load variance $\Lambda(R)$ *of routing* R *is the statistical variance of the load over all links.*

Now it is possible to define a way of comparing two routings which classifies first by total path-compactness and then by load variance.

Definition 5.9 *Given a network and routing functions* R *and* R'*, define a strict partial order* $\ll$ *by*

$$R' \ll R \quad \textit{iff} \quad \Phi(R') < \Phi(R) \textit{ or } \Phi(R') = \Phi(R) \textit{ and } \Lambda(R') < \Lambda(R).$$

This is the lexicographic order of Λ within Φ.

5.4.1 Routing Optimization

The following is a greedy algorithm which given a set of spanning trees for the network with no overlap-cycles, progressively makes any refinements which do not introduce overlap-cycles. The initial spanning trees are assumed to be given, but they may be computed by finding an undirected spanning tree rooted at some node in the network and basing all the directed spanning trees on this.

```
Repeat
  for each node p
    for each spanning tree s
      for each adjacent node q to p
        if R[p, s ↦ q] ≪ R and R[p, s ↦ q] has no overlap-cycles then
            R := R[p, s ↦ q]
Until no change
```

Since the relation $\ll$ gives preference to reducing path length over reducing the load variance, the latter may actually be increased by some steps of the algorithm. Note that any ordering on routings could have been used instead of $\ll$ in the algorithm above.

Figure 5.5 shows an example of the application of the algorithm to a 5-cycle network. The spanning trees in the top row are the initial solution based on the optimal spanning tree for node 0. The bottom row of spanning trees are produced by one iteration of the algorithm. No further improvements are possible without introducing overlap-cycles. Section 5.6.2 discusses cycle networks further.

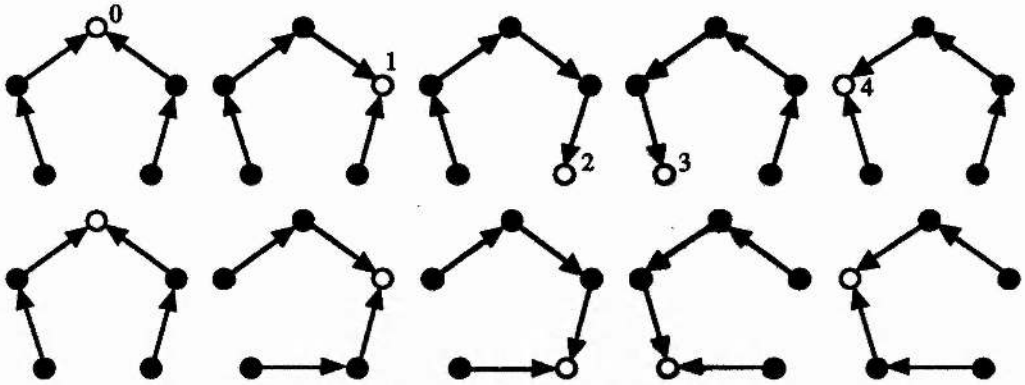


Figure 5.5: Routing algorithm applied to a 5-cycle.

5.4.2 Local Minima

It is possible that the algorithm may become trapped in a state where no improvements can be made without introducing overlap-cycles, but which is not the best possible deadlock-free solution either from the point of view of path reduction (i.e., Φ is not minimal) or load balancing (i.e., Λ is not minimal) or both. This has been found to occur, both with respect to path-optimality, and even more often with respect to load distribution. From theorem 5.3 it follows that it is always possible to detect whether the algorithm has found a path-optimal solution, since if reductions in path length are possible which don't introduce cycles, they can always be made in single steps. However it sometimes happens that several adjustments need to be made simultaneously in order to achieve an improvement in load-balancing. Since the algorithm only considers single steps, it cannot detect this kind of suboptimality in the load distribution.

It has been found that starting the algorithm with different initial spanning trees affects the final result. One solution might therefore be to run the algorithm several times for the same network, with different starting points. Alternatively a probabilistic technique such as simulated annealing may be used.

5.4.3 Implementation

The algorithm has been implemented and tested on a number of different types of network such as grid, torus, binary cube, cycle and cube-connected cycle (CCC) networks and found to work well as Table 5.1 shows, finding path-optimal solutions, if they exist and generally also balancing the load, or coming close. For many of these types of network it is also possible to construct deadlock-free routing functions using methods described in Section 5.5.

It should be noted (as explained in Section 5.6.3) that there are no routings which are both deadlock-free and path-optimal for a cycle of greater than 4 nodes

Network	Λ(initial)	Λ(solution)	Φ(initial)	Φ(solution)	Φ(optimal)
2×2 grid	2	0	20	16	16
3×3 grid	46.5	0.5	204	144	144
4×4 grid	370.4	12.4	976	644	632
5×5 grid	1788	42.2	3200	2000	2000
3-d hypercube	29.2	0.3	136	96	96
4-d hypercube	276	2.1	784	512	512
5-d hypercube	2254.1	11.9	4128	2560	2560
6-d hypercube	17267	75.5	20544	12288	12288
5-cycle	4.8	0.2	40	32	30
6-cycle	8.7	1	70	58	54
7-cycle	16	4.1	112	94	84
3×3 torus	47	0	204	108	108
4×4 torus	355.3	2.2	976	512	512
5×5 torus	1686.4	25.9	3200	1602	1500
3-d CCC	1961	249.1	3076	2048	1776
4-d CCC	58759	12065.9	35328	24512	18944

Table 5.1: Performance of routing algorithm for various networks.

or a torus of size greater than 4×4. This prevents the algorithm from achieving the optimal value in the right most column of Table 5.1 for these networks.

5.5 Extending Deadlock Free Networks

Graph theory gives us ways of constructing quite complex networks which are built up from simple components. The results in this section show that given suitable components for which the routing problem has been solved, it is possible to construct routings for higher dimension networks built from them.

In this section the definition of a routing R is extended to a collection of paths which are sequences of successive links. Partial routings in which some nodes have no path between them are allowed and there can be multiple paths between nodes.[9] The overlap cycle of the previous section becomes a cycle of links such that each successive pair lies on some path.

Definition 5.10 *Given any two networks $G = (N_G, L_G)$ and $H = (N_H, L_H)$, the* cartesian sum $G + H$ *is the network $(N_G \times N_H, L_G \times N_H \cup L_H \times N_G)$, which*

[9] Although Testbed routings are always complete and only one path exists between nodes.

has a link between nodes (p, q) *and* (r, s) *if either* $q = s$ *and* $[p, r] \in L_G$ *or* $p = r$ *and* $[q, s] \in L_H$.

Example 5.1 Let G and H be simple two node networks with nodes labelled 0 and 1 and a single link $[0, 1]$. Then $G + H$ is the square network with nodes $00, 01, 10$ and 11 and links $[00, 10]$, $[01, 11]$, $[00, 01]$ and $[10, 11]$, as illustrated in Figure 5.6.

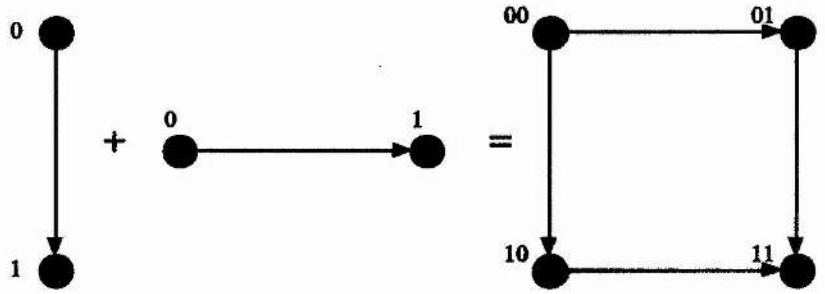


Figure 5.6: The cartesian sum of two simple pair networks.

Cartesian sum networks include many useful regular topologies such as the grid, hypercube and torus.

Definition 5.11 *Given any two networks* G *and* H *as above, the* normal product $G \cdot H$ *is the network* $(N_G \times N_H, L_G \times N_H \cup L_H \times N_G \cup L_G \times L_H)$, *which has a link between nodes* (p, q) *and* (r, s) *when one of:*

$$q = s \text{ and } [p, r] \in L_G,$$

$$p = r \text{ and } [q, s] \in L_H,$$

or

$$[p, r] \in L_G \text{ and } [q, s] \in L_H.$$

Notice that neither of these constructed networks has loops.

Example 5.2 Let G and H be as in example 5.1. Then $G \cdot H$ is the square network with nodes $00, 01, 10$ and 11 and links $[00, 10]$, $[01, 11]$, $[00, 01]$, $[10, 11]$ and $[00, 11]$, as illustrated in Figure 5.7. If the link in both G and H is bidirectional then $G \cdot H$ is a completely connected 4 node network.

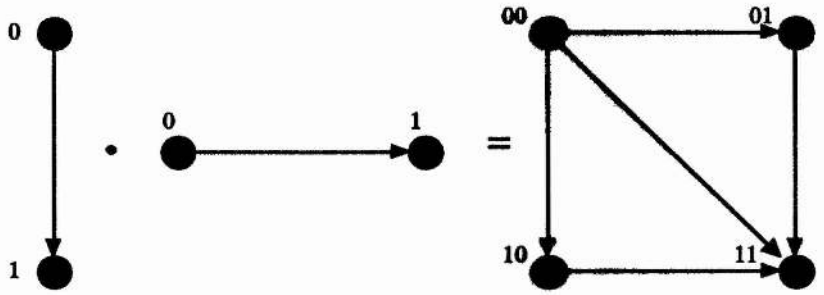


Figure 5.7: The normal product of two simple pair networks.

5.5.1 Combination Routings

In the following definitions and theorems G, H and I are taken to be networks with $G + H \subseteq I \subseteq G \cdot H$ also R^G, R^H and R to be routings in G, H and I.

Definition 5.12 *The* projection P_G *of a path P between two nodes in I onto G is the path defined by mapping edges* $[(p,q),(r,s)] \mapsto [p,r]$ *when* $p \neq r$ *and omitting the edge when* $p = r$. P_H *is defined similarly. The projection routing* R_G *is the set of projections of the paths in R onto G. The* q-copy *of G in I is the subnet*

$$G_q = (\{(p,q)|p \in N_G\}, \{(l,q)|l \in L_G\}).$$

The G-part of path P in I is the subset of links

$$P_G{}' = \{[(p,q),(r,q)] \in L_P | q \in N_H\}.$$

The H-part is similarly defined, while the diagonal *part of P is defined as*

$$P_{GH} = \{[(p,q),(r,s)] \in L_P | p \neq r \text{ and } q \neq s\}.$$

Definition 5.13 *R is a* combination routing *of R^G and R^H when for any pair of nodes $(p,q),(r,s) \in N_G \times N_H$ connected by path P according to R, P_G is a path between p and r in G given by R_G and P_H is a path between q and s in H given by R_H.*

Unfortunately not all combinations of deadlock-free routings are deadlock-free. For example the right hand routing in Figure 5.2 has an overlap cycle even though it is a combination of the routing for the simple two node network. It should also be noted that combination routings are not unique. The left hand routing in Figure 5.2 is a different combination for the same network which is deadlock-free.

Proposition 5.4 *R is a combination routing of R_G and R_H.* □

Theorem 5.5 *Given a network (N, L), with L partitioned into a set of white links, L_W and black links L_B, let R_W and R_B be (possibly incomplete) routings on (N, L_W) and (N, L_B) respectively which are both deadlock-free. Then the composite routing R_{WB} of routes which follow first a white route and then a black one (either could be empty) is deadlock-free.*

Proof: Suppose there is an overlap cycle O in R_{WB}. If O contains both black and white links, it contains some black link b followed by a white link w. There is thus a route in R_{WB} of the form $\ldots bw \ldots$, which is a contradiction. Thus O is "monochrome", in which case every pair of links in O lies on a route of that colour, making O an overlap cycle in either R_W or R_B which is also a contradiction. □

This result is based on the well known resource ordering technique for avoiding deadlock. In this technique all processes request common resources in the same order, making deadlock cycles impossible.

Lemma 5.6 *The length of a path P in I, $|P| = |P_G| + |P_H| - |P_{GH}|$.*

Proof:

$$\begin{aligned} |P| &= |P_G{}'| + |P_H{}'| + |P_{GH}|, \\ |P_G| &= |P_G{}'| + |P_{GH}| \end{aligned}$$

and

$$|P_H| = |P_H{}'| + |P_{GH}|.$$

Therefore

$$|P| = |P_G| + |P_H| - |P_{GH}|.$$

□

Lemma 5.7 *If R is path-optimal in I then so are R_G and R_H in G and H respectively.*

Proof: Consider the subnet G_q for some $q \in N_H$, then for any pair of nodes $p, r \in N_G$, R gives a route P which is a shortest path from (p, q) to (r, q) in I. Now

$$\begin{aligned} |P_G| &= |P| - |P_H| + |P_{GH}|, \quad \text{by Lemma 5.6} \\ &= |P| - |P_H{}'| \\ &\leq |P| \end{aligned}$$

Since P_G is a path in G and G_q is isomorphic to G, there is a corresponding path, P_q say, in G_q and $|P_q| = |P_G| \leq |P|$. Consequently P_q is a shortest path in G_q and so P_G must be a shortest path in G. The same argument holds for H. □

Theorem 5.8 *Let R be a routing for $G + H$, then R is path-optimal if and only if R_G and R_H are path-optimal for G and H respectively.*

Proof: ($\Leftarrow$) Let Q be some path between node (p, q) and node (r, s) and let P be a path given by R. Since P_G, is a path in G between p and r given by R_G, Q_G can be no shorter, by the optimality of R_G. Similarly Q_H, can be no shorter than P_H. Thus the total length of Q, $|Q| = |Q_G| + |Q_H| \geq |P_G| + |P_H| = |P|$, by Lemma 5.6. It follows that P is optimal.

($\Rightarrow$) Follows from Lemma 5.7. □

This result shows that there are ways to construct path-optimal routings for cartesian sum networks based on path-optimal routings for the components. In order to get a useful network deadlock-freedom must also be preserved. It turns out that there is a combination routing which has this property.

Corollary 5.9 *If R is a routing for $G + H$ which is both deadlock-free and path-optimal then R_G and R_H are both deadlock-free and path-optimal in G and H respectively.*

Proof: If R is deadlock-free and path-optimal then path-optimality of both R_G and R_H follows from Theorem 5.8, so it suffices to show that they are deadlock-free. Given $q \in N_H$, then for any two nodes $p, r \in N_G$, let P be an optimal route from (p, q) to (r, q) given by R. Since P_G is a shortest path in G and G_q is isomorphic to G, $|P| = |P_G|$, so Lemma 5.6 gives us that $|P_H| = 0$, so P lies entirely in G_q and therefore P is isomorphic to P_G. It follows that R restricted to G_q is isomorphic to R_G, and since R is deadlock-free in G_q, R_G must be deadlock-free in G. The same holds for H. □

Note that the path-optimality is necessary in the previous proposition. An overlap-cycle in G can be broken in the embedded copies of G in $G+H$ if the path is allowed to deviate from the plane (i.e., to move in H as well), but the resulting routing could not be path-optimal. This result is included to give a way of determining that there are no path-optimal deadlock-free routings for some networks, because they are cartesian sums of other networks for which there are no such routings. An example is the 5×5 torus, which is a cartesian sum of 5-cycles.

Definition 5.14 *Given routings R^G for G and R^H for H, define the* ordered sum routing R_{os}, *a routing for $G+H$ as the routing which given $(p,q),(r,s) \in N_{G+H}$, contains all the paths of the form PQ, where P is a path in G_q isomorphic to a path given by R^G between p and r and Q is a path in H_r isomorphic to a path given by R^H between q and s.*

In other words R_{os} first takes a route in G and then a route in H.

Theorem 5.10 *R_{os} is deadlock-free in $G+H$ if and only if R^G and R^H are deadlock-free in G and H respectively.*

Proof: ($\Rightarrow$) The path between any two nodes in G given by R^G is equivalent to the path between the corresponding nodes in G_q given by R_{os}. Consequently if R_{os} is deadlock free then so must R^G be. The same holds for R^H.

($\Leftarrow$) Follows from Theorem 5.5. □

Theorems 5.8 and 5.10 show that given deadlock-free path-optimal routings for the components, the ordered sum routing provides a path-optimal deadlock-free routing for a cartesian sum network.

Note that for cases where there is no path-optimal deadlock-free routing for the components in a cartesian sum network, the ordered sum routing constructed from the best suboptimal component routings may not give the best routing or even the best composition routing. However Corollary 5.9 guarantees that there is no path-optimal deadlock-free routing for the sum. Perhaps the algorithm in Section 5.4.1 could be used in these cases.

Definition 5.15 *Given routings R^G for G and R^H for H, define the* ordered product routing R_{op}, *a combination routing of R^G and R^H for $G \cdot H$ such that if $P \in R_{op}$ then P is of the form DQ, where D consists of all the diagonal links, i.e., $D_{GH} = P_{GH} = D$ and either $|P_{GH}| = |P_G|$ or $|P_{GH}| = |P_H|$.*

In other words R_{op} routes in both G and H simultaneously until one or both of the coordinates of the origin and destination match and then in whichever coordinate is different.

Theorem 5.11 *R_{op} is path-optimal in $G \cdot H$ if and only if R_G and R_H are path-optimal in G and H respectively.*

Proof: ($\Leftarrow$) Let Q be a path between two nodes in $G \cdot H$ and let P be a path given by R_{op}. Then

$$|P| = |P_G| + |P_H| - |P_{GH}| \quad \text{and} \quad |Q| = |Q_G| + |Q_H| - |Q_{GH}|$$

Suppose, without loss of generality, that $|P_{GH}| = |P_H|$, then $|P| = |P_G|$ and since P_G is a shortest path in G, $|Q_G| \geq |P_G|$. Also $|Q_H| = |Q_H'| + |Q_{GH}|$, so $|Q| \geq |P|$. Therefore P is a shortest path and so R_{op} is path-optimal in $G \cdot H$.

($\Rightarrow$) Follows from Lemma 5.7. □

Theorem 5.12 *R_{op} is deadlock-free in $G \cdot H$ if and only if R^G and R^H are deadlock-free in G and H respectively.*

Proof: ($\Rightarrow$) Identical to Theorem 5.10.

($\Leftarrow$) Theorem 5.5 will be used for this part. The links in $G \cdot H$ are partitioned into $L_W = L_G \times L_H$, the diagonal links and $L_B = L_G \times N_H \cup L_H \times N_G$, the horizontal and vertical links. Similarly take R_W and R_B to be the subroutings of R_{op} on $(N_{G \cdot H}, L_W)$ and $(N_{G \cdot H}, L_B)$ respectively. Then since every link in L_W projects into a link of both G and H, any overlap cycle of R_W would have to correspond to one of both R^G and R^H, so deadlock-freedom of both R^G and R^H implies that of R_W.

Now R_B consists of paths which are entirely within an embedding of one of G or H in $G \cdot H$. Consequently deadlock-freedom of both R^G and R^H implies deadlock-freedom of R_B.

The result now follows from Theorem 5.5. □

The pairwise normal products and cartesian sums which have so far been considered are extended to general finite sets of graphs in [9] and the results presented here may be carried over by induction.

With these results combination routings for networks expressible as cartesian sums or normal products can be generated which preserve path-optimality and deadlock-freedom and for networks lying in between, deadlock-freedom can be preserved but not necessarily path-optimality.

5.5.2 Fixed Link Valency Networks

For real networks composed of components (such as transputers) with a small fixed number of links, product networks have the disadvantage that the number of links required per node goes up at best as the sum of the number of links in the components (in the case of cartesian sum networks) and at worst the product (in

the case of normal product networks). This section discusses a common attempt to solve this problem.

Definition 5.16 *The network J is a* G-connected H *if there exists a mapping* $C : L_G \mapsto N_H \times N_H$ *such that J is of the form* $(G \times H, L_H \times N_G \cup L_G)$, *having a link between nodes* (p, q) *and* (r, s) *if either* $p = r$ *and* $[q, s] \in L_H$ *or* $[p, r] \in L_G$ *and* $C([p, r]) = (q, s)$.

The most common example of this type of network is the Cube-Connected Cycle (CCC) [77] in which G is a k dimensional cube and H is a k-cycle, with C mapping links which follow dimension i to the node pair $(i, i) \in N_H \times N_H$.

Cartesian sum, normal product and intermediate networks are special cases of G-connected H networks, with suitable definitions of the C relation. However, when fewer links exist between the copies of H than for the cartesian sum, such as with the CCC, it becomes difficult to find combination routings which preserve deadlock-freedom and path-optimality. The missing links mean that attempting to follow the R^G part of the route requires extra hops in H in addition to those of the R^H part. Even the algorithm in Section 5.4.1 performs poorly on these networks compared with other constructed networks such as the hypercube.

[78] gives a way of constructing hypercubes from fixed degree nodes, similar to cube connected cycles except that there is a deadlock-free (though not path optimal) routing. [63] gives a shortest path routing for CCC and derives network diameter, but does not consider deadlock.

5.6 Application to Common Network Classes

5.6.1 Grids and Hypercubes

A $k \times k$ grid network is a cartesian sum of two k node linear chains, each of which contains no cycles, so its shortest-path routing is deadlock-free. Consequently the ordered sum routing of the grid, which first takes the shortest route within the source column to the destination row and then the shortest route across the columns to the destination, is path-optimal and deadlock-free. This method can be extended to a general n dimensional hypercube (which is a sum of n chains) and is known as e-cube routing [95]. It also has the property that the load on the links is exactly balanced.

5.6.2 Simple Cycles

A cycle of 3 or fewer nodes is trivial since it is completely connected and a cycle of 4 nodes is a grid. However simple cycles of 5 or more nodes are the most pathological of networks! In cycle networks, an overlap-cycle exists when each link is used by at least two spanning trees, so any deadlock-free routing must have a link which is used by at most one spanning tree in each direction. In Figure 5.5, the initial solution has this property, as neither links $[2,3]$ nor $[3,2]$ are used for any spanning tree. In the derived solution, link $[1,2]$ in the clockwise direction is used by the spanning tree for node 2 only and link $[4,3]$ in the anticlockwise direction is used by the spanning tree for node 3 only. Attempting to improve either of the sub-optimal spanning trees for nodes 2 and 3 violates this property. In general the routing based on a single rooted spanning tree can be improved by exactly one hop for each spanning tree, other than the root. This means that in the best case there are always 2 trees with worst case $n-2$, 2 with worst case $n-3$ and so on down to $\lceil \frac{n}{2} \rceil$.

5.6.3 The Torus

The torus is a cartesian sum of cycles. Consequently by Corollary 5.9 there is no path-optimal deadlock-free routing for greater than 4×4 nodes since there are none for cycles of more than 4 nodes.

5.7 Conclusions

During development, testing and debugging, it is important that parts of the system which are not part of the application are reliable. Otherwise when a bug shows up, the developer can never be sure whether it is the application code which is at fault or the system. For this reason the routing subsystem of Testbed should be free from deadlock and should be as predictable as possible in performance.

Due to the requirements of low overhead and determinism for embedded systems, Testbed does not use the deadlock avoidance schemes or adaptive routing methods which are common in the literature. This motivated the search for ways to construct deadlock-free routing functions for given interconnection topologies, which are as near to path-optimality and load-balancing as possible. The algorithm in Section 5.4.1 has been found to work well in tests. Section 5.5 showed that a wide range of networks may be constructed from simple ones while preserving path-optimality and deadlock-freedom. The combination of these algorithmic and constructive techniques provide powerful tools for solving the routing problem.

Chapter 6

Monitoring and Background Debugging

6.1 Introduction

Traditional interactive source level debuggers are powerful tools when used with sequential non-real-time programs. However they are often useless for systems which exhibit non-deterministic concurrency, where the behaviour of the system is affected by the relative speeds of the processes or real-time systems where the elapsed time taken by a process or group of processes to perform some task is critical to correct behaviour. The former type of behaviour is known as *temporal*, while the latter is called the *timing* behaviour of the system. Interference with either caused by debugging or monitoring activity is the probe effect. Interactive debugging involves stopping a process at *breakpoints* which usually correspond to statements in the original program source code. The user can then examine and sometimes update (poke) variables and then continue the execution. Unless the system is entirely simulated by a sequential process, the other processes and the global clock continue. If these processes expect to communicate with the stopped process then both the temporal and timing behaviour of the system may change. Incorrect behaviour which occurs when the debugger is not present may be masked by this effect or new errors may be introduced. A common approach to avoiding the probe effect is passive monitoring in which information is recorded at breakpoints without stopping the program. There is still some danger of interference however as this code takes some time to execute.

The actions and message passing in Testbed programs provide natural breakpoints at which to perform monitoring. Depending on the application, standard system monitoring facilities can often be applied, without the application's knowledge, to detect and report events and state values at points where this can safely be

done without perturbing the application's temporal or timing behaviour. Where these standard facilities cannot be used safely or do not provide sufficient flexibility, code can be inserted into the application to report information. Testbed's dynamic modification facilities make this a realistic approach.

One way to minimize the interaction bottleneck is by means of background debugging. Debugging decisions and actions are devolved as far as possible onto a user proxy, or *surrogate*, a process or embedded code, running as part of the debugging system with access to the full range of debugging facilities available to the interactive user. The surrogates can also have capabilities closed to the user, such as gathering statistics and profiling the client system behaviour.

Even in passive monitoring and background debugging systems, care must be taken to avoid introducing delays which may perturb sensitive parts of the application. The debugging system should provide a variety of monitoring and debugging mechanisms from which the skilled user can choose, based on knowledge of the application, to gather the required information without causing interference.

6.2 Related Work

6.2.1 Monitoring

The most common method for monitoring a process is to put hooks into system or kernel routines which either record information about the events requiring system intervention (such as message passing and process creation/destruction) or pass it on to an active monitor process [18].

Often monitors just record information for later post-mortem analysis [64] (DPM), [65], [32], [34] (TAP). The amount of information recorded varies depending on what the history is used for. Often the processes are "played back", in which case events (usually communications with other processes) are recorded.

In some cases monitored events along with snapshots of the state of processes are checked against specifications to verify if they are valid, while the process is active. This is the background debugging approach.

Breakpoints

Breakpoints can be used to monitor the current state of a process without passing control back to the user. They are usually implemented by a trap in a system call which passes control to the debugger. In some cases a branch to the breakpoint code is inserted dynamically into the executing code [2] (Parasight), [49]. This method has the advantages that the breakpoint code has access to the process'

global variables and does not require a context switch. In some systems the latter could be quite a significant time saving. With Transputers however, it is not so important.

Avoiding the Probe Effect

Approaches to reducing interference are as follows:

- Move as much as possible of the monitoring activities away from the target processor and on to the host [40] (CHILLScope), or on to special monitoring nodes [1] (Parasight).
- Merely record information and perform post-mortem analysis [34] (TAP), [64, 65, 32], [89] (SPIDER).
- Leave monitoring code in place even when not debugging (usually it is built into the kernel) [34] (TAP), [99] (ART). This avoids unmasking synchronization problems when the debugger is removed.
- Introduce delays in non-monitored events which may be affected by the delay caused by monitoring an event [5]. Unfortunately the specification of events and placement of delays could be as error prone as the actual program!

6.2.2 Background Debugging

High-Level Debugging

One approach to background debugging is *high-level debugging*[1] [5, 7, 8, 11, 75], which is the integration of debugging with formal specification. The common philosophy of this work is counter-example oriented, aimed at showing that a system fails to meet its specification. The counter-example search will generally be compiled from a specification of the application written in some fixed specification language. This approach is complementary to that of proving correctness: a failed proof indicates where to look for a counter-example, and failure to find a counter-example indicates a possible line of proof. The special problems of embedded systems make this approach too restrictive. The difficulty of formalizing a specification means that different languages are appropriate in different circumstances, and that sometimes no language is appropriate. In the latter case the user must have recourse to a more exploratory approach, and needs to be able to construct their own counter-example searches directly.

[1] The term is also used by some authors to mean simply source level debugging.

Event Recognition and Filtering

[7] describes a toolset based on a model called Event Based Behaviour Abstraction (EBBA) for clustering and filtering events defined using a special Event Definition Language (EDL). This is used to support interactive monitoring and debugging of multi-process systems by recognizing and presenting to the user only those events which are of interest. The clustering capability (describing events in terms of other events) allows quite complex interleaved behaviour patterns to be detected. EDL resembles path expressions in that it is based on regular expressions. Event recognition is compared to the problem of syntactic pattern recognition.

Behaviour Specification Languages

[5] describes a concurrent debugger for a CSP derived language called MuTEAM. The debugger checks the behaviour of a program against a specification which is defined in terms of the specifications of the component processes. These take the form of *Behaviour Specifications* which define a partial ordering of *Event Specifications* along with a set of assertions to be verified at given points. The only types of events monitored are communications and process creation and termination. The behaviour specifications closely resemble path expressions. The debugger consists of a set of communicating processes DP_i, one for each process P_i being debugged. Each DP_i maintains a Behaviour Specification BS_i for P_i as well as a subset of the state of the debugger. Each time P_i makes a system call to perform some concurrent activity, DP_i is notified. It then checks the behaviour specification to determine whether the activity is correct and determines whether the activity may proceed or should be delayed until some other event has been monitored. This is determined from communication with other debugging processes.

[11] describes an extension of path expressions called *path rules* which are aimed at detecting deviations from correct behaviour of processes at the language level. [44] extends this to *Data Path Expressions* (DPE) which have features for describing concurrent execution. The objects referred to in the DPE's are functions in the code or shared variables. The checking mechanism uses *Predecessor Automata* to represent the DPE's supplied by the user. This preserves causal independence, allowing potentially concurrent events to be detected in addition to invalid orderings.

Embedded Behaviour Specification

[25] describes a behaviour specification (assertion) language for occam programs. It includes specifiers for the temporal relationship of events such as ALWAYS,

NEVER, OVERLAPS, PRECEDES, etc. Labels for events and states (expressed as relationships among variables) are embedded in the code along with assertions about their temporal relationships. As with DPE, activities which have the potential to overlap (i.e., are causally independent) may always be detected independently of interprocess scheduling. This is because of the way events and states are represented as intervals, rather than discrete points in time.

The implementation involves associating any code segment enclosed by an assertion with a monitor process. The monitor process is informed through a special channel of the logical timestamps for the beginning and end of the interval to which it corresponds allowing temporal assertions to be checked.

6.3 Testing and Debugging

Testing and debugging are closely related activities. The difference is that during testing it is unknown whether the application contains bugs while debugging involves determining the cause of a bug once it has been discovered through testing. After a modification has been made which is intended to correct a bug, testing must be performed again to check whether the bug has indeed been fixed and to look for new bugs which may have been introduced by the change. Thus testing and debugging phases alternate until the testers have sufficient confidence that the application functions correctly. Both testing and debugging involve monitoring and searching for behaviour patterns and both are susceptible to the probe effect which may cause the testers to see bugs which are not really present or to miss bugs and may mask bugs which the debugger is looking for.

6.3.1 Capture

The test phase involves capturing patterns of behaviour which violate the application's specification. Subsequent to the detection of a violation, the debugging phase is entered. During debugging, behaviour patterns which are likely to be related to the original observation are also captured in an attempt to isolate the bug. If the bug is found to be in the specification then this will be refined or modified to include the negation of the erroneous behaviour. The capture of behaviour involves patterns of events and state.

Process State and State Transitions

There are two levels at which the behaviour of a client process can be modelled by state transitions, differing in granularity. The fine grain approach defines a

state by instantaneous values of program variables. These transient states are changed by internal computation within an action. The coarse grain approach focuses attention on those checkpoint states which occur when the current action begins, sends a message or terminates or when no action is currently in progress. A state transition in this model corresponds to a communication followed by some dynamic computation through a series of transients to find the next checkpoint. The checkpoint notion of state underlies languages like CCS [68] and LOTOS [10]. Most debugging activity concentrates on checkpoints, as Testbed is not a source level debugger, although it is possible to "spot check" transients by insertion of extra code in the application.

Patterns of Events

Sometimes it is sufficient to capture a single local event or state which should not occur. This is especially true during the early stages of testing. However during debugging it is often necessary to detect combinations of events or events and state which may be causally related to the failure previously detected. These patterns are often distributed across several slots which may be located on different processors. The order of events and states is likely to be significant as is the clock time, as embedded systems are often also real-time systems. For this reason each captured state and event should have a timestamp which is comparable across the network. A common approach to distributed event detection is to use logical clocks to give each event in the system a consistent timestamp. This idea goes back to Lamport [51], whose Clock Algorithm generated a total[2] extension of causality, and has since been generalized to partially-ordered vector timestamps whose order exactly characterizes causality. Unfortunately this approach tends to be costly to implement, scales poorly and is of no use when the specification involves real time. Instead Testbed relies on simple scalar timestamps taken from the local clock and uses the result in [51] that as long as physical clocks are synchronized with sufficient accuracy relative to the minimum message latency, then they will be consistent with causal ordering.

6.4 Monitoring

Monitoring in Testbed is divided into events (messages) and state (variables and ports). Each slot may have a number of monitoring clients, which are other slots in the system. In general these will either be the host server or surrogate slots. Each

[2]Lamport's timestamps are simply numbers.

client makes a new monitoring request or updates a previous request by sending a message to a system port. Each request may be related to a particular event type or simply a request for the value of a variable to be returned either once or periodically. When monitoring subevents other than message arrival, the values of a subset of the state may also be reported.

Along with the event or state data, each reported item has a real timestamp taken from the local processor's clock. The local clocks are synchronized using a simple protocol which achieves an accuracy which is an order of magnitude smaller than the minimum latency for sending a message between two nodes.

6.4.1 Event Monitoring

Events are divided into four subevent types for monitoring: message arrival, action invocation, action termination and message sending. A monitoring client (user or surrogate) may request monitoring of any combination of these for any subset of ports. A level of detail (0–4) may be requested. In addition for the invocation, termination and send subevents, a variable name may be given, in which case the value of this variable is logged along with the subevent and reported at the same time. The detail levels return information as follows:

0: No information is reported about the event, but if variables are to be monitored then these are logged and reported.

1: Only the port id, subevent and the local time are reported.

2: As with 1 but the destination (in the case of send) or the source (otherwise) is also reported.

3: As with 1 but the size is reported.

4: The report message contains a verbatim copy of the message.

Monitoring of different subevents is provided to allow performance of the system to be evaluated (e.g. delay between arrival and invocation or invocation and termination may be measured). It is also useful to be able to examine the state of a variable before and after an action is invoked or at the time a message is sent. The various levels of detail allow the overhead of monitoring to be controlled. For example if a message contains a large data field then logging or reporting a verbatim copy will be costly and may interfere unacceptably with the behaviour and/or performance of the application.

Event monitoring requests also contain return ports to which the reported events and state values are addressed on the client. In this way both human monitoring and background debugging are supported.

The arrival subevent is special in that it must be performed by the kernel rather than the system layer. In order to keep the kernel's involvement in monitoring to a minimum, variable logging is not supported for message arrival. The points at which snapshots of a variable are of interest are before during and after a particular action and these are covered by the message invocation, message send and termination subevents along with the capability of inserting monitoring code at arbitrary points in the action body. No extra benefit would be gained by providing variable logging on arrival. However monitoring the time of arrival is useful as a way of measuring the delays experienced by messages between arrival at the node and the commencement of processing.

6.4.2 State Monitoring

In addition to the state values which may be logged when a subevent occurs, a one-off peek or periodic reporting of variable values is provided. As with event monitoring the request contains a return port to which the captured value is addressed at the client slot.

6.4.3 Implementation

Event Logging and Reporting

To reduce delay in the response to events and the processing of actions, reporting takes place after the action terminates. The report messages are sent by the action harness as if the action itself had sent them. This means that the monitoring is above the kernel[3] and takes place in the system layer. Since user actions may not modify the message header, the arrival, invocation and termination subevents do not require that a copy of the message be made. The system records the time at which each of these subevents occurs in a field of the schedule entry encapsulating the message which is being processed and logs the value of any variable which is to be monitored. However a copy of the message is made for each monitored send subevent. The logged messages and values are kept in a list attached to the schedule entry. A separate copy is made for each client who requested it. It is not likely that there will be many clients requesting the same information. When the action completes, a report is sent to each client who requested each subevent

[3] apart from recording the time of arrival

and the log of messages sent and captured state values is traversed and its entries reported to the various clients.

Monitoring requests are stored as entries in the port table as monitoring is performed on particular ports. Each port descriptor contains an entry for each subevent type which is a list of clients requesting that subevent with the level of detail and return port for each.

A message can be sent cancelling monitoring of a particular subevent for a particular client.

Peeking

Peeking is performed by a system action on the target slot. This action sends a message to a return port, specified in the body of the peek request message, on the slot which sent it, containing the contents of a variable and its type. When a peek is requested by the user, the port contained in the peek request message is H_DISPLAY_VAR. The host action h_display_var calls the recursive function display_var which displays any scalar, pointer, array or structure type, including all elements or fields of the latter. Scalars types contain a printf/scanf format string, which is used for printing or scanning in poke. Each peeked variable is displayed by the host server in a separate top level widget which contains fields and buttons for performing more peeks and also pokes. The reported variable is accompanied by a timestamp.

The peek requests sent to slots contain a period. If the period is 0, then the peek is performed once. Otherwise the peek action reschedules the message for a time in the future given by the period.

6.4.4 User Interface to Monitoring

The interactive user of Testbed can request event monitoring using the "Event Monitoring..." item in the "Slot Commands" menu at the top of the window corresponding to the slot on which the monitoring is to be performed. This brings up the dialog box shown in Figure 6.1. The "Don't Monitor" button causes a cancel message to be sent. If a variable is entered in the "Monitored Variable" field then a dialog box containing the current value of that variable is automatically displayed. In the interface to the ROV shown in Figure 3.3 the `motor_settings` variable is being displayed in this way upon termination of the `Update_Motors` action in the LOW_CONTROL slot. Subevent reports are displayed in the scrolling text subwindow at the bottom of each slot window as shown in Figure 6.2.

Variable peeking may be performed using the Variables menu for a particular

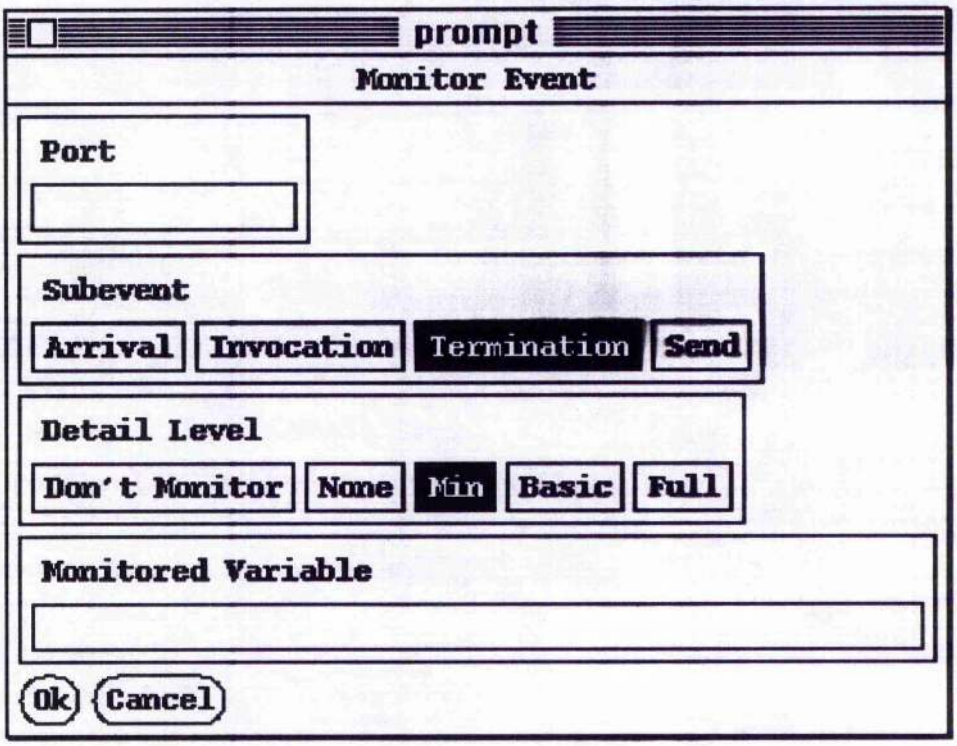


Figure 6.1: Dialog box for the "Event Monitoring" command.

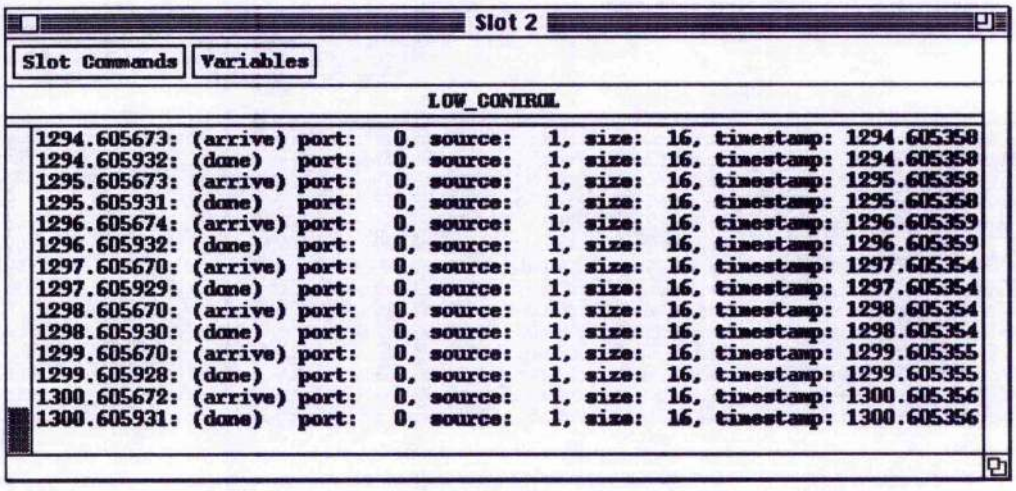


Figure 6.2: Scrolling event log showing arrival and termination subevents.

slot. This menu contains a list of all static variables created on that slot by the application as well as the "Peek..." and "Port..." entries which allow peeking of certain system variables which are also available to applications and entries in the port table. When the menu entry for a particular variable is chosen, a dialog box showing that variable is displayed.

Each monitored or peeked variable is reported to the host server together with its type and this slot contains code for displaying variables using information from the type table.

Inserted Code

Testbed does not support source level debugging, so to investigate the internal behaviour of an action the user must either:

1. use the mechanisms described above to monitor the messages it sends and take snapshots of the state before and after the action or when a message is sent, or

2. place code in the application which monitors the behaviour and reports detailed state-related events and state values at certain key points.

Method 2 is much more convenient with Testbed than with other programming systems due to the dynamic replacement facilities it provides. Parts of the application can be reloaded on the fly with debugging code inserted without the time consuming and tedious requirement of stopping and restarting the application. Library functions are provided for displaying text and variable values. Alternatively the user can add new actions to the host server to display or log information reported by the application or create new slots which collect, process and respond to patterns of events. These are the surrogates which perform background debugging as described in Section 6.6.

6.5 Avoiding Interference

Testbed's monitoring facilities are not guaranteed not to interfere with the timing of an application. Just as the application designer must choose scheduling parameters such as the criticality of certain actions and priority or deadline for particular slots and the mix of slots on each node in order to meet the timing constraints, the person using the monitoring facilities must take account of the characteristics of the application and choose the type of monitoring to perform in order to avoid perturbing the behaviour under investigation. Fortunately there is usually a wide

range of possible ways of monitoring a particular aspect of application behaviour, some of which cause less interference than others. For example, if the response time of a slot to a particular type of message is critical, then extra processing and message traffic due to monitoring of other aspects of the application may cause unacceptable delays and capturing large amounts of state data upon message invocation will certainly delay the response. However it is likely to be safe to monitor the message arrival and action invocation without logging any state variables as this introduces no extra overhead before the message is processed, though extra messages are transmitted afterwards. If the action performs some work then sends a message and the time between the event occurance and sending the response is critical then it is safe to monitor the message sent, as well as logging any important variables since this is done afterwards and will not interfere. Similarly monitoring on action termination will not interfere with the response. There is likely to be some slack time between events during which the monitoring may be performed without interfering with the application. In cases where even sending messages after processing an event is not safe, then code inserted in the application must be used to record the required information until it can safely be reported.

6.5.1 Example

Figure 6.3 shows a snapshot of the LOW_CONTROL window for the ROV implementation described in Section 2.10. The arrival and termination subevents for port 0, which is the `update_motors` port are initially monitored and the timestamps show a latency of approximately 0.00026s[4] in processing the update. However after requesting that the variable `motor_settings` be monitored on invocation of the action, this latency goes up to around 0.00053s[5]. If the variable had been monitored on termination then the response latency would have been unaffected.

In fact monitoring performed on the node containing the LOW_CONTROL slot is liable to delay the `Switch_Off` actions, causing some motors to be driven slightly too long. This can sometimes be avoided by performing the monitoring on one of the other nodes. For example the HIGH_CONTROL slot also has a copy of the `motor_settings`. Sometimes Testbed's standard monitoring and display facilities are insufficient and must be supplemented by extra application specific code. For example the actual proportion of the duty cycle for which the motors are active can be found by monitoring the invocation of the `Rov_Message` action in

[4]difference between the arrive and done times before the arrow in Figure 6.3

[5]difference between arrive and done times after the arrow in Figure 6.3

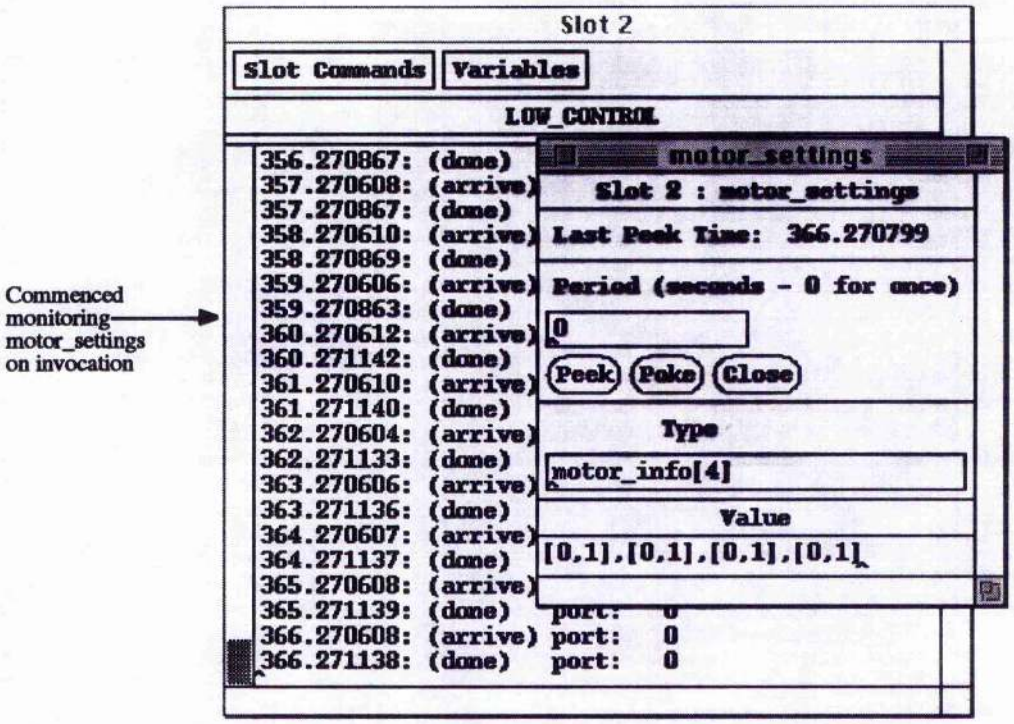


Figure 6.3: Interference caused by variable logging on action invocation.

the ROV slot. However since the same action is responsible for turning the motors on and off, changing the direction and requesting the current depth and compass values, the user sees many spurious events and it is difficult to pick out which ones are responsible for turning particular motors on and off. An alternative would be to insert code into the body of the `Rov_Message` action to report the length of time each motor spends in the on state. Combining this with code inserted into the HIGH_CONTROL slot to report the value of the `motor_settings` allows the performance of the LOW_CONTROL slot to be checked without interfering with it.

A Design Flaw Uncovered

The experiment described above was performed and highlighted a subtle shortcoming of the original design of the `Switch_On` action in the LOW_CONTROL slot described in Section 2.10.3. As shown in Figure 6.4, the length of time for which each motor is switched on in the simulated ROV is consistently short of the value which should be produced by the `motor_settings` value by approximately 0.0015 to 0.002 seconds. To see this note that the `duty_cycle` for the LOW_CONTROL slot, shown in the box in the bottom right of the figure, is 1 second and that the `motor_settings` variable is an array of four (speed,direction) pairs. Each speed value is a percentage of the duty cycle, so for example, a speed

of 18 for motor 0 shown in the window for the HIGH_CONTROL slot should correspond to a time of 0.18 seconds appearing in the window for the ROV slot. The problem is due to the fact that the SWITCH_ON messages are scheduled before the MOTORS_ON message is sent to the ROV, with delay relative to the time at which they are sent.

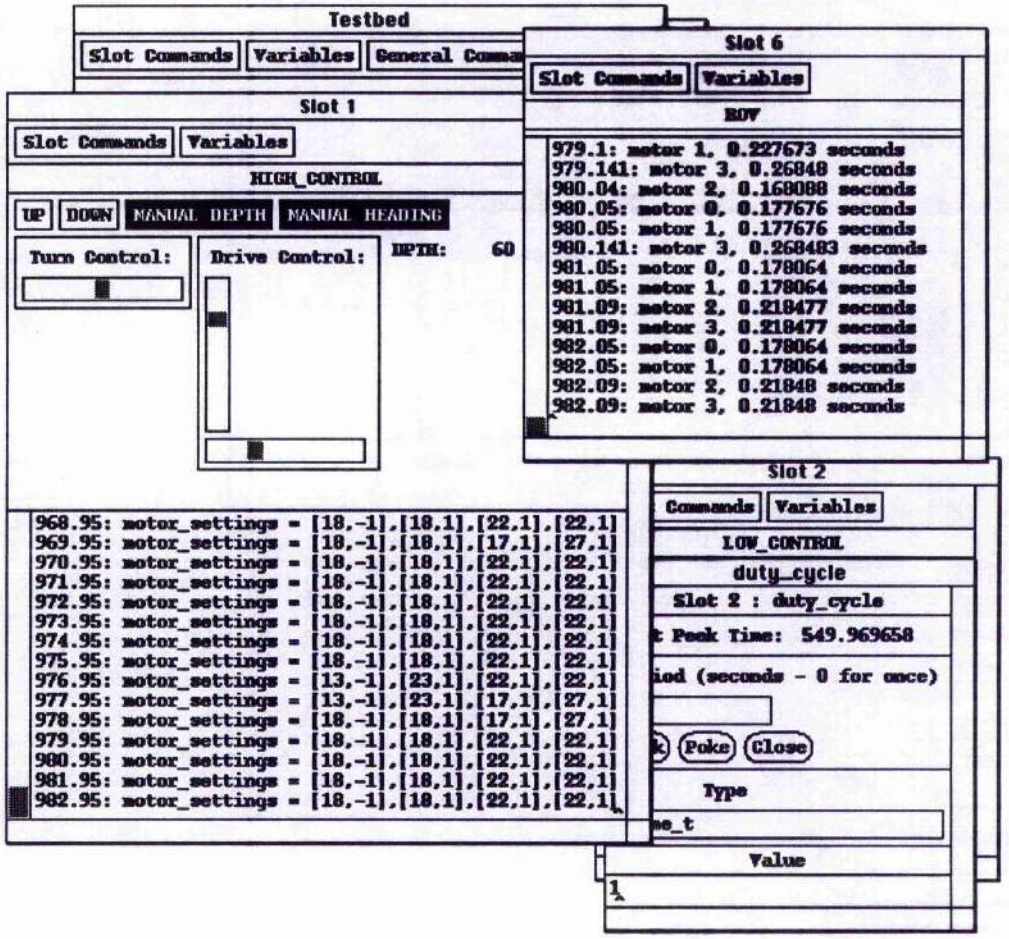


Figure 6.4: A subtle design flaw uncovered by monitoring.

A modified version of the action follows. This differs from the original version in Section 2.10.3 in that the messages setting the motor directions and switching them on are sent before the SWITCH_OFF messages are scheduled and that these are scheduled relative to a fixed time just after the motors are switched on rather than each one being delayed relative to the current time.

```
Switch_On:

while next_duty_cycle < Now do
    next_duty_cycle := next_duty_cycle + duty_cycle;
motors_on.set := motors_plus.set
```

```
    := motors_minus.set := EMPTY_SET;
for i := 0 to nmotors-1 do
  begin
    if motor_settings[i].speed > 0 then
        add(i,motors_on);
    if motor_settings[i].direction = 1 then
        add(i,motors_plus)
    else
        add(i,motors_minus)
  end;
Send(MOTORS_PLUS);
Send(MOTORS_MINUS);
Send(MOTORS_ON);
on_time := Now;
for i := 0 to nmotors-1 do
    if different_speed(i) then
      begin
        speed := motor_settings[i].speed;
        motors_off.set := EMPTY_SET;
        add_motors_with(speed,motors_off);
        if speed < 100 then
          begin
            Schedule(SWITCH_OFF,
                     on_time + duty_cycle * speed/100
                     + fudge_factor);
            on_count := on_count+1
          end
      end;
if on_count = 0 then
    Schedule(SWITCH_ON,next_duty_cycle).
```

After modifying the action in this way, the results shown in Figure 6.5 are produced. Note that there is still a small discrepancy of approximately 0.00016 seconds in each reported time. This is due to overhead at the ROV slot in processing the messages which are sent ahead of the `MOTORS_ON` message. Since this is almost constant, it can be measured and compensated for by the `fudge_factor` added to the scheduled time for the `SWITCH_OFF` message. However, since the delay for processing messages in the simulated ROV will be different to that in the real one, this tuning may need to be done at a later stage in development.

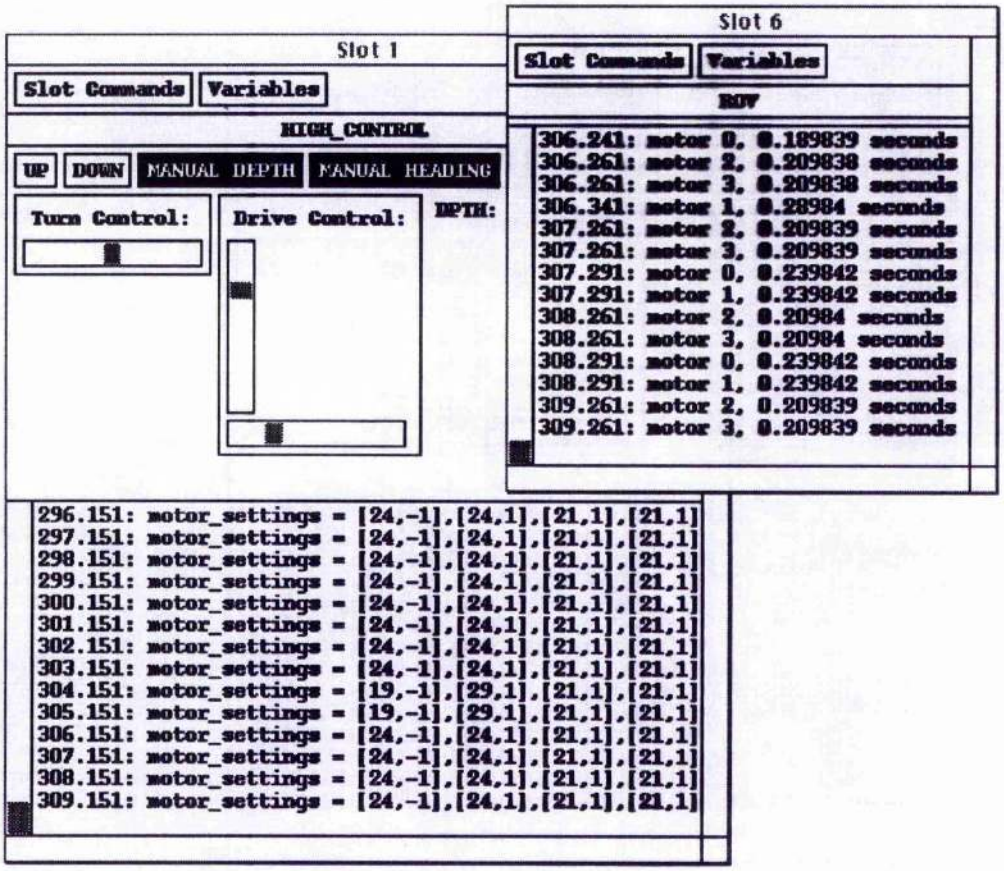


Figure 6.5: Results with corrected `Switch_On` action.

The monitoring described in the above example can be taken a step further by creating a new slot which monitors both pieces of information and checks for excessive discrepancies between the desired and actual motor speeds. This is an example of *background debugging* as described in Section 6.6.

6.6 Background Debugging

In the monitoring described in the previous sections events are repeatedly reported and an interactive user is required to observe anomalies which may be single abnormal events in a stream or compound events requiring comparison of events occurring in different parts of the application. This approach can be effective as a first pass when the developer is not searching for specific faults but watching for any potential problem. It often provides enough information to isolate the fault as illustrated by the example above, where an unexpected anomaly was noticed and led to the detection of a defect in the code. However there are a number of drawbacks to simple interactive monitoring:

- The rate of raw event occurance may be too great for a human observer to cope with or even for the Testbed user interface to display.
- Complex combination events are difficult to identify by watching several streams of event reports.
- The event report messages may cause interference with the application, resulting in introduction or masking of erroneous behaviour, i.e., the probe effect.

These problems can often be avoided by introducing automatic observers or surrogates which collect event reports and watch for certain event patterns. This constitutes background debugging. Surrogates may be:

- special slots added to the application, often on extra processors or
- actions added to application slots or
- code embedded in the application at key points to watch for certain local event patterns or states.

Just as with applications, surrogates (of all forms) can be created and modified on the fly.

6.6.1 Where to Place the Surrogate

Sometimes the behaviour pattern to be captured is entirely local to a slot, such as the arrival of a certain type of message when one of the slot's variables has a particular value or falls within or outside some range. In this case surrogate code can take any of the three forms listed above. It is generally desirable to minimize the effect of the surrogate on the application while it is behaving normally, though when the condition of interest has been detected, it may be acceptable to interfere. If the checks which the surrogate has to make are smaller than the effect of sending messages, then embedded code is preferable from this point of view. Actions added to the application slot have the advantage that the application code does not need to be modified. The overhead in this case depends on whether the values of state variables need to be captured before the application action has terminated. If not then since the surrogate has direct access to the slot's variables the overhead consists of that required to generate the message scheduling the surrogate action which takes place after the application action has terminated.

For behaviour patterns which involve more than one slot, a surrogate slot is often called for. This may cooperate with other surrogates either embedded within

the application code of the slots involved in the behaviour pattern or surrogate actions on those slots.

6.6.2 Example

The monitoring example of Section 6.5.1 can be converted into one of background debugging. Instead of the ROV and HIGH_CONTROL slots reporting the durations of motor drive and motor settings respectively to be displayed on the host slot, they can be sent to a surrogate on another node which checks whether the durations are within an acceptable tolerance of the correct proportion of the duty cycle corresponding to the motor settings. Subtleties in the correct design of this surrogate include determining which pairs of `motor_settings` and durations correspond. The surrogate can receive a new `motor_settings` value before a duration which corresponds to the previous `motor_settings`. In fact the new `motor_settings` do not take effect until the next duty cycle of the LOW_CONTROL slot. This latency effect can be seen by careful examination of Figures 6.4 and 6.5. A human observer can easily see from the pattern of settings values and durations which ones correspond. However to simplify the surrogate it is better that the LOW_CONTROL slot's value of `motor_settings` be monitored by the surrogate on termination of the `Switch_On` action instead of being reported by the HIGH_CONTROL slot. The implementation of the surrogate slot is presented below.

Actions and Ports

Two actions `Settings` and `Check_Duration` are needed for this slot. Ports are defined as follows:

```
settings        [action = Settings;
                         data = new_motor_settings]
check_duration [action = Check_Duration;
                         data = duration]
```

Outgoing Messages

The surrogate sends an initial monitoring request to the LOW_CONTROL slot during its initialization. The ROV slot has code embedded within it for sending the durations to the surrogate which is activated by poking a variable. When an error is detected a report is sent to the HOST slot.

```
REQUEST_SETTINGS   [dest = LOW_CONTROL;
```

```
                          port = monitor_request;
                          data = request]
REQUEST_DURATIONS [dest = ROV; port = poke;
                          data = report_durations_on]
ERROR_REPORT      [dest = HOST; port = h_prints;
                          data = error_report]
```

Actions

```
Settings(new_motor_settings):

motor_settings := new_motor_settings;
motor_settings_received := TRUE.

Check_Duration(duration):

if motor_settings_received then
    if |motor_settings[duration.motor] - duration.value|
       > tolerance then
        Send(ERROR_REPORT).

Initialisation:

motor_settings_received := FALSE;
Send(REQUEST_SETTINGS);
Send(REQUEST_DURATIONS).
```

6.7 Conclusions

The most important goal in monitoring and debugging embedded systems is elimination of the probe effect. Due to the varied nature of embedded applications and the desire to use standard hardware, Testbed does not guarantee to avoid this problem automatically. Rather a range of monitoring options are supported and it is up to the user to choose those which are least likely to interfere with a particular application.

Testbed's event-action programming model provides natural breakpoints for monitoring and debugging. Monitoring code in the system layer can be activated by sending system request messages to a slot. This monitoring code creates report messages containing variable amounts of detail on one of four subevent

types: message arrival, action invocation, message sending and action termination. Values of state variables can be captured for each of the last three subevent types. In addition variables may be peeked in between application actions by sending system messages to the slot. Both these types of monitoring support background debugging by recording the client slot which sent the request and a return port to which the report is sent. If required, code can be inserted into the application to assist in the monitoring and debugging process by providing a finer grained view. This is a more reasonable approach with Testbed than some systems due to its dynamic modification capabilities; inserting debugging code does not require stopping and restarting the application.

This chapter has described the technique of background debugging and how it can be performed in Testbed. Unlike some systems where background debugging code is special and must be hooked into the system, Testbed allows surrogates to be created in exactly the same way as applications from slots, actions and embedded code.

Chapter 7

Dynamic Modification

7.1 Introduction

Long lived software such as databases or embedded systems occasionally needs to be changed to fix bugs and to add or modify functionality. It is undesirable and sometimes unacceptable to interrupt the service provided by the software in order to perform these updates. A shutdown may be unacceptable for safety reasons, as in a spacecraft control system, for economic reasons, as in bank-transaction processing systems, or a combination of these as in telecommunications switching systems. This problem has motivated much research on dynamic modification systems, a good survey of which can be found in [85]. Even during development of this type of software it is often desirable to be able to modify parts of the system on the fly due to the high startup cost.

One method of dynamically modifying an application provided by Testbed as well as more conventional debuggers is to poke the values of key variables. This can be a very useful way of interactively tuning performance. For example in the corrected version of the `Switch_On` action described in Section 6.5.1, the variable `fudge_factor` adds a small correction to make up for extra overhead in the ROV slot delaying the start of the motor drive period. The value of the required correction can be found by monitoring[1] and then a poke command can be issued to update the value of `fudge_factor`. The result can then be verified visually.

The simple type of dynamic modification just described requires a certain degree of prescience on the part of the programmer in recognizing those parts of the code which may require parameterization. It is often difficult or impossible to predict what variables may be required in order to dynamically tune the application in this way. As with `fudge_factor` in the example above, the need for the

[1] Figure 6.5

variable may only be apparent after the system is tested. Of course it is often necessary to perform more complex transformations than mere poking allows or to modify the actual code and/or type definitions of the data. In fact in the example described, the `Switch_On` action required significant modification. Testbed provides the capability of making such alterations without stopping and restarting the application.

Dynamic modification is supported by the on-line replacement of code modules. Since data types and state variables are defined dynamically by declaration functions within these modules, this allows both the application code and the state structure to be updated. It is natural for data restructuring to be associated with code updating as it is always necessary to replace (at least recompile) some code when data types are changed. In cases where only a single module is affected on one slot, a one off replacement may be performed by loading the module (re-invoking initialization functions if necessary). This is also the method used to add new modules (such as debugging modules) to a slot. The new module is immediately linked and if it replaced an existing module, all modules are relinked to correct references to the functions it contains. User pointer variables and those in the port table are updated. State variables whose types have not changed are unaffected, while changed variables are reallocated and data is automatically converted. Initialisation functions are not automatically reinvoked as the default behaviour is to preserve as much of the current state as possible across a reload. Once the automatic conversion has been completed, a user supplied conversion function may be invoked to perform any extra reinitialization not coped with by the automatic conversion mechanism. It is quite possible that an update will introduce inconsistencies into the system which would result in a failure at some time afterwards. This may be due to partial failure of the load, for example some modules may fail to compile correctly or there may be a failure when a module is being linked in the slot. It is also possible that type changes result in inconsistent pointer assignments as will be discussed in detail below. The replacement system should support failure atomicity, i.e., the system should detect such failures and be able to reverse all of the changes. This is achieved by saving the definitions before reloading. If the reload fails for any reason, all the old versions of modules, variables and types are restored and the user is alerted.

In general more than one module will need to be replaced on more than one slot in an atomic operation. This is achieved using a protocol which ensures that data in messages are also correctly converted.

The challenge in dynamic modification is to perform the replacement in a way that transforms the old state of the system into one which is consistent with the new

version of the code while preserving as much as possible of the state information. This problem can be divided into two parts: when to perform the modification and how to do the conversion. The first subproblem is what most previous work has focussed on. The approaches include:

1. finding a safe point at which to perform the update [58] and
2. performing the update gradually, possibly resulting in several versions of the code active with special conversion functions to interface between versions [42].

Testbed follows approach 1. The Testbed programming model provides natural breakpoints between actions when it is safe to perform an update. Unlike more conventional programming systems Testbed does not have the problem of active code and stack variables to deal with. Testbed can replace all of the code, update the data declarations and perform all necessary conversions atomically before the next application action becomes active. The only possibility for version conflict arises from the interaction between slots. If no precautions were taken then either messages could arrive in an out of date format after a slot has been updated or messages could be sent and arrive in a new format at a slot which has not yet been updated. To avoid these possibilities a protocol is used for ensuring that all slots which are being updated do so atomically[2].

The conversion subproblem includes matching variables and procedures between versions and performing transformations such as mapping of procedure parameters and fields of structured variables. It will sometimes be difficult to find a sensible mapping and this may indicate an error in the new version which the system should detect.

The bulk of this chapter focuses on the automatic *on-line* conversion of data whose type definitions have changed, and specifically on the difficult problem of relocating pointers to parts of data objects, while checking consistency of the reassignment. This problem has been side-stepped by dynamic modification systems in the literature, either by disallowing such pointers or by replacing each object and its associated data as a unit, with conversion of old to new internal representation of the data performed entirely by user supplied code. This work is novel in that it allows the programmer to supply new versions of code and data declarations and automate the type conversions. This includes the relocation of pointers to arbitrary subcomponents of the variables which make up the process state. Where needed this automatic conversion may be augmented by user supplied code.

[2]See Section 7.9.

In order to implement the techniques which will be described, the system must know the type of each allocated memory object. Given this information all of the pointers in the system can be found. The type and symbol tables used for monitoring purposes also allow automatic data conversion and consistency checking using the algorithm and techniques described in this chapter. The target language of Testbed, C, permits great flexibility in pointer assignment. Consequently finding a new assignment of pointers is non-trivial and many forms of inconsistency may be introduced by an update.

The presence of pointers to variables or fields of variables creates alternative identities for the same storage which may have different types. These alternative identities are referred to as *aliases*. This is consistent with the use of the term in programming language literature, where it denotes different expressions with the same lvalue. Testbed's aliases may be created dynamically at run-time, may identify with an entire variable or just a field of a structure and may have a different type than that variable or field[3]. It should also be realized that there is a subtle distinction between a pointer and an alias. In Figure 7.1 the pointer `ap`, when made to point to the `a` field of the variable `block`, gives rise to an alias `*ap` to that field. The technique described is independent of the programming language used to create the aliases.

7.2 The Data Conversion Problem

There are two ways in which data definitions can change. The individual variables may be given different types, for example a variable declared as `int x` might change to `float x`. Alternatively the declaration may be unchanged, but the definitions of the types themselves may change, for example `int` might change from two bytes to four. Though the data conversion problem is qualitatively the same in both cases the latter is easier to deal with in that one only needs to know how to convert between the versions of those types which have changed. In this case each type needs a single conversion function associated with it (which may be an identity). If on the other hand, variable declarations have also been modified, extra conversion functions are required. Initially only the case of modified type definitions is considered, to simplify the notation. However Section 7.6 indicates how the results can be generalized to cope with variable redeclarations.

Figure 7.1 illustrates the creation of a set of aliases from pointers as might happen in a C program. The simple structure types `info` and `overlay` are defined, with fields of type `scalar`, which for simplicity is assumed to have

[3] with certain consistency restrictions, to be discussed later

unit size. A variable `block` of type `info` is declared and a pointer variable of type `overlay*` (pointer to overlay) is declared and initialized to point to `block`. Likewise a pointer variable `ap` of type `scalar*` points to the first field of `block`. The old and new versions of the object corresponding to the variable `block` are shown at the bottom. The aliases are shown above aligned vertically with the parts of the object to which they correspond. The memory locations in the object are numbered to show how they are mapped by the type conversion. This chapter is concerned with situations where it is desirable to replace the definitions of data in a running program, converting the data in an appropriate way and relocating any pointer variables. The diagram shows how the data in the variable block is mapped under the assumption that fields with matching names are identified and how the aliases which arise from pointers are relocated (allowing the pointers to be updated).

Old Type (t)	Variables	New Type (t')

`typedef struct`
`{`
`  scalar a,b;`
`} info;`
`typedef struct`
`{`
`  scalar x,y;`
`} overlay;`
	`info block;`
	`overlay *overp = &block;`
	`scalar *ap = &block.a;`
		`typedef struct`
		`{`
		`  scalar b,c,a;`
		`} info;`
		`typedef struct`
		`{`
		`  scalar y,z,x;`
		`} overlay;`

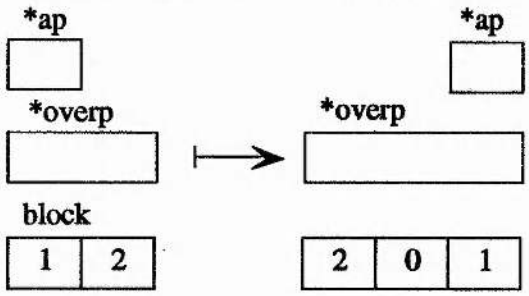


Figure 7.1: Relocation of aliases during a type change.

Section 7.3.2 presents a simple algorithm for relocating pointers during type conversions such as the one in Figure 7.1, in which fields of structures are rearranged or inserted. Unfortunately it is not always possible to compute a relocation of the aliases in such a way that the components of these continue to refer to the same data in memory. Figure 7.2 illustrates such a situation. This is called an *inconsistency* and it is likely to result in failure of the program when the alias `*overp` is used. As in Figure 7.1 the old and new versions of the object are shown at the bottom line of each example with the aliases above, vertically aligned with the locations in the objects to which they correspond. The numbers in the memory locations of the object and the aliases indicate how the *word mapping* of the type

of each would map those locations. Only the allocated object's word mapping corresponds to the data conversion. Where a field has been inserted the location is marked with a 0.

Old Type (t)	Variables	New Type (t')

`typedef struct`
`{`
`  scalar a,b;`
`} info;`
`typedef struct`
`{`
`  scalar x,y;`
`} overlay;`
	`info block;`
	`overlay *overp = &block;`
		`typedef struct`
		`{`
		`  scalar b,c,a;`
		`} info;`
		`typedef struct`
		`{`
		`  scalar z,y,x;`
		`} overlay;`

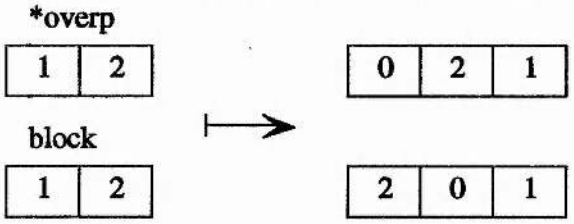


Figure 7.2: A conversion after which fields of aliases are identified differently.

Even when it is possible to realign aliases so that all memory locations which are identified in the original version are still matched in the new one, there may still be problems introduced by the conversion. Figure 7.3 shows a case where an extra field has been introduced into the `overlay` type, so that trying to preserve consistency results in the alias `*overp` being located before the start of `block`. This results in the alias `*overp` containing unallocated storage, which is also likely to result in failure when it is accessed. Figure 7.4 shows a similar situation where the end of `*overp` is unallocated. Although Figure 7.4 appears to be a trivial variation on Figure 7.3, it is included as it may sometimes be permissible for an alias type to extend beyond the reach of allocated storage as long as the extra fields are never accessed, but it is much more likely to be an error if the address of an alias (value of a pointer) does not lie in allocated memory. Figure 7.5 shows a more subtle case involving an alias which matches a part of another. The alias' type (`overlay`) has an extra field introduced where the type of the variable (`block`) does not. In this case no unallocated storage is reachable but `*overp` no longer exactly matches any one field in `block`. This too is likely to result in failure when the `z` field of `*overp` is accessed. Finally Figure 7.6 shows an even more subtle problem. Here `*overp` continues to match the whole of `block` exactly, but after the conversion the `y` field of `*overp` does not exactly match any field of `block`. Again this is unlikely to have been the intended result.

Old Type (t)	Variables	New Type (t')

typedef struct
{
  scalar a,b;
} info;
typedef struct
{
  scalar x,y;
} overlay;
	info block;
	overlay *overp = █
		typedef struct
		{
		  scalar b,c,a;
		} info;
		typedef struct
		{
		  scalar z,y,w,x;
		} overlay;

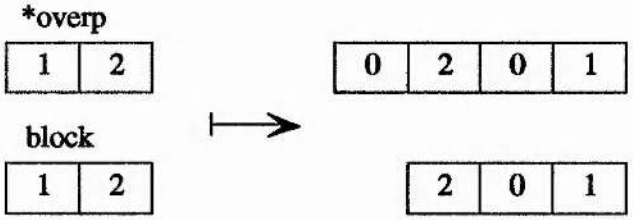


Figure 7.3: The new location of an alias is in unallocated storage.

Old Type (t)	Variables	New Type (t')

typedef struct
{
  scalar a,b;
} info;
typedef struct
{
  scalar x,y;
} overlay;
	info block;
	overlay *overp = █
		typedef struct
		{
		  scalar b,c,a;
		} info;
		typedef struct
		{
		  scalar y,w,x,z;
		} overlay;

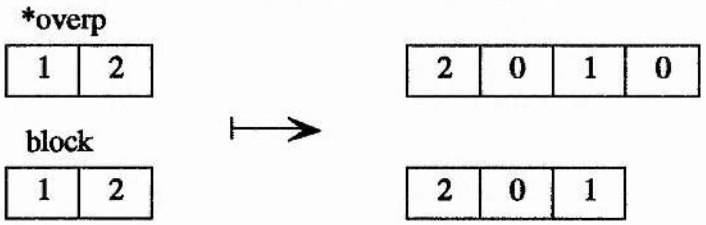


Figure 7.4: An alias extends outside allocated storage after conversion.

7.3 Relocating Aliases

The problem of converting state data is divided into two parts. First the data must be converted between the two versions according to a mapping between the data types. Then any pointers in the state must be updated. The pointer updating is achieved by relocating the aliases to which they give rise based on the relationship between the two versions of the data. In this section the second part of the conversion problem is formalized and an algorithm is presented which performs alias relocation under a type change. That is, this section will produce an algorithm which given a mapping A from aliases to addresses, attempts to construct a relocation function A' giving the new location of each alias.

The examples in Section 7.2 used C-style structure types. However in order

Old Type (t)	Variables	New Type (t')

typedef struct
{
  scalar a,b;
} info;
typedef struct
{
  info i,j;
} doubleinfo;
typedef struct
{
  scalar x,y;
} overlay;
	doubleinfo block;
	overlay *overp = &block.i;
		typedef struct
		{
		  scalar a,b;
		} info;
		typedef struct
		{
		  info i,j;
		} doubleinfo;
		typedef struct
		{
		  scalar x,y,z;
		} overlay;

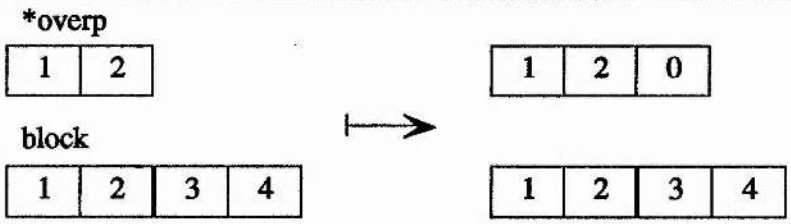


Figure 7.5: After conversion the alias does not exactly match any field of an allocated object.

to decide whether it is possible to find a relocation of aliases and if so, compute it only the word mapping introduced in that section is needed. Section 7.4 shows how information about the structure of types can be used to construct this word mapping. A consistency condition will be derived which corresponds to that which is violated in Figure 7.2. This condition helps us derive the algorithm for which it forms a correctness specification.

7.3.1 Assumptions and Definitions

Assume that the data resides in a collection $\mathcal{O}$ of memory *objects*. Each object $o \in \mathcal{O}$ will be treated as residing in a private memory space, represented as a linear array of *words* indexed (addressed) by the integers. Of course, in practice, all the objects lie in a single physical memory space. Nevertheless, they are logically independent, having been created separately either as variables or by dynamic memory allocation.

In addition, the data within an object can also be accessed via one or more *aliases*, as explained in Section 7.2. Thus any word of an object may have multiple means of access, and the conversion procedure must ensure that these semantic identities are preserved.

Each alias has a *type* t drawn from a set $\mathcal{T}$, with associated *size* $|t|$. This is the number of words of the *host* object within the range of any alias of type t, these will be numbered 0 through $|t| - 1$. Since each object will have been created with

Old Type (t)	Variables	New Type (t')

typedef struct
{
  scalar x,y;
} sub;
typedef struct
{
  scalar a;
  sub b;
} info;
typedef struct
{
  scalar a,b;
} osub;
typedef struct
{
  scalar x;
  osub y;
} overlay;
	info block;
	overlay *overp = █
		typedef struct
		{
		  scalar x,y;
		} sub;
		typedef struct
		{
		  scalar a;
		  sub b;
		  scalar c;
		} info;
		typedef struct
		{
		  scalar a,b,c;
		} osub;
		typedef struct
		{
		  scalar x;
		  osub y;
		} overlay;

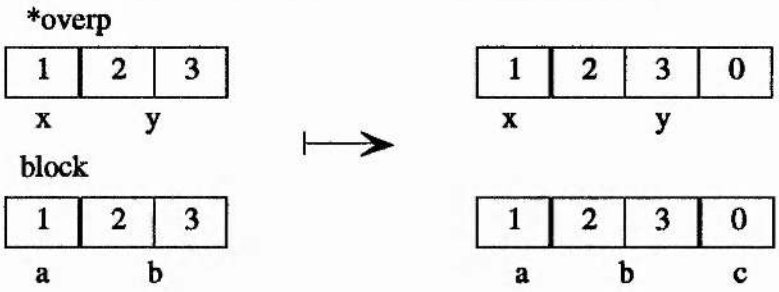


Figure 7.6: A field of an alias fails to match any field of an allocated object after conversion.

some type, it is itself an alias. So $\mathcal{O}$ is taken to be a subset of the aliases.

Type conversions are specified in terms of two functions: a function from $\mathcal{T}$ to $\mathcal{T}$ called a *retype*, and a word mapping which is a partial function W over $\mathcal{T}^2$. The retype corresponds to the redefinition of types which accompanies a dynamic modification. The word mapping is represented in the examples in Section 7.2 by the permutation of numbered locations in the diagrams. Section 7.4 shows how W can be computed based on assumptions about the structure of types, but for now it is assumed that W is given. The image of t under the retype shall systematically be written as t'.

When $W_{t,s}$ is defined for $t, s \in \mathcal{T}$, it specifies how the words within an alias of type t will appear when that alias is converted to type s. In other words the word mapping indicates how the old data appears in the new format. To this end, every $W_{t,s}$ is a $|t|$-tuple of integers in the interval $[0, |s|)$, which implies that deletion of words is disallowed. The i^{th} element of $W_{t,s}$ is written as $W_{t,s}(i)$. $W_{t,s}$ is also assumed to have no repetition—i.e., no two words of t are mapped to the same

destination position in s. Thus $|s| \geq |t|$, but equality is not required as insertions are possible. Section 7.6 indicates how some of these assumptions may be relaxed to give a more general type conversion.

The word mapping must be defined over any pair of types between which may need to be converted, in particular $W_{t,t'}$ must be defined for all $t \in \mathcal{T}$. As discussed in Section 7.2, this may be a result of variables being redeclared with different types, or of the types themselves being redefined. Although both these cases are essentially the same problem, initially only the latter is considered to simplify notation. In this case W is only required to be defined for pairs of the form (t, t') and $W_{t,t'}$ shall be written simply as W_t. Section 7.6.5 considers situations where the domain of W may need to be strictly larger than the retype.

In practice types will have some internal structure which will act as a vehicle for both defining and constraining W. This aspect is examined in more detail in Section 7.4.

The rest of this section concerns a set of aliases $\mathcal{X}$ within a single fixed object $o \in \mathcal{O}$. Each alias $x \in \mathcal{X}$ has a start address $A(x)$ in o and an associated type $T(x)$. As intimated at the start of this section, the task will be to find a new address $A'(x)$ for each alias which is consistent with the retype and word mapping in a sense which shall be defined below. $|T(x)|$ shall be abbreviated as $|x|$ and $p \in [A(x), A(x) + |x|)$ as $p \in x$. In this way x also represents its own memory range within o. Likewise $W_{T(x)}$ is abbreviated to W_x.

Thus the set $\bigcup \mathcal{X}$ is the set of addresses covered by the aliases in $\mathcal{X}$. Since it is unsafe to allow access to addresses outside o (i.e. to unallocated memory), the *compactness* property $\bigcup \mathcal{X} \subseteq o$ shall be assumed. Unfortunately, the assumption cannot just be made once and for all, because conversion may not preserve compactness as shown in Figures 7.3 and 7.4. Thus it is necessary to be able to check for compactness explicitly. This issue will be returned to in Section 7.5.

When address $p \in x \in \mathcal{X}$, p can be *converted against* x under the retype into the number $I_x(p) \stackrel{\text{def}}{=} W_x(p - A(x))$. This is the offset relative to the new location of x of the image of the memory location p given by the retype of $T(x)$. This suggests that the notion of a global conversion of the whole memory range $\bigcup \mathcal{X}$ might be defined by converting every $p \in \bigcup \mathcal{X}$ against some alias.

But there is the possibility that the same word may convert differently as in Figure 7.2, or that distinct words be conflated, against different containers. Thus global conversion includes an intrinsic consistency requirement.

In the following definition it is assumed that there is a relocation $A'(x)$ for each alias x and the image of each word $p \in x$ is considered according to the retype of x. This is given by the expression $A'(x) + I_x(p)$.

Definition 7.17 *A* consistent conversion *of $\mathcal{X}$ under the given retype is an address map A' such that, for all $x, y \in \mathcal{X}$ with $p \in x$ and $q \in y$, the following holds:*

$$A'(x) + I_x(p) = A'(y) + I_y(q) \quad \text{iff} \quad p = q \tag{7.1}$$

In other words a conversion is consistent when no word of memory is mapped differently according to two different aliases and no two distinct words are mapped to the same location according to any pair of aliases. Definition 7.1 is a partial correctness specification for the algorithm which is derived below.

For $x = y$, (7.1) is immediate from the injectivity condition for W. Moreover, the relation between x and y represented by the left-hand equation in (7.1) for fixed $p = q$ is symmetric and transitive.

(7.1) suggests that every address in the region of overlap between any pair of aliases may need to be checked and does not give a method of generating the A' values. To answer this question, it is first shown how compactness simplifies the statement of consistency.

Theorem 7.13 *Given compactness, the conversion is consistent iff the following holds for all $p \in x$:*

$$A'(x) + I_x(p) = A'(o) + I_o(p) \tag{7.2}$$

Proof: Necessity of (7.2) follows by noting that compactness ensures that $p \in o$ and taking $q = p$ and $y = o$ in (7.1). For sufficiency, let p, q, x, y be as in Definition 7.17. By the assumption that W_o has no repetition (i.e. is injective):

$$W_o(p - A(o)) = W_o(q - A(o)) \quad \text{iff} \quad p = q,$$

i.e.,

$$I_o(p) = I_o(q) \quad \text{iff} \quad p = q.$$

Hence

$$A'(o) + I_o(p) = A'(o) + I_o(q) \quad \text{iff} \quad p = q.$$

Then (7.2) gives

$$A'(x) + I_x(p) = A'(o) + I_o(p) \quad \text{and} \quad A'(y) + I_y(q) = A'(o) + I_o(q),$$

from which (7.1) follows. □

Thus every $A'(x)$ is uniquely determined once $A'(o)$ has been fixed.

Of course, there may not be any consistent conversion. The next section presents an algorithm which fails if there is no consistent conversion, otherwise it constructs the relocation (A') of one.

7.3.2 Conversion Algorithm

The following algorithm computes new locations of all the aliases relative to $A'(o)$ if there exists a consistent conversion, or fails if there is none.

for $p = A(o)$ **to** $A(o) + |o| - 1$
 for all $x \in \mathcal{X} \setminus \{o\}$ **with** $p \in x$
  if $A(x) = p$ **then**
   $A'(x) := A'(o) + I_o(p) - I_x(p)$
  else
   if $A'(x) + I_x(p) \neq A'(o) + I_o(p)$ **then**
    abort

In words:

> For each address $p \in o$ and alias x containing p, either compute $A'(x)$ from $A'(o)$ if x starts at p, or check consistency of x at p and abort on failure.

Note the compactness assumption and Theorem 7.13 ensure that aliases need only be checked against o at addresses within o. It is not necessary to check all aliases which overlap relative to each other. Theorem 7.13 is used at two points in the algorithm. Firstly if $A(x) = p$ then $A'(x)$ has not yet been found and so it is computed by rearranging (7.2). Otherwise (7.2) is used directly to check that the previously computed $A'(x)$ is valid at p.

Figure 7.7 illustrates the algorithm by showing how it operates on Figure 7.1, assuming that $A(\text{block}) = A'(\text{block}) = 0$. The aliases are relabelled, *overp as over and *ap as a, to improve readability.

Address	Action	Effect
0	$A'(\text{over}) := A'(\text{block}) + I_{\text{block}}(0) - I_{\text{over}}(0)$	$A'(\text{over}) = 0 + 2 - 2 = 0$
	$A'(\text{a}) := A'(\text{block}) + I_{\text{block}}(0) - I_{\text{a}}(0)$	$A'(\text{a}) = 0 + 2 - 0 = 2$
1	$A'(\text{over}) + I_{\text{over}}(1) = A'(\text{block}) + I_{\text{block}}(1)$	$0 + 0 = 0 + 0$

Figure 7.7: Operation of the algorithm on the conversion from Figure 7.1.

7.4 Defining the Mapping

So far the word mapping W has been assumed to have been defined. It would be tedious for the user of a dynamic modification system to have to provide a

word-by-word mapping for each type modification, so a method of constructing W automatically is now described. For this a model of what types are is needed. The idea of structures which appear in C and other languages is used. These are composed of contiguous fields which may in turn be structures or scalars. Initially type changes which consist of rearrangements and retyping of fields and of insertion of extra fields are considered. Section 7.6 discusses how this may be extended to allow deletion of fields, changes to objects other than those given by the retype and more general mappings.

The set $\mathcal{T}$ of *types* is defined recursively such that an element $t \in \mathcal{T}$ is either a scalar or an n-tuple of type values, for some positive integer n which is the number of fields. Then $|t| = 1$, if t is a scalar and $\sum_{i=0}^{n-1} |t_i|$, otherwise. For type t define the location L_t of field i by $L_t(i) = \sum_{j=0}^{i-1} |t_j|$. The assumption that there are no alignment requirements and that the size of a structure is the sum of the sizes of its fields, can be made due to the fact that all scalars are the same size.

The division of types into structures with fields suggests that the word mapping W can be automatically constructed based on a *field mapping* M which would be a partial function over $\mathcal{T}^2$ defining how the fields in one structure type map to those in another. $M_{t,s}$ would be a tuple of field indices giving the association between the fields of t and s which is defined for those pairs (s,t) where it is needed. W_t would then be defined by recursively traversing the nested fields of t progressively refining the resulting offset in t'.

The problem with this approach is that it does not permit fields at one nesting level of the structure type t to be mapped to ones at a different level in s. For example it is sometimes desirable to "flatten" a nested structure. In this situation subfields in the original type are mapped to top level fields in the new one as illustrated in Figure 7.8. Similarly it is sometimes desirable to add structure or

Old Type (t)	Variables	New Type (t')

`typedef struct`
`{`
`  scalar x,y;`
`} sub;`
`typedef struct`
`{`
`  scalar a;`
`  sub  b;`
`} info;`
	`info block;`	`typedef struct`
		`{`
		`  scalar a,x,y;`
		`} info;`

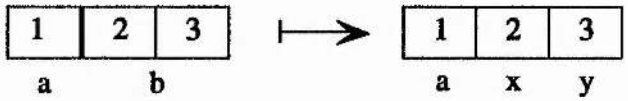


Figure 7.8: Flattening a nested structure.

simply move fields between levels in a hierarchy. A more powerful field mapping function will be introduced to cope with such situations which gives the offset within the destination type. As with the word mapping, the assumption that fields are only mapped according to the retype is made, so M only needs to be defined for pairs of the form (t, t') and $M_{t,t'}$ is abbreviated to M_t. Then $M_t : [0, n) \rightarrow [0, |t'|)$, where n is the number of fields in t. Note that the fact that M_t gives each field in t an offset in t' does not permit fields to be deleted. Section 7.6.4 relaxes this restriction. W_t is now defined using M_t and the word mappings for the fields of t.

$$W_t(p) = \begin{cases} 0, & \text{if } t = \textit{scalar} \\ M_t(i) + W_{t_i}(p - L_t(i)), & \text{if } L_t(i) \leq p < L_t(i+1) \end{cases} \tag{7.3}$$

$W_t(p)$ computes the offset in t' of the p^{th} word in t, by locating the field i of t containing p and mapping it to t' using the field mapping as $M_t(i)$. Then W is recursively invoked to compute the location of the offset of p within the field i according to $(t_i)'$. Section 7.6.5 shows how the general version of M may be used to define $W_{t,s}$, between arbitrary pairs of types.

Now clearly some restrictions need to be made on M in order to get a valid W. To this end define $F_t(i) = [M_t(i), M_t(i) + |(t_i)'|)$ for $i \in [0, n)$. Then it is required that for every $i \neq j \in [0, n)$:

$$F_t(i) \subseteq [0, |t'|) \tag{7.4a}$$

and

$$F_t(i) \cap F_t(j) = \emptyset. \tag{7.4b}$$

(7.4a) ensures that M_t projects each field of t into a valid part of t'. This property will be used to show that W_t is well defined. (7.4b) prevents any two distinct fields of t from overlapping in t'. This property will be used to show the injectiveness of W_t, which was required in Section 7.3.

Lemma 7.14 *Let t be a structure type with field i. If*

$$W_{t_i}(q) \in [0, |(t_i)'|), \qquad \forall q \in [0, |t_i|), \tag{7.5}$$

then for any address $p \in [0, |t|)$, with $L_t(i) \leq p < L_t(i+1)$, $W_t(p) \in F_t(i)$.

Proof: By definition,

$$W_t(p) = M_t(i) + W_{t_i}(p - L_t(i)).$$

Rewriting this gives:

$$W_{t_i}(p - L_t(i)) = W_t(p) - M_t(i). \tag{7.6}$$

Now $p - L_t(i) \in [0, |t_i|)$ by the definition of t and so

$$0 \leq W_{t_i}(p - L_t(i)) < |(t_i)'|, \qquad \text{by (7.5).}$$

Using (7.6) and rearranging gives:

$$M_t(i) \leq W_t(p) < M_t(i) + |(t_i)'|,$$

so $W_t(p) \in F_t(i)$. □

W_t is now shown to be well defined.

Proposition 7.15 $W_t(p) \in [0, |t'|), \quad \forall p \in [0, |t|), \forall t \in \mathcal{T}$.

Proof: The proof works by structural induction. The proposition is clearly true when t is scalar. Let t be a structure type such that for all fields i of t,

$$W_{t_i}(p) \in [0, |(t_i)'|), \qquad \forall p \in [0, |t_i|).$$

Then by Lemma 7.14, for any address $p \in [0, |t|)$, $W_t(p) \in F_t(i)$, where $L_t(i) \leq p < L_t(i+1)$, so by (7.4a), $W_t(p) \in [0, |t'|)$ and by induction this holds $\forall t \in \mathcal{T}$. □

Theorem 7.16 *W_t is injective, i.e.,* $W_t(p) = W_t(q) \Rightarrow p = q, \forall t \in \mathcal{T}$.

Proof: (Also by structural induction.) The result is clearly true for scalar types. Let t be a structure type such that for all fields i of t,

$$W_{t_i}(p) = W_{t_i}(q) \quad \Rightarrow \quad p = q. \tag{7.7}$$

Suppose $W_t(p) = W_t(q)$, for $p, q \in [0, |t|)$ with p in field i of t and q in field j say. Now Proposition 7.15 implies that (7.5) holds for all fields, so by Lemma 7.14, $W_t(p) \in F_t(i)$ and $W_t(q) \in F_t(j)$. Then (7.4b) implies that $i = j$, so

$$M_t(i) + W_{t_i}(p - L_t(i)) = M_t(i) + W_{t_i}(q - L_t(i)),$$

so

$$W_{t_i}(p - L_t(i)) = W_{t_i}(q - L_t(i)),$$

and then (7.7) implies $p - L_t(i) = q - L_t(i)$ and thus $p = q$ as required. Hence W_t is injective and by induction this is true $\forall t \in \mathcal{T}$. □

7.4.1 Automatically Deriving Field Mappings

So far field mappings have been assumed to exist where required but no indication has been given of how they are created in the implementation. It is generally undesirable for the user to have to manually work out at what offset in each destination type each field should be mapped. In most cases each field in the source type will be mapped to some field in the destination. In fact the names of the fields will often be the same. In this case the field mapping can be calculated automatically by the system by using the locations of the destination fields. There are cases where the user needs to define either individual field mappings or word mappings manually. This is done in advance of the automatic calculation of W and M and these predefined mappings are used whenever they are encountered. In Figure 7.8 the new version of the `info` type (`info'`) has no field named `b` and there is not even any one field which the `b` field is to be mapped into. In this case the field mapping for `info` needs to be defined manually as:

$$M_{\texttt{info}} = (0, 1).$$

Assuming that the `sub` type has not been redefined, (7.3) can be used to give the desired identity mapping:

$$\begin{aligned} W_{\texttt{info}} &= (M_{\texttt{info}}(0) + W_{\texttt{scalar}}(0), M_{\texttt{info}}(1) + W_{\texttt{sub}}(0), M_{\texttt{info}}(1) + W_{\texttt{sub}}(1)) \\ &= (0 + 0, 1 + (M_{\texttt{sub}}(0) + W_{\texttt{scalar}}(0)), 1 + (M_{\texttt{sub}}(1) + W_{\texttt{scalar}}(0))) \\ &= (0, 1 + (0 + 0), 1 + (1 + 0)) \\ &= (0, 1, 2). \end{aligned}$$

Section 7.6.5 explains how to cope if the `sub` type has been redefined in a way which does not produce an identity word mapping.

7.5 Additional Consistency Tests

In addition to the basic conversion consistency requirement (7.1), termed *word consistency*, there may be other irregularities present in an attempted conversion[4]. Therefore *static consistency* tests are presented which can be used to detect such irregularities after the conversion.

[4] See Figures 7.3–7.6

7.5.1 Static Consistency

Four levels of static consistency are defined each of which contains its predecessors. $\mathcal{X}$ is Level 1 statically consistent when

$$A(x) \in o, \qquad \forall x \in \mathcal{X}.$$

This means that all the aliases in $\mathcal{X}$ at least start within an allocated object. This is a rather weak form of consistency which is not even as strong as compactness, though the algorithm in Section 7.3.2 could actually be applied. Figure 7.3 showed that a conversion which is word consistent may nevertheless result in a state which violates this requirement.

$\mathcal{X}$ is Level 2 statically consistent when

$$A(x) \in o \text{ and } A(x) + |x| \leq A(o) + |o|, \qquad \forall x \in \mathcal{X}.$$

This ensures that $\mathcal{X}$ represents a set of valid aliases to parts of an object. Note that this is the same as compactness. Figure 7.4 shows a word consistent conversion which preserves Level 1 consistency but not Level 2 consistency.

Level 3 static consistency is defined using the recursive boolean function C_3 whose parameters are the type of an object and the offset and size of an alias to that object:

$$C_3(t,i,d) = \begin{cases} \text{F}, & \text{if } i < 0 \text{ or } i + d > |t| \\ \text{T}, & \text{if } i = 0 \text{ and } |t| = d \\ C_3(t_j, i - L_t(j), d), & \text{otherwise, with } L_t(j) \leq i < L_t(j+1) \end{cases}$$

This function recursively searches t for a subfield with offset i and size d, returning T if one is found and F if the interval $[i,d)$ is not nested within t or one of its subfields. $\mathcal{X}$ is Level 3 consistent when

$$C_3(T(o), A(x) - A(o), |x|), \qquad \forall x \in \mathcal{X}.$$

In other words, each alias exactly matches some subfield of the object o in both location and size. Figure 7.5 shows how a violation of this criterion may arise.

Level 4 static consistency is similarly defined using the boolean function C_4 whose parameters are the type of an object and the offset and type of an alias to that object:

$$C_4(t,i,s) = \begin{cases} \text{F}, & \text{if } i < 0 \text{ or } i + |s| > |t| \\ \text{T}, & \text{if } i < |t| \text{ and } |s| = 1 \\ \forall \text{ fields j, of s}, C_4(t, L_s(j), s_j), & \text{if } i = 0 \text{ and } |t| = |s| > 1 \\ C_4(t_j, i - L_t(j), s), & \text{otherwise,} \\ & \text{with } L_t(j) \leq i < L_t(j+1) \end{cases}$$

This function initially behaves like $C_3(t, i, |s|)$ until a field is found in t which matches s and then checks that all the fields and subfields of an alias of type s beginning at offset i in t match some subfield of t. $\mathcal{X}$ is Level 4 consistent when

$$C_4(T(o), A(x) - A(o), T(x)), \qquad \forall x \in \mathcal{X}.$$

Note that if x is scalar, it is always field consistent, as it must match some scalar subfield, so the process can stop short of refining t down to scalars and return T in the second line of the definition. Figure 7.6 shows how a conversion which satisfies all the previous conditions may still fail to be Level 4 consistent.

Static inconsistencies may arise due to careless programming before any conversions have been carried out, so these checks may usefully be applied as debugging aids at any time.

The examples in Section 7.2 showed that neither the word consistency test nor the static checks are sufficient to pick up all irregularities which might be introduced by a conversion, so both types of consistency check should be applied.

7.6 Generalizations and Practical Considerations

In order to simplify what has been presented so far, certain assumptions were made which are too restrictive in practice. These are that the aliases have already been found and all lie within a single allocated object o, and that only changes to type definitions occur and not changes to variable declarations. Structure types were assumed to be defined in terms of a single unit scalar with no alignment requirements and no fields deleted. Finally it was assumed that the word mapping W can be automatically defined in terms of a field mapping M. This section indicates how the algorithm and conditions presented can be generalized to cope with relaxation of these assumptions. The dynamic modification mechanism in Testbed contains these generalizations.

7.6.1 Arrays

Arrays may be considered as a subclass of structures, for which all the fields have the same type. In the implementation they are treated specially for efficiency reasons as it is not necessary to represent an array with a separate type descriptor for each array element as is done with structures.

7.6.2 Different Scalar Types

In practice scalar types may not have the same size but can be modelled by choosing some common unit of size (such as the byte) and defining larger types to be structures consisting of a number of these units. If alignment is required, then the location function L can be defined to take this into account. In practice simple byte copying with rearrangement will not be sufficient. In this case special conversion functions need to be defined for mapping between scalar types, such as integer to real conversion.

7.6.3 Pointers in Aliases

So far the pointers which represent aliases have been considered as separate from the data, but they are actually part of it and need to be converted. This happens after the conversion of the rest of the data. Pointers are initially copied from the old to new versions of variables and then updated with the new values computed using the algorithm given above. An additional form of inconsistency may arise when an alias contains a pointer which is not present in the host object. Since the data is only converted using the real object's type conversion, the pointer will not be updated. This situation may be added as an extra consistency constraint to be checked.

7.6.4 Deleted Fields

It may well be desirable to be able to delete fields from structures. This situation is dealt with by introducing a special *null* value ($\perp$) into the ranges of the mapping functions W and M. The algorithm must be modified to cope with the situation where the start of an alias x is located in a field which is deleted from the real object. In this case if the first non-deleted field of x corresponds to a deleted word in o, then either the algorithm can be aborted or the new address of the alias can be set to an invalid value which will show up in the word consistency test. Otherwise $A'(x)$ is computed from the image of the first word in the first non-deleted field of x.

7.6.5 Extending the Retype

So far the assumption has been made that the new type of an object is exactly that corresponding to the retype. Similarly it was assumed that the retype had the property that the new type of a field was that given by retype for its original type. These assumptions were made to simplify the notation and the algorithm.

In general it is desirable to relax them and allow $W_{t,s}$ and $M_{t,s} : [0, n) \to [0, |s|)$, where n is the number of fields of t, to be defined for some cases when $s \neq t'$. Consider Figure 7.9. Here it is assumed that the `sub` type does not change and

Old Type (t)	Variables	New Type (t')

`typedef struct`
`{`
`  scalar x,y;`
`} sub;`
`typedef struct`
`{`
`  scalar a;`
`  sub  b;`
`} info;`
	`info block;`	`typedef struct`
		`{`
		`  scalar y,x;`
		`} bus;`
		`typedef struct`
		`{`
		`  scalar a;`
		`  bus  b;`
		`} info;`

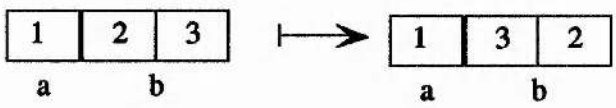


Figure 7.9: Extending the retype.

that the new type `bus` has been introduced. The `b` field in `info'` has type `bus`. The `b` field is to be mapped so that the subfields with matching names correspond, i.e., `x`↦`x` and `y`↦`y`. This would work if $M_{\texttt{sub}}$ were defined manually as:

$$M_{\texttt{sub}} = (1, 0).$$

However, there may be other variables of type `sub` or with fields of that type which retain this type in the new version. Since the `sub` type is not changed, the above definition of $M_{\texttt{sub}}$ would be wrong for these variables. An extra mapping is needed:

$$M_{\texttt{sub,bus}} = (1, 0).$$

In order to define $W_{\texttt{info,info'}}$ using this mapping instead of $M_{\texttt{sub,sub'}}$, some way of indicating the destination type of a field when mapping between a particular pair of types is needed. To this end the *local retype* $R_{t,s} : [0, n) \to \mathcal{T}$ is defined. Then the general version of the word mapping is defined as:

$$W_{t,s}(p) = \begin{cases} 0, & \text{if } t = \textit{scalar} \\ M_{t,s}(p) + W_{t_i, R_{t,s}(i)}(p - L_t(i)), & \text{if } L_t(i) \leq p < L_t(i+1) \end{cases}$$

The generalization of the F function of Section 7.4 is:

$$F_{t,s}(i) = [M_{t,s}(i), M_{t,s}(i) + |R_{t,s}(i)|).$$

Conditions (7.4a) and (7.4b) generalize in an obvious way and Proposition 7.15 and Theorem 7.16 and their proofs are similar.

As with the simpler case, the definition of $M_{t,s}$ may be automated by matching fields with like names or done manually where this does not work. The local retype can be defaulted for pairs (t, t') given by the retype to $R_{t,t'}(i) = (t_i)'$, but this can be overridden and extra mappings provided as required. Returning to Figure 7.8, the local retype would be:

$$R_{\texttt{info},\texttt{info}'} = (\texttt{scalar}, \texttt{sub}).$$

Note that the local retype of the b field is sub, not sub', so the resulting $W_{\texttt{info},\texttt{info}'}$ is the desired one even if the sub type is redefined.

7.7 Data Conversion

Whenever an existing state variable is redefined with a new type, a new variable is created with the new type and this is initialized first with the initial value provided in the definition and then with data converted from the old. This conversion is only attempted between variables of the same basic type (e.g., two structures) as described below. The old variable is retained so that it is accessible to a user supplied conversion function.

Automatic conversion is performed between similar types to the extent that scalars are converted by assignment and fields of structures/array elements are converted recursively. In the case of structures, if a field with the same name exists in the new structure then a conversion is performed. Arrays are only converted if the number of elements match. If conversion is impossible the new variables are initialized using any supplied initial value.

Data conversion of the "static" (i.e., named) state variables is performed when the redefinitions are found in case subsequent definitions use the values of their data. Pointers cannot be adjusted since it is not known what the new locations of the objects to which they point will be until the end of the reload operation. Instead the old values are copied and any new variables which are initialized with these values will be adjusted along with the old ones when the new locations are known.

After all the automatic conversions and pointer updates a user supplied conversion routine is executed if one has been supplied. At present this is specified by the user interactively before the reload operation is requested. This routine takes the place of the initialization functions which are invoked when the application is first loaded. These functions are not reinvoked during the reload, since much of the initialization that they perform will not generally be appropriate for a live system. Any reinitialization or extra conversion of the state which is required after

the reload operation is performed by the conversion function which has access to both the old and new versions of the state through function calls. To illustrate the use of conversion functions a simple example is presented.

7.7.1 Example

Figures 7.10 and 7.11 show a simple application consisting of a single slot with an action called `test`. This action accepts a message containing a string which it echoes in the window displayed by Testbed for output from this slot. The slot also contains a state variable `bedstuff` which is of the structure type `bed_info`, having fields `count` and `val`. The value of this variable is displayed by the `test` action and then updated. In version 0 `count` has type `int` while `val` has type `double` and the update consists of incrementing `count` and halving `val`. In version 1 `bed_info` has been redefined with the fields reversed and `val` has changed to `int` and is now doubled each time `test` is invoked. When the modules `bed.h`, `bedtypes.c` and `test.c` are replaced by those of version 1 (Figure 7.12), and a reload operation performed, the Testbed initializes the new version of `bedstuff` from the old one by copying the `count` field and by performing an double to integer conversion on `val`, probably resulting in it having the value 0 from then on. Since the system has no notion of the meaning of these fields, this is all that it can do.

It may be that the contents of `bedstuff` after the reload should appear as if version 1 had been in place from the start. To achieve this effect a conversion function is required. Figure 7.13 shows a function which is compiled and loaded after the `bed.h` module has been updated but before the reload. It contains a function which if specified as the user conversion function, produces the desired effect. Note that the value must be obtained from the old version of `bedstuff.val` as the new version will have lost precision when it was converted.

The functions and macros in Figures 7.10 and 7.13 are described in Appendix A.

7.8 Pointer Relocation and Conversion of Dynamic Variables

The adjustment of pointers is achieved by first locating all pointers in static variables which are preserved in the new versions of these and adding them to a set of objects and then recursively performing the same operation on the objects to which these point. Only pointers which are found to point to objects in the symbol table are followed. Any object in the symbol table which is found to be pointed to

```
/* bed.c - Configuration module contain-
ing an initialisation
   function which runs on the host slot and creates and
   initialises the other slots. */

#include "bed.h"
#include <host.h>

void bed()
{
  Slot(TESTSLOT,1,"Bed");
  Global_Declare(bedtypes);
  Module(TESTSLOT,test);
  Declare(TESTSLOT,test_decs);
}

/* test_decs.c - Declaration module which de-
fines the variables
   and ports. */

#include "bed.h"

void test_decs()
{
  bed_info init_info;

  init_info.count = 0;
  init_info.val = 1;

  Data_Port(TEST_PORT,test,char);
  Var_I(bed_info,bedstuff,init_info);
}
```

Figure 7.10: Conversion example — common part.

is added to the set of objects. It is not possible for dynamically created variables to change type, as these variables are not redefined during the reload operation. However it is possible that the definition of their types may have changed. If this is found to be the case during the search for pointers, then the object is redefined in the same way as for static variables.

Once all the pointers have been located in this way, new addresses are computed for the pointers contained in those symbols which are still in use, and consistency of the conversion is checked. If the checks fail, the user is notified and the reload operation is temporarily suspended until awaiting a continue or abort decision.

```
/* bed.h - Type declarations for the compiler. */

#define TESTSLOT 1
#define TEST_PORT 0

typedef struct bed_info
{
  double val;
  int count;
} bed_info;
#include <action_lib.h>

/* bedtypes.c - Declaration module which de-
fines the types. */

#include "bed.h"

void bedtypes()
{
  define_structure(FALSE,"bed_info",2,
                   "double","val","int","count");
}

/* test.c - Application module containing the test ac-
tion. */

#include "bed.h"

extern bed_info bedstuff;

void test(char *test_message)
{
  PrintS(test_message);
  Print_Var(bed_info,bedstuff);
  bedstuff.count++;
  bedstuff.val /= 2;
}
```

Figure 7.11: Conversion example — Version 0.

```
/* bed.h - Type declarations for the compiler. */

#define TESTSLOT 1
#define TEST_PORT 0

typedef struct bed_info
{
  int count;
  int val;
} bed_info;
#include <action_lib.h>

/* bedtypes.c - Declaration module which
   defines the types. */

#include "bed.h"

void bedtypes()
{
  define_structure(FALSE,"bed_info",2,
                   "int","count","int","val");
}

/* test.c - Application module containing the test ac-
tion. */

#include "bed.h"

extern bed_info bedstuff;

void test(char *test_message)
{
  PrintS(test_message);
  Print_Var(bed_info,bedstuff);
  bedstuff.count++;
  bedstuff.val *= 2;
}
```

Figure 7.12: Conversion example — Version 1.

```
/* When convert_bed.c is compiled the new version of
   bed_info should be defined.  We also need the old
   version to be able to perform the conversion. */

#include "bed.h"

typedef struct
{
  double val;
  int count;
} old_bed_info;

extern bed_info bedstuff;

void convert_bed()
{
  old_bed_info *oldstuffp
    = (bed_info *)old_symbol_ptr("_bedstuff");

  bedstuff.val = 1./oldstuffp->val;
}
```

Figure 7.13: Example of a user conversion function.

7.9 Preserving Consistency of Messages

To avoid version conflicts, all messages received by the slot being reloaded need to be in the old format up to the point at which the slot's waiting messages are converted and in the new format thereafter. It must not be possible either for messages in old format to arrive after the conversion or for messages in new format to arrive before conversion. To achieve this it is necessary to synchronize the reloading of all the slots which may send messages to each other containing types which are to change. It is assumed that the set of slots which are reinitialized during a reload operation includes this set and so these are synchronized. The synchronization proceeds as follows:

1. The host slot counts the number of slots which are reinitialized during the reload. Each of these slots is sent a RELOAD message which causes the slot to be suspended. This is followed by initializations (containing changed state variable and type definitions) and replacement code.

2. The host slot sends the count with a RESUME_SLOT message which follows the reload.

3. Each suspended slot performs its pointer relocation upon receiving the RESUME_SLOT message and then broadcasts a GROUP_SUSPEND message to all other slots.

4. Each suspended slot waits for count GROUP_SUSPEND messages. At this point The monotonicity of the routing ensures that the slot will receive no further messages until any of the other slots resume. The messages are converted and the slot broadcasts a GROUP_RESUME message.

5. Each suspended slot waits for count GROUP_RESUME messages. At this point the slot knows that all other slots are ready to receive messages in the new format so it is safe to resume normal operation.

7.9.1 Aborting a Reload

To allow an abort of the entire reload operation the GROUP_SUSPEND message carries a flag indicating whether the conversion of the sending slot was successful and if not the receiving slot resets all types, modules and variables to their original state before resuming. Any subsequent GROUP_SUSPEND messages from other slots are ignored.

7.10 Related Work

Other work on dynamic modification systems tends to concentrate on the problem of maintaining consistency in communication between the units of replacement, whether these be procedures [26, 38, 52], processes [50, 58], abstract data types [23, 35, 106, 91] or objects [6, 42]. Most of these systems allow types to change, but do not provide automatic mechanisms for data conversion and pointer updating. This task generally has to be performed by conversion functions which do not form part of the application itself and must be written specially for each modification which contains a type change. While such functions will sometimes be necessary for the Testbed system as a supplement to automatic conversion, the mechanism presented in this paper will generally reduce the amount of manual intervention required per modification. The OTGen system for database transformation [53] most closely resembles this work, though in an object oriented setting with classes and variables rather than structure types and fields. They allow "sharing" in which two objects may have variables which are effectively pointers to a third object. This introduces a kind of aliasing, however it appears to be much more restrictive, with aliasing of complete objects and aliases to the same region of store required to

have the same type. These restrictions eliminate the consistency problems which are addressed in this chapter.

7.11 Conclusions

Testbed provides dynamic modification at two levels. The simple approach is to allow variables to be poked with new values in order to tune parameterized application code. This requires that the programmer foresee the need to adjust parameters and provide appropriate state variables. Since more general modifications to data and code are often required and it is undesirable to stop the system, dynamic updates of individual modules and data declarations is also provided. This makes use of the natural breakpoints provided by the action model to perform the update at a safe point. The update itself uses the slot's symbol table and the dynamic linking capability which is used during initial system load.

This chapter has described the problems arising from dynamic modifications to data type definitions when there are general aliases to parts of data objects. These include the problem of consistently mapping these aliases to the new version of the state and of detecting when no consistent mapping is possible. Initially data objects were modelled as intervals of memory, divided into words, whose lengths depended on their types and aliases as subintervals with an associated type. Then data conversion was defined in terms of a retype mapping between types and a set of word mappings between pairs of types. With these minimal definitions it was possible to derive a consistency criterion and an algorithm which computes a consistent relocation of the set of aliases if possible. Structure types were then considered and the word mapping function was defined in terms of structure based mappings. The structure model of types allows static tests for the consistency of a set of aliases to be defined at different levels. These tests are useful as part of the conversion process for detecting possible errors introduced as well as at other times for general debugging.

The ability to modify data types on the fly while preserving the relationships between the data has many applications. The algorithm and consistency checks presented here are incorporated into Testbed, where they allow a developer to experiment with different versions of code without the costly process of shutting down the system and recovering the state from scratch.

One of the goals of the Testbed dynamic modification system was that the programmer should not be restricted in the kind of data interrelationships possible. Testbed allows pointers to parts of objects which may have different types. These are two features which are disallowed by most dynamic modification systems.

Because Testbed allows such general aliases it must provide the means to check that a type modification preserves consistency which is the main contribution of this work.

Chapter 8

Migration

8.1 Introduction

Testbed provides a global communication model in which the location of another slot is logically irrelevant. However since performance is usually important and often critical to embedded systems, choosing the allocation of slots to nodes is a key part of the development process. There are two aspects to this allocation which affect performance: the distance between slots which determines communication latency and the mix of slots on each node which affects the processing performance of each slot. It is therefore useful to be able to experiment easily with the location of slots and dynamic migration is desirable for the same reasons as dynamic modification.

As in the dynamic modification case, the problem of when to perform the migration is not present for Testbed applications as there is a natural breakpoint between actions. The problem of implementing process migration in Testbed may be divided into two parts: how to correctly migrate the process state and how to preserve communication and access to resources. Since Testbed supports non-virtual memory systems, the former problem includes the problem of updating pointers at the destination. Testbed's global communication model ensures that any slot can communicate with all other slots and devices from any node. The second problem reduces to one of preserving message order. As with dynamic modification, after the migration the system should reach a state which is the same as if the system had started in the new configuration, i.e., as if the slots had resided at their new locations all along. Messages to and from the migrated slot should be routed in the normal way, following a normal route. The simplest way to achieve this is to suspend all application processing until the migration is completed, all messages in transit at the time of the migration have been delivered in the correct order and all routing tables have been updated.

Migration is inevitably an intrusive operation at least for the migrated process and generally also for any processes which are communicating with it at the time of migration. Even processes which reside on the source, destination and any intermediate processors can be affected. The fact that Testbed runs on non-virtual memory systems means that all code and data needs to be copied across the network and relinked at the destination. The requirement of preserving message order requires that a quiescence protocol be used. All of these sources of delay can cause an equivalent of the probe effect which occurs in monitoring. An alternative to the simple global suspension method for routing table update is to simply suspend the source and destination nodes and allow others which may not be involved in communication with the migrating slot to continue. Each of these nodes is updated when a message is sent to the migrated slot. This version of the migration protocol is intended to reduce the impact of migration from a single long delay which may affect the correct behaviour of some critical part of the system to a number of shorter ones which can be tolerated.

8.2 Related Work

[90] surveys systems which support process migration and concludes that port or message based systems (DEMOS/MP, V) implement process migration more easily than other designs. However this work is mainly concerned with problems of continued access to resources and continuity of communication. The problem of relocating data which contains pointers is not mentioned. Many of the systems in which process migration has been implemented in have virtual memory.

XOS [67] is an active object/message based system supporting migration. Messages arriving at the old node after an object has migrated are NAKed.

DEMOS/MP [76] is a message based system with a migration mechanism in which a forwarding address is left behind. Senders are updated when they send a message to the old node (while the message is forwarded), but as the communication is synchronous (RPC style), there is no need to guard against overtaking.

Processes in V [98] execute within a logical host. Process migration is performed by copying this logical host. Precopying is used to reduce total suspension time. Messages are routed via kernel tables which are cached, with the cache being updated by the kernel broadcasting a request for the new location of a logical host after several retransmissions. During migration messages are queued, to be NAKed after the migration. Each logical host has its own address space. A future plan is to implement a global demand paging system to speed up migration.

The Accent system [105] uses a copy-on-reference scheme for migrating virtual address spaces.

In Sprite [73] the process is frozen, transferred in one step then unfrozen. Shared backing files for virtual memory make the transfer simply a matter of writing out the process' dirty pages then transferring the page table information to the new host. Sprite processes communicate via kernel calls and those which are machine dependent are directed to the machine on which the process was created. This allows communication and access to the environment to be preserved during migration. However with this approach, migrated processes incur greater overheads.

The EMPS multiprocessor [100] provides a mailbox communication mechanism. This means that only the communication paths between a process and the mailboxes which it uses need to be updated during the migration. This has the same disadvantage as the Sprite approach, that communication may be inefficient due to the location of the mailboxes. Virtual memory is used to simplify the task of migrating the process state.

8.3 Synchronous Migration

This method works by suspending all slots in the system, sending the slot state (user types, data, actions, messages) updating the location table on each node and then resuming all slots. The other slots are suspended while routing information is updated and not resumed until all outstanding messages are known to have arrived at the new location in order to preserve message order. This requires a complex sequence of acknowledged messages as illustrated in Figure 8.1. Migration is triggered by the arrival of a MIGRATE message from the host server at the migrating slot. The system action which responds to this message broadcasts a SYS_MIGRATE message to the centres on every node. Each centre then sends a SUSPEND_SLOT message to all slots on the node. While a slot is in the suspended state only system messages are processed. Once each has responded with an ACK the centre sends a MIGRATE_ACK back to the migrating slot. Once all centres have acknowledged the slot's state is transferred to the destination node in a number of messages. At the end of the state transfer the slots message queue is flushed and the slot is marked as not present in a system table before all the messages are forwarded to the new node. This is a rare case where the slot accesses and modifies its own message queue. After all messages have been forwarded a BROADCAST_RESUME message is sent to the new instance of the slot. The monotonicity of the routing ensures that once the new slot receives this it can be

sure that no further messages will arrive from the old slot and it is safe to resume the application. The new slot broadcasts a SYS_RESUME message to all centres which send RESUME_SLOT messages to each slot. At the same time the old slot sends a SYS_DELETES message to the centre which deletes it.

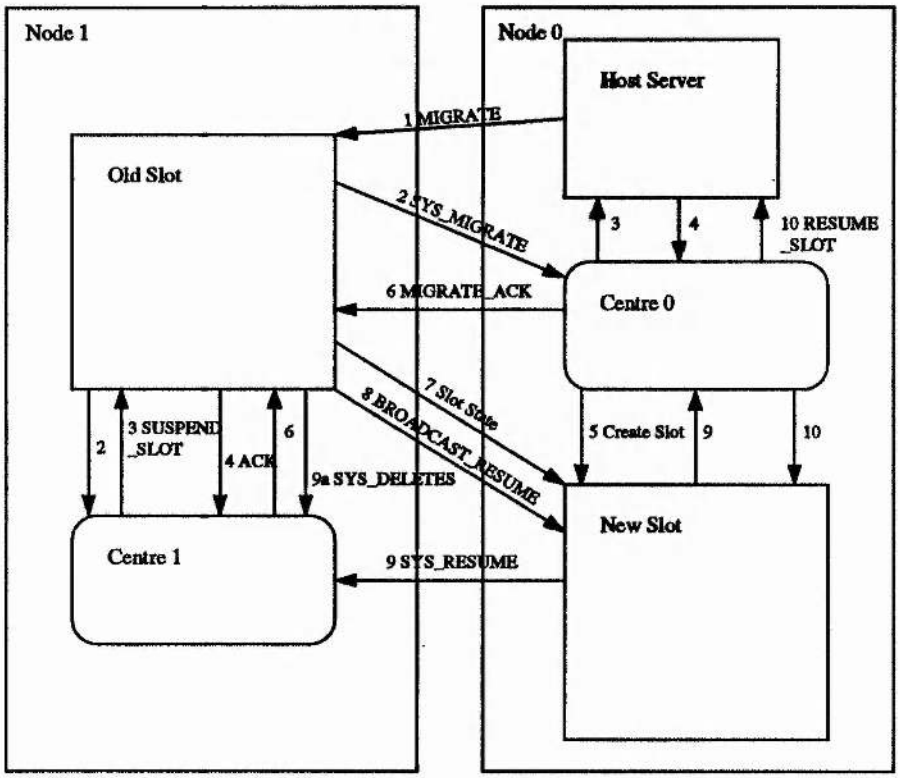


Figure 8.1: Synchronous migration of a slot onto the root node showing the sequence of messages. Messages are sent in ascending order with 9 and 9a independent.

8.4 Asynchronous Migration

The second method provided is intended to make migration a less drastic operation on a system with a large number of nodes, some of which may not be in frequent communication with the migrating slot. In this version only the source and destination nodes are frozen during the migration and only these nodes have their location tables updated to contain the new address of the migrated slot. Other node's routing tables continue to hold the old location of the migrated slot. When a message arrives at its destination node but the slot indicated by the dest field in the message header is not present, the message is wrapped up by the guardian kernel thread in a system message with the CENTRE_FORWARD port and passed to the

centre slot. The centre forwards the message to the slot's new location and then initiates the update of the source node in a similar fashion to the way the source and destination nodes were updated, except that the slot state is not transferred in this case. In the asynchronous version the BROADCAST_RESUME request has been replaced by a FORWARD_RESUME which contains the node to be resumed as data.

8.4.1 Complications

Although at first sight the asynchronous method might appear to be a straightforward extension of the synchronous one, it gives rise to several unpleasant new special cases which complicate the implementation.

- Since not all application slots have been suspended, it is possible for new messages to arrive for the migrating slot at any time. This means that the action which performs the flushing and then marks the slot as deleted must run in critical mode as the kernel must not be allowed to access the message queue during this operation and afterwards all messages should be forwarded. In order to prevent the centre from forwarding some new messages before all earlier ones have been sent from the migrating slot, the centre is suspended along with all the application slots and the CENTRE_FORWARD port is in the user range rather than the system range so that they are not actioned until after the migration.

- The slot may migrate back to a node which it resided on in the past. In this case at the end of the migration any messages which were held by the centre pending forwarding, must be flushed atomically into the slot's queues before any new messages are received. This is carried out by a critical action which scans the centre's queue in response to a new MIGRATE_DONE message sent to the new version of the slot at the end of migration.

- If the second migration is synchronous then all nodes are involved, and in particular they will all try to send MIGRATE_ACK messages to the migrating slot, which must be received. Since some nodes may not be aware of the location of the migrating slot the MIGRATE_ACKs must be sent to the originating node rather than routed in the normal way. This is possible since the message header contains the source node.

- The centre may receive SYS_MIGRATE messages for several slots concurrently. This means that it needs to keep track of who to send MIGRATE_ACK messages to when the node has been suspended and also

needs to keep count of how many SYS_RESUME messages are required before it is safe to resume the local slots.

8.4.2 Pointer Updating

As each variable is added to the symbol table at the destination, an extra *ghost* entry is added containing the address of the variable on the old node and a pointer to the real entry for the variable. Function symbols are also contained in the symbol table and are transferred to the new node. Once all variables have been transferred, (signalled by a message from the old slot) the symbol table is scanned and all the pointers in variables are updated by finding the old variable which contains them and obtaining the new address. The action pointers in the user part of the port table are updated using the same method as for pointers in application state variables. Once this process is complete all the ghost entries are removed.

8.5 Correctness Properties

The migration protocol needs to fulfill the following requirements:

1. Message order is preserved.
2. All slots which are suspended are eventually resumed.

8.5.1 Correctness of Synchronous Migration

Message Order

Messages can only arrive out of order if they are sent to the new address of a migrated slot before messages sent to the old address have arrived. In this case, due to the different route taken it would be possible for the newer messages to overtake the old ones. The event numbers in Figure 8.1 form a precedence relation on the events. This precedence prevents old messages from arriving after new ones since:

1. The centre updates the location table to contain the new node for the migrated slot before sending the RESUME_SLOT message to any local slots.
2. No messages may be sent using the updated routing information (except by the migrating slot) until a RESUME_SLOT message arrives at the sending slot.

3. The RESUME_SLOT message is sent after the arrival of the SYS_RESUME message at the centre.

4. The SYS_RESUME message is sent by the new slot after it receives the BROADCAST_RESUME message from the old slot.

5. The BROADCAST_RESUME message arrives after the slot state which includes all messages.

6. The slot state is sent by the old slot after it receives MIGRATE_ACK messages from all nodes.

7. The MIGRATE_ACK message arrives from node i after all other messages from any slot on that node. This is because:

8. The MIGRATE_ACK message is sent by centre i after it receives ACK messages from all slots on node i.

9. Each slot sends no messages between sending the ACK message and receiving a RESUME_SLOT message.

Effectively the MIGRATE_ACK messages and the BROADCAST_RESUME message flush all old messages through from the other slots to the new slot before it sends SYS_RESUME messages which trigger the resumption of other slots and allows new messages to be sent.

Resumption of Slots

All slots will clearly be resumed eventually as long as all messages eventually arrive and each process completes all of its operations in finite time. This in particular means that each slot must suspend in finite time, which is only dependent on having actions of finite duration.

8.5.2 Correctness of Asynchronous Migration

If one slot migrates once then the correctness follows in the same way as in the synchronous case. The updating of the routing information in nodes other than the source and destination nodes is delayed until after the migration but is otherwise identical. Complications arise however when a slot is migrated more than once and when multiple slots have migrated. This was discussed in Section 8.4.1.

8.6 Example

This example illustrates the use of the Testbed for performance tuning components in a simplified version of an autodepth controller (ADC) for a robot submersible (ROV). In the example the initial configuration places the ADC slot on the same processor as other slots which are performing background intensive computations. This results in a high and irregular latency or response time for the depth controller. After migration of the ADC slot to a free node, the latencies are reduced to a small constant as shown in Figure 8.2.

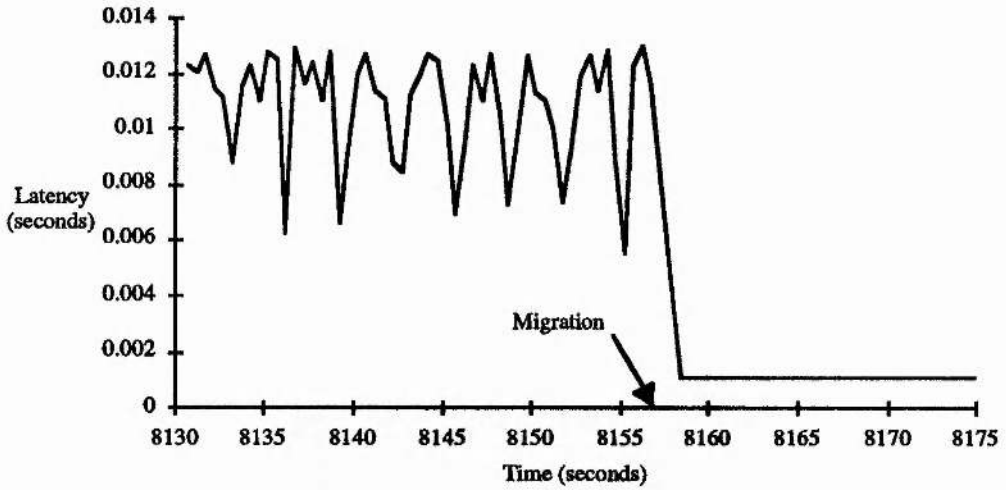


Figure 8.2: Latency as a function of time for the response by the ADC to a depth message, as measured from the ROV, before and after a migration.

8.7 Conclusions

Slot migration is a useful tool for performance tuning which is consistent with the dynamic experimentation philosophy of Testbed. Due to the fact that Testbed is implemented on a distributed non-virtual memory platform and the added requirement of preserving message order, the migration process is more difficult than in some systems. Because of Testbed's global communication model and the fact that the system cannot tell which slots might communicate, all need to be updated with the new location of the migrated slot. The simplest way of doing this is by suspending the entire application for the duration of the migration. Since this is liable to interfere with timing-sensitive activities which may not be directly involved with the migrating slot, an alternative version of migration is provided in which only the source and destination nodes are involved and others are informed of the new location of the slot when the first message is sent to the migrated slot.

Chapter 9

Conclusions

The motivation for this research is a test bed for embedded systems, analogous to an engine test bed, in which the embedded software under development can be observed, tinkered with and tuned away from the often harsh and uncompromising real-world environment. Specifically it was felt that the test bed should support multiple processors, powerful yet non-intrusive monitoring, dynamic modification and process migration. In the process of achieving these goals, the slot/action programming model has been developed which is based on experience of the nature of embedded systems software and is amenable to the goals. An operating system kernel was constructed to support this programming model. Almost all of the debugging and testing facilities are provided in a system layer above the kernel.

Testbed is implemented using a configurable multiprocessor whose processors are the same ones which are used in the target system, front ended by a Unix workstation. Due to its modularity, Testbed is highly configurable according to the application. External devices which are not available at the time of development may be simulated by using extra slots, often on separate processors which provide the same interface to the application as the real hardware. The windows based user interface was implemented as a Testbed slot, enabling it to be extended and tailored to the application.

9.1 The Testbed Programming Model

The choice of programming model for Testbed was driven by the necessity of finding breakpoints in the application code at which the debugging system could safely interrupt a process and perform monitoring, replace the code and data with a new version or migrate the entire process. With a programming model in which there are a few sequential processes or an occam-style model in which each code

entity may split up into numerous light-weight threads there are serious difficulties in identifying these breakpoints. A traditional debugging approach in which the debugger steps through the process statement by statement, allowing the user to identify breakpoints had been ruled out because of its excessive intrusiveness. The application processes (slots) had to be divided into coarser grained units (actions) which communicate with the system. This communication would then form the breakpoint.

It was observed that most embedded systems are ether reactive or periodic and that periodic systems can be thought of as reactive if timeouts are seen as events. The implementation of such systems tends to involve processes which repeatedly wait for some event and then dispatch it through a case statement or interrupt table. In Testbed this dispatcher is removed from the application into the system and the application designer is left with the task of providing the event handling action functions. Since Testbed is designed to support multiple tasks which communicate by message passing, it is natural for the application not to distinguish between external events and messages, thus each new message is an event and is processed by a new action. Timestamps in the messages combined with deadlines or priorities associated with each slot allow actions to be scheduled ahead of time and deadline/priority scheduling to be supported. The debugging operations are carried out by system actions triggered by system messages to the slot. Since at most one action may be executed per slot at a time, there are no concurrency problems, the application is guaranteed to be in the "in-between-actions" state.

Experimental implementations such as the ROV presented in Chapter 2 and also a version of a hexapod robot developed at Paisley University [59] suggest that the Testbed model is well suited to both specification and implementation of this type of embedded system.

9.2 Implementation

Testbed has been implemented on a Sun hosted Meiko Computing Surface with 24 T800 Transputers. The implementation consists of three distinct layers, the application, system and kernel.

The system layer forms a harness for the slot, replacing the `main()` function of a conventional C process. This function dispatches actions in response to messages and performs event-based monitoring. Each slot has a separate copy of this action harness thread and of system data structures such as the port table. The system also provides a library of utility functions available to the application and

a set of built in system actions for performing debugging operations requested by the user or by surrogates.

The kernel layer is responsible for delivering messages, handling synchronous devices and scheduling slots. The kernel is implemented as a collection of cooperating uninterruptable threads each of which is activated by an external or internal event and may schedule a new action harness thread or initiate a preemption.

Each Transputer has a system slot called the centre, which assists in debugging activities such as slot creation and migration as well as coordinating device configuration and single device read operations. Access to the host machine is provided through a root Transputer whose centre contains extra library functions for performing host I/O as well as actions which manage the graphical user interface.

The first generation of Transputers and similar processors do not contain hardware for routing messages. The Testbed kernel contains threads which multiplex and demultiplex messages through shared links to enable more than one slot to communicate using the same link and to allow messages to be sent to non-adjacent nodes. This routing is deterministic due to the requirement of predictability for embedded systems and is based on spanning trees rooted at each destination node. It is important that this routing be free from deadlock. Chapter 5 gives a criterion for deadlock-freedom and a simple algorithm is described for attempting to minimize path length and link load while maintaining deadlock-freedom. Methods of constructing complex regular networks from simpler ones while preserving deadlock-free and path-optimal routings are also discussed.

The modular, microkernel nature of the system allowed the debugging features of Testbed to be implemented using the same model as applications, i.e., as built-in actions in each slot which are invoked by, and communicate using messages. It is also easy for the user to customize the debugging system to a particular application by adding monitoring and simulation functions, actions and slots alongside the application. Extra widgets can be added to the graphical interface along with actions on the host server slot which display information in application specific ways.

9.3 Testing and Debugging

Monitoring and debugging of distributed systems is a much studied field, however most work concentrates on the problem of detecting temporal rather than timing problems. Testbed provides a range of event and state monitoring facilities allowing the user to select those which cause least interference, based on characteristics of the application. These monitoring facilities exploit the natural breakpoints

provided by action invocation, sending of messages and termination. The Testbed programming model makes it easy to add application specific monitoring code in the form of embedded code, extra actions or slots which can report information in more detail than is possible with the standard facilities. The role of the user in the debugging process can be partially automated by background debugging where these additions, then known as surrogates, are used to detect and react to behaviour patterns. Background debugging is a useful way of avoiding the probe effect when the rate of events is high or when timing is critical, as is often the case with embedded systems.

9.4 Dynamic Experimentation

Due to real-time and performance aspects, embedded systems development often involves a great deal of tuning as well as logical correctness testing. In this phase, the developer must perform numerous iterations of testing and modification. The start-up cost of embedded systems is often very high and so it is useful to be able to experiment with the system dynamically. The simplest method of achieving this is to allow the values of state variables to be poked by the user. This is a facility provided by most conventional interactive debuggers. Testbed takes it much further by allowing new versions of the application code to be loaded while the system continues to run. Modification of user defined data types is supported and conversion of the data between versions is automated[1]. Since Testbed supports the C language for applications programming, the conversion problem includes that of updating pointers and checking for consistency in the new assignment. This makes up the bulk of Chapter 7. The dynamic modification capability is important since Testbed does not support source level interactive debugging[2]. Instead the same functionality is provided by embedded code and surrogates. To achieve the power of an interactive debugger, the user must be able to make and modify these additions on-the-fly, rather than having to stop and restart the entire system with the extra code in place.

Testbed also provides dynamic slot migration in order to manually balance the processing load or adjust communication latencies. Testbed's global communication model ensures that migration is transparent to the application. The implementation uses a protocol which ensures that message order is preserved and that routing tables are updated following migration so that the effect is of the slot having resided on its destination processor all along.

[1] supplemented by user supplied conversion functions.

[2] as it is too intrusive for embedded systems

9.5 Major Original Contributions

The most significant original contributions of this thesis are

Chapter 5 in which

- a criterion for deadlock freedom is presented,
- a simple but effective static routing optimization algorithm is described which preserves deadlock-freedom,
- methods of constructing complex networks from simple components and of extending routings to preserve deadlock-freedom and path-optimality are studied. These results generalise earlier work on specific networks such as hypercubes and routings such as the e-cube algorithm.

Chapter 7 in which dynamic modification facilities are described. The C language, commonly used for embedded systems programming, allows pointers to fields of structures and pointers whose base type differs from their target. The aliases which these pointers give rise to are formally modelled and an algorithm is derived which performs a valid pointer relocation, if possible, during data restructuring.

9.6 Further Work

The Testbed project covers several diverse fields with many complex sub-problems and it has not been possible to cover all of these in as much depth as they deserve. Although consisting of almost 10000 lines of C source code, the current implementation of Testbed is little more than a prototype. This section indicates the potential for enhancement of the Testbed as a development system and for further research both within the embedded systems arena and in more general contexts.

9.6.1 Adding Features to Testbed

Testbed has been designed in such a way as to enable new features to be added easily as well as application specific testing and debugging aids. The library functions, system actions and user commands provided by the current version of Testbed and detailed in Appendices A and B are a small subset of those which would be required if Testbed were a finished product. They represent little more than the bare minimum required to implement the goals of the Testbed and example applications such as the ROV.

The programming interface is currently rather primitive, with only the C language supported. Since Testbed is an object-based system it would seem natural to add support for object oriented languages. Even with the current language support, the requirement of providing a function call for each global variable declaration and user type definition is cumbersome. It could be replaced with a preprocessor which extracts these from the source code and creates the function calls automatically. Testbed currently requires raw statically allocated numbers for identifying slots and devices. A simple enhancement would be to provide a name server slot. Other system server slots could also be provided.

No configuration or change management (apart from the Unix make utility) is currently provided. Having applications specified[3] using a configuration function written in C and loaded initially on the host slot simplified the implementation, but restricts the possibilities for dynamic code modification to either reloading a single module or reloading the entire system. It also makes it difficult for dynamic changes such as to the assignment of slots to nodes to be included automatically in the application specification. A separate configuration file which could be displayed and modified graphically would enhance the users view. Testbed could even be enhanced to a complete CASE tool including a graphical programming extension for constructing, configuring testing and debugging applications.

9.6.2 Device Support

The only devices supported by the current implementation are those connected through Transputer links. It is expected that shared memory devices could be supported at the application level since they do not require blocking to read. The device handler provided by the kernel is configurable to a limited extent by selecting the destination slot and port for either one-off read operations of variable sized data or continuous input of fixed sized messages. Greater flexibility could be added by further parameterizing the device handlers or by allowing user functions to be called from them.

9.6.3 Heterogeneous Systems

Testbed currently only supports Transputer networks, however many embedded systems are implemented using a variety of different processors and so it would clearly be useful to be able to run Testbed on heterogeneous systems. Although the current Testbed implementation is Transputer specific, the programming model is

[3] apart from the separate network description file

not, in fact Testbed programs are fundamentally different from the conventional CSP model normally used for Transputer applications.

9.6.4 Using Memory Protection

Since the first generation of Transputers do not provide memory protection, misbehaving application code can easily crash the system. During development this is unhelpful as the failed program cannot be probed to determine the cause of the crash and the system must be restarted. Once the system is in the field the consequences of a crash may be much more severe. If Testbed were implemented on architectures such as the Inmos T9000 Transputer [45] which provide memory protection then the system could be protected from damage by the application and the ability to trap errors such as invalid memory accesses could be used to halt an errant action and allow the developer to examine the state and determine the source of the failure.

9.6.5 Real-Time Scheduling

Although an effort has been made to make the BED kernel predictable and to provide support for delayed actions and deadline-driven scheduling, no comprehensive effort has been made to determine the bounds on system operations either formally or by measurement. The problem of analyzing the interactions between multiple communicating slots has not been dealt with either.

In order to increase predictability, the aim was that the overhead of inserting a new message into the schedule be bounded by a multiple of the number of slots rather than dependent on other factors such as the number of messages queued. This together with a need for simplicity and the requirement of preserving message order, led to the two tier schedule consisting of a FIFO queue for messages waiting at each slot and a deadline ordered pool with at most one entry per slot. However to provide greater flexibility to the application programmer, it would be desirable to be able to specify that the slot manage its own queue with an application specified policy. This may increase the scheduling delay for particular slots but if implemented within the system layer need not affect the overall performance or predictability of the kernel. In addition, alternatives to the earliest deadline first policy which the kernel uses to schedule slots could be specified by the application.

9.6.6 Routing

Only one link[4] is currently allowed between any two nodes and the route a message takes is entirely determined by its destination node. In some embedded applications it might be useful to have more than one link between the same pair of nodes and to be able to specify that the route depend on the message type and the source and destination slots. For example a link might be reserved for high priority traffic which should not be held up by other less important messages.

Most of the work in Chapter 5 is not specific to Testbed and could be applied to general parallel processors. There is a great deal of scope for extending the simple algorithm presented in Section 5.4.1 through the use of probabilistic algorithms and for experimentation with different cost functions. There are many more ways of constructing networks out of simple components than those studied in Section 5.5.

9.6.7 Dynamic Modification

Dynamic modification is a very complex issue and there are many situations where the facilities provided by Testbed could be enhanced. For example the user can provide neither the word mapping function of Section 7.3.1 nor the field mapping of Section 7.4. Instead these are computed automatically by matching array elements and fields with like names. These restrictions rule out moving fields between nesting levels as in Figure 7.8.

Dynamic modification and process migration in Testbed are provided as development and debugging aids, however they are often employed in the field in order to dynamically adapt to changing load (in the case of migration) and to update systems for which the cost of a complete shut-down would be too high. In these situations safety becomes a much more important issue than during development.

Chapter 7 considered only C types and did not even cope with unions. In more advanced languages higher level types with inheritance need to be considered and will require a more abstract view than taken here.

9.6.8 Background Debugging

Background debugging is a very active field [66] which has only been touched on here. Testbed provides the basis for surrogates which perform background debugging but no automatic mechanisms are provided for mapping specifications into code for behaviour capture and reaction. There is a need for background debugging research in the real-time field where techniques which introduce a high

[4] not counting raw links

overhead into communication such as vector timestamps cannot be used. Although the real timestamps which Testbed uses cannot capture causal relationships, they make up for it by providing the timing information necessary in debugging real-time systems.

9.6.9 Fault Tolerance

Embedded systems are often used in situations where failure can be costly both financially and in terms of human life[5]. For this reason fault tolerance is often traded off against cost and performance in the design of the system.

The current implementation of Testbed does not provide built-in fault tolerance. Although it is possible for the designer to build fault tolerant systems at the application level by using redundant processors and higher level protocols, these capabilities or support for them could be added to the Testbed system itself.

9.7 Closing Remarks

Embedded systems are an increasingly important application of computer technology, both software and hardware. Yet this is an area which has been largely neglected by computer scientists as evidenced by the scarcity of journals and conferences devoted to it. The main focus of current research efforts is in the important field of real-time scheduling, however this is not the only aspect which deserves study. There is currently only one IEEE and ACM sponsored symposium and one workshop devoted to real-time systems and no journals published by either body. This contrasts with the large number of publications and conferences devoted to parallel and distributed systems. The issues involved in designing embedded systems are not the same as those in general purpose parallel and distributed computing and these publications and conferences seldom carry papers in this field.

Debugging is another neglected area with a single biannual ACM sponsored workshop devoted to parallel and distributed debugging. Many in the software engineering field espouse the view that "debugging should not be necessary in an ideal world". However this view ignores the fact that transforming requirements into specifications is a highly error-prone activity which can never be completely formalized. The debugging phase is as important in understanding the problem and getting the specification right as it is in detecting programming errors.

This thesis has addressed the important, yet difficult problem of testing and

[5] safety critical systems

debugging an embedded system. While the highly varied nature of these systems makes it impossible to produce a completely general solution, it is hoped that the approach embodied in Testbed is appropriate to a significant range of potential applications.

Bibliography

[1] Z. Aral and I. Gertner. High-Level Debugging in Parasight. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 151–162, June 1988.

[2] Z. Aral and I. Gertner. Non-Intrusive and Interactive Profiling in Parasight. In *Proceedings of the Symposium on Parallel Programming*, pages 21–30, July 1988.

[3] N. Audsley. Deadline Monotonic Scheduling. Technical Report YCS 146 (1990), University of York - Department of Computer Science, 1990.

[4] N. Audsley and A. Burns. Real-Time System Scheduling. Technical Report YCS 134 (1990), University of York - Department of Computer Science, 1990.

[5] F. Baiardi, et al. Development of a Debugger for a Concurrent Language. In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 98–106, March 1983.

[6] J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 311–322, May 1987. San Francisco.

[7] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behaviour. *ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[8] P. Bates and J.C. Wileden. An Approach to High-Level Debugging of Distributed Systems (Preliminary Draft). In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 107–111, March 1983.

[9] C. Berge. *Graphs and Hypergraphs.* North-Holland, 1973.

[10] F. Brinksma (Ed.). LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard ISO 8807 (draft).

[11] B. Bruegge and P. Hibbard. Generalized Path Expressions: A High Level Debugging Mechanism. In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 34–44, March 1983.

[12] P. Burgess and M.J. Livesey. A Testbed for Embedded Transputer Systems. Position paper, Workshop on Abstract Machine Models for Highly Parallel Computers, Leeds, March 25–27 1991.

[13] P. Burgess, M.J. Livesey, and C. Allison. A Testbed for Embedded Transputer Systems. *IEE Computing and Control Divison Digest No: 1992/204*, 1992. Colloquium on Applications of Parallel and Distributed Processing in Automation and Control, Savoy Place, London, 13 November, 1992.

[14] P. Burgess, M.J. Livesey, and C. Allison. An Execution Harness for Transputer Based Embedded Systems. In J. Kerridge, editor, *Transputer and Occam Research: New Directions*, volume 16, pages 25–40. IOS Press, 1993.

[15] P. Burgess, M.J. Livesey, and C. Allison. BED: A Multithreaded Kernel for Embedded Systems. In *Proceedings of the 19th IFAC/IFIP Workshop on Real Time Programming*. Pergamon Press, 1994.

[16] C. Caerts, R. Lauwereins, and J.A. Peperstraete. PDG: a Process-Level Debugger in GRAPE. In *ACM SIGSOFT'92*, 1992.

[17] J.D. Choi and J.M. Stone. Balancing Runtime and Replay Costs in a Trace-and-Replay System. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*. In ACM SIGPLAN Notices, 26(12):26–35, December 1991.

[18] A. d'Acierno, G. De Pietro, and U. Villano. A Method for Monitoring Occam Internal Channels. In *OUG-12 Tools and Techniques for Transputer Applications*, pages 190–197. IOS Press, 1990.

[19] W.J. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[20] W.J. Dally, et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, pages 23–39, 1992.

[21] M. Diaz. Modelling and Analysis of Communication and Cooperation Protocols using Petri Net Based Models. In C. Sunshine, editor, *Protocol Specification, Testing and Verification*, pages 465–510. North-Holland Publishing Company, 1982.

[22] J.T. Draper and J. Ghosh. Multipath E-Cube Algorithms (MECA) for Adaptive Wormhole Routing and Broadcasting in k-ary n-cubes. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 407–410. IEEE Computer Society Press, March 23–26 1992.

[23] R.S. Fabry. How to Design a System in Which Modules Can Be Changed on the Fly. In *Proceedings of the Second International Conference on Software Engineering*, pages 470–476, 1976.

[24] C.J. Fidge. Reproducible Tests in CSP. *The Australian Computer Journal*, 19(2):92–98, May 1987.

[25] C.J. Fidge. Detecting Synchronisation Errors in Occam Programs. In *Proceedings of the 12th Australian Computer Science Conference*, February 1989.

[26] O. Frieder and M.E. Segal. On Dynamically Updating a Computer Program: From Concept to Prototype. *Journal of Systems Software*, 14:111–128, February 1991.

[27] J. Gait. A Debugger for Concurrent Programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[28] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[29] P.T. Gaughan and S. Yalamanchili. Adaptive Routing Protocols for Hypercube Interconnection Networks. *IEEE Computer*, 26(5):12–23, May 1993.

[30] D. Gelernter. A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks. *IEEE Transactions on Computers*, 30(10):709–714, October 1981.

[31] A.P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring Consistent Global States of Distributed Computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*. In ACM SIGPLAN Notices, 26(12):144–154, December 1991 1991.

[32] G.S. Goldszmidt, K. Shmuel, and S. Yemini. Interactive Blackbox Debugging for Concurrent Languages. *ACM SIGPLAN Notices*, 24(1):271–282, January 1989.

[33] H. Gomaa. Software Development of Real-Time Systems. *Communications of the ACM*, 29(7):657–668, July 1986.

[34] A.J. Gordon and R.A. Finkel. Handling Timing Errors in Distributed Programs. *IEEE Transactions on Software Engineering*, 14(10):1525–1535, October 1988.

[35] H. Goullon, R. Isle, and K-P. Löhr. Dynamic Restructuring in an Experimental Operating System. *IEEE Transactions on Software Engineering*, 4(4):298–306, July 1978.

[36] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. In Operating Systems Review 25(5):68–80, October 1991.

[37] I. Graham and T. King. *The Transputer Handbook*. Prentice Hall, 1990.

[38] D. Gupta and P. Jalote. On-line Software Version Change Using State Transfer Between Processes. *Software—Practice and Experience*, 23(9):949–964, September 1993.

[39] W.A. Halang. Load Adaptive Dynamic Scheduling of Tasks with Hard Deadlines Useful for Industrial Applications. *Computing*, 47:199–213, 1992.

[40] S.O. Hallsteinsen. Source Level Debuggers: Experience from the Design and Implementation of CHILLscope. In *International Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag, 1986.

[41] D. Hamlet. Debugging "Level": Step-Wise Debugging. In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 4–8, March 1983.

[42] G. Hedin and B. Magnusson. Supporting Exploratory Programming in Simula. Technical Report LU-CS-TR:88-31, Lund University, 1988.

[43] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[44] W. Hseush and G.E. Kaiser. Modeling Concurrency in Parallel Debugging. In *Proceedings of the Symposium on Principles & Practice of Parallel Programming*, pages 11–20, March 1990.

[45] Inmos Ltd. *The T9000 Transputer Hardware Reference Manual*, 1 edition, 1993.

[46] F. Jahanian and A. K-L. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[47] E.D. Jensen. The Kernel Computational Model of the Alpha Real-Time Distributed Operating System. In A.K. Agrawala, K.D. Gordon, and P. Hwang, editors, *Mission Critical Operating Systems*. IOS Press, 1992.

[48] M. Johnson. The Inquest Transputer Network Debugger. In J. Kerridge, editor, *Transputer and occam Research: New Directions*, pages 1–10. IOS Press, 1993.

[49] P.B. Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 78–84, June 1990.

[50] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[51] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[52] I. Lee. DYMOS: A Dynamic Modification System. Technical Report 503, Computer Science Department, University of Wisconsin-Madison, May 1983.

[53] B. Staudt Lerner and A.N. Habermann. Beyond Schema Evolution to Database Reorganisation. In *OOPSLA90*, pages 67–76, October 1990.

[54] C-C. Lin and R.J. LeBlanc. Event-based Debugging of Object/Action Programs. *ACM SIGPLAN Notices*, 24(1):23–34, January 1989.

[55] D. H. Linder and J. C. Harden. An Adaptive and Fault Tolerant Wormhole Routing Strategy for *k*-ary *n*-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[56] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[57] M.J. Livesey, P. Burgess, and C. Allison. An Integrated Approach to the Development and Testing of Embedded Systems. In *Proceedings of the Workshop on Design Methodologies for Microelectronics and Signal Processing, Gliwice-Cracow, Poland*, 20–23 October 1993.

[58] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[59] D.R. Matthew and K.T. Macfarlane. A Distributed Walking Robot Controller. In *Transputer Applications and Systems '93*, volume 1, pages 97–105. IOS Press, 1993.

[60] C. E. McDowell and D.P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[61] Meiko Ltd. *SunOS CSTools.*

[62] Meiko Ltd. *CSTools C Compiler.*

[63] D.S. Meliksetian and C.Y.R. Chen. Optimal Routing Algorithms and the Diameter of the Cube-ConnectedCycles. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1172–1178, October 1993.

[64] B.P. Miller. DPM: A Measurement System for Distributed Programs. *IEEE Transactions on Computers*, 37(2):243–251, December 1988.

[65] B.P. Miller and J-D Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.

[66] B.P. Miller, et al., editor. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*. In ACM SIGPLAN Notices 28(12), 1993.

[67] B.P. Miller and D. Presotto. XOS: An Operating System for the X-TREE Architecture. *ACM Operating Systems Review*, 15(2):21–32, April 1981.

[68] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[69] R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing '92*, 1992.

[70] L.M. Ni and K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, February 1993. Good survey of the various routing techniques which describes the deadlock problem.

[71] O'Reilly & Associates, Inc. *The Definitive Guides to the X Window System*, 1993.

[72] G. Ostheimer. Parallel Functional Programming for Message-Passing Multiprocessors. PhD Thesis CS/93/8, University of St Andrews, March 1993.

[73] J.K. Ousterhout, et al. The Sprite Network Operating System. *IEEE Computer*, pages 23–36, February 1988.

[74] Parsys. IDRIS Technical Overview.

[75] M.K. Ponamgi, W. Hseush, and G.E. Kaiser. Debugging Multithreaded Programs with MPD. *IEEE Computer*, 8(3):37–43, May 1991.

[76] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119. in Operating Systems Review 17(5), October 1983.

[77] F.P. Preparata and J. Vuillemin. The Cube-Connected Cycles: A Versitile Network for Parallel Computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[78] D.J. Pritchard and D.A. Nicole. Cube Connected Möbius Ladders: An Inherently Deadlock-Free Fixed Degree Network. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):111–117, January 1993.

[79] Ready Systems. *VRTX32 Programmer's Guide.*

[80] P.K. Rowe and B. Pagurek. Remedy: A Real-Time Multiprocessor, System Level Debugger. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 230–240, 1985.

[81] M. Rozier, et al. Overview of the CHORUS Distributed Operating System. Technical Report CS/TR-90-25, Chorus Systèmes, 1990.

[82] T.G. Saponas and R.B. Demuth. The Distributed iRMX Operating System. In K.D. Gordon A.K. Agrawala and P. Hwang, editors, *Mission Critical Operating Systems*, volume 1 of *Studies in Computer and Communications Systems*, chapter 16, pages 208–231. IOS Press, 1992.

[83] W. Schütz. *The Testability of Distributed Real-Time Systems.* Kluwer Academic Publishers, 1993.

[84] K. Schwan, H. Zhou, and A. Gheith. Real-Time Threads. *Operating Systems Review*, 25(4):35–46, October 1991.

[85] M.E. Segal and O. Frieder. On-The-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, March 1993.

[86] A.C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[87] K.M. Shea, M.H. Cheung, and F.C.M. Lau. An Efficient Multi-Priority Scheduler for the Transputer. In A.R. Allen, editor, *Transputer Systems — Ongoing Research*, pages 139–153. IOS Press, 1992.

[88] K.D. Shere and R.A. Carlson. A Methodology for Design, Test, and Evaluation of Real-Time Systems. *IEEE Computer*, 27(2):35–48, February 1994.

[89] E.T. Smith. A Debugger for Message-based Processes. *Software Practice and Experience*, 15(11):1073–1086, November 1985.

[90] J.M. Smith. A Survey of Process Migration Mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):28–40, July 1988.

[91] M. Stadel. Object Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program. *ACM SIGPLAN Notices*, 26(1):99–108, January 1991.

[92] J.A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

[93] J.A. Stankovic and K. Ramamritham. The Spring Kernel. In K.D. Gordon A.K. Agrawala and P. Hwang, editors, *Mission Critical Operating Systems*, volume 1 of *Studies in Computer and Communications Systems*, chapter 9, pages 86–117. IOS Press, 1992.

[94] A.D. Stoyenko and L. Georgiadis. On Optimal Lateness and Tardiness Scheduling in Real-Time Systems. *Computing*, 47:215–234, 1992.

[95] H. Sullivan and T. R. Brashkow. A Large Scale Homogeneous Machine. In *Proceedings of the 4th Annual Symposium on Computer Archtecture*, pages 105–124, 1977.

[96] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1981.

[97] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[98] M. M. Theimer, K.A. Lantz, and D.R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, 1985.

[99] H. Tokuda, K. Makoto, and C.W. Mercer. A Real-Time Monitor for a Distributed Real-Time Operating System. *ACM SIGPLAN Notices*, 24(1):68–77, January 1989.

[100] G.J.W. van Dijk and M.J. van Gils. Efficeint Process Migration in the EMPS Multiprocessor System. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 58–66. IEEE Computer Society Press, March 23–26 1992.

[101] T. von Eicken. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture.* ACM Press, May 1992.

[102] H.F. Weddle, et al. DRAGON SLAYER/MELODY A Highly Adaptive Distributed Operating System for Mission Critical Computing. In K.D. Gordon A.K. Agrawala and P. Hwang, editors, *Mission Critical Operating Systems*, volume 1 of *Studies in Computer and Communications Systems*, chapter 11, pages 131–145. IOS Press, 1992.

[103] C. Whitby-Strevens. The Transputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 292–300. IEEE Computer Society Press, June 1985.

[104] J. Xu and D.L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems*, pages 132–144, December 1991.

[105] E.R. Zayas. Attacking the Process Migration Bottleneck. *ACM Operating Systems Review*, 21(5):13–24, November 1987.

[106] S.B. Zdonik. Can Objects Change Type? Can Type Objects Change? In F. Bancilhon and P. Buneman, editors, *Workshop on Database Programming Languages, Roscoff, France*, pages 241–247, September 1987.

Appendix A

Testbed v1.0 User Guide and Reference

A.1 Overview

This guide is divided into three sections. Section A.2 explains how to invoke the Testbed (command line parameters and environment setup). Section A.3 describes how to use Testbed (commands etc). Section A.4 describes how to write action functions and structure a Testbed application. Section A.5 lists all standard C library functions, and gives detailed descriptions of all the special functions, which are callable from Testbed actions.

A.2 Running the Testbed

The Testbed is invoked with a command of the form:

```
testbed <route> [ -c <application> ] [ -m <makefile> ] [ <X args> ]
```

`<route>`: a file which defines the interconnection and routing information for the desired network, in a form described in Section A.2.2.

`<application>`: the name of a module (without extension) containing an initialization function with the same name as the module which when run by the host server slot, creates all the slots for the target application and downloads application, init and declaration modules[1]. If the application and makefile parameters are omitted, then the Testbed will start up with just the host server slot defined.

[1]Described in Section A.4

<makefile>: Before loading any user actions (including the main application definition), the host server performs a "make -f <makefile> all" system call, or "make all", if the makefile parameter is omitted.[2]

<X args>: Standard X windows arguments such as -display <display-name>.

If either the display is specified on the command line or the DISPLAY environment variable is defined, then Testbed attempts to open X windows on that display. Due to a bug in the Meiko Xlib implementation, the display must contain an host name and not an IP address.

A.2.1 Environment Setup

In order to run Testbed, other than with all files in the same directory, the environment variables: XAPPLRESDIR, MODULEPATH and TPATH.

A.2.2 Routing Files

The routing file serves to define both the connection topology and the routing function at the same time. It takes the form of an integer n giving the number of nodes in the system, followed by an $n \times n$ matrix. The element in row i, column j of this is the next node from j on the path to i (where the nodes are numbered $0..n-1$). Testbed will attempt to configure the Computing Surface so that each pair of nodes which are adjacent in the routing are connected by a hard link. If this is not possible due to insufficient links then the system will abort. At the end of the file extra raw links between nodes and external device links may be specified. External device links are specified as follows:

```
e <i> <portname> <id>
```

where <i> is the node to which the device link should be attached, <portname> is the name of the external Computing Surface port[3] to which the link should be connected and <id> is a positive integer to be used as the device id. Messages to the device should be sent as normal Testbed messages with the device id in place of a slot id. This id may not be used for slots.

Raw links are specified as follows:

```
r <i> <j> <i_id> <j_id>
```

[2] An example make file may be found in the rov directory.

[3] See Meiko Documentation.

where <i> and <j> are nodes and <i_id> and <j_id> are their device ids. This causes an extra link to be allocated between <i> and <j> (independent of whether the nodes are connected by the normal routing network). Each end of this link appears as a device with the given id.

The node to which device or raw links are specified must have sufficient links free after the routing network is set up or the configuration will fail. Adding an entry for a device or raw link is sufficient for the application to be able to send messages to it. In order to receive input from such a link, however, the configure_device system call must be used as described below.

A.3 Using Testbed

The Testbed X windows interface features a main root window associated with the host server slot and a separate window for each application slot. Commands are selected from drop-menus attached to each window. These menus are accessed via a button labelled "Testbed Slot Commands", located in a system command box at the top of each window. The main Testbed window, which doubles as the output window for the host slot, contains an extra menu labelled "General Testbed Commands", containing functions which are not specific to a slot. Below the system command box in each window is another reserved for application specific controls. The application command boxes are empty by default, but the application may add widgets to it.

A.3.1 Testbed Commands

The standard Testbed menus currently contain the following commands. Those commands which require additional information (indicated by trailing "...") pop up a dialog box.

Testbed Slot Commands:

Load Module... Loads a module. The module may replace an existing one, and all modules will be relinked, so that any calls to functions in the replaced module are updated.

Initialize... Loads and invokes an initialization module.

Send Message... Sends a message to the slot. Only text data may be supplied. Defaults: port_id = NULL_PORT, delay = 0.

Conversion... Specifies the name of an extra conversion function which operates after the automatic data conversion to perform reinitialization and extra data conversion which cannot be achieved by simple remapping. During a reload operation, init functions are not reinvoked, so any reinitialization or initialization of additions to the state must be performed by this function.

Event Monitoring... Allows monitoring of one of the four subevents: message arrival, action invocation, action termination and message send for a given port. Arrival, invocation and termination monitoring requests should be directed to the destination slot (through its window), while send monitoring requests should be directed to the sending slot. Different level of detail options are supported or the **Don't Monitor** option allows cancellation of monitoring for a particular subevent. In addition a variable name may be given, in which case the value of this variable is logged along with the subevent and reported at the same time. Several variables may be monitored by making several requests. Variables cannot be monitored on message arrival. If a variable name is given for this case it will be ignored. The detail levels reported are as follows:

None: No information is reported about the event, but if variables are to be monitored then these are logged and reported.

Min: Only the port id, subevent and the subevent timestamp are reported.

Basic: As with Min but the source (in the case of arrival or invocation) or destination (in the case of send) is also reported.

Full: As with Basic but the size and the message timestamp are also reported.

The event reports are printed in the scrolling output window for the slot, while the values of monitored variables are displayed in the display panel available from the **Variables** menu. To select several variables to display, these must be selected separately.

The following are not present on the main Testbed window as the host server slot may not be migrated:

Migrate (synchronous)... Migrates the slot to a different node, updating location tables on all nodes to contain the new location of the slot. This may cause significant delays as processing on all nodes is suspended while the routing tables are updated. However, once the migration is complete the application's performance will be as if the slot had been loaded onto the destination node initially. Default target node is 0.

Migrate (asynchronous)... Migrates the slot to a different node, updating only the source and destination node's location tables before allowing application slots on both source and destination nodes to continue processing. The original node forwards any subsequent messages for the migrated slot and updates the routing table at the sending node. The effect of this is that there will be extra delays when a slot on a node other than the source or destination of the migration first sends a message to the migrated slot. The message (or messages if there are several within a short space of time) are delayed as they have to follow an indirect route and the entire sending node is also delayed as it must be suspended while its routing table is updated. There is also a small overhead introduced to the forwarding node and the final destination node.

Variables Menu:

This menu contains an entry for each application variable which has been defined in the slot and additionally for each peeked variable (there are some system variables which may be peeked, such as `this_node`). Selecting a variable from this menu causes a display to appear (which may be moved and resized) through which single or periodic peeks may be performed, as well as pokes. The display shows the time in seconds since startup at which the last value was reported as well as the slot id, name, type and value of the variable. Selecting a variable for the first time causes a peek request to be sent. Entering a positive peek period, and selecting the **Peek** button causes the variables value to be reported at regular intervals. If the variable was already being reported periodically, then the period will be changed after the next report. Note that the same display is used for showing the values reported during event monitoring, which may appear at a different rate than indicated by the period in the dialog. To control these reports the **Event Monitoring** command on the main menu should be used as described above. The variables menu also contains the following entries:

Peek... Displays the value of a given state variable. Arrays and structures are displayed with their elements and fields but pointers are simply displayed as addresses in hexadecimal. The output is displayed in the appropriate slot window. This command is equivalent to the Peek button in the display for the variable with a zero period.

Port... Displays the contents of a selected port entry in the output window.

General Testbed Commands:

Create Slot... Creates a new slot on a given node with a given id (which must be unused). Defaults: slot id = 1, target node = 0.

Reload Reloads the configuration module specified on the command line. This is done in such a way that consistency is preserved as much as possible. Init functions are not reexecuted during the reload, though declaration functions are. Any reinitialization or new initialization which needs to be done should be performed by a conversion function. Those slots which have modules reloaded are suspended immediately before the reload and resumed at the end of the entire operation. This prevents actions from erroneously being invoked before they have been linked.

Unfortunately the application specific user interface state information can not be preserved. All application specific extensions are destroyed and recreated during the reload, as if the system were restarted from scratch.

At the end of the reload, each slot executes a user conversion function if one has been selected via the **Conversion** command. See the next section for details of how to write and use such functions.

Quit Exits the Testbed.

A.3.2 Support for Dynamic Modification

The **Create Slot** and **Reload** options in the general menu and **Load Module**, **Initialize**, **Poke** and **Conversion** options in the slot menu provide dynamic modification mechanisms. The **Load Module** and **Initialize** options load a single module, recompiling if necessary, and cause all modules to be relinked, invoking a function with the same name as the module in the case of **Initialize**. If an initialize command on the host slot results in modules being reloaded on other slots, declaration functions are reinvoked but not init functions. When declaration modules are reloaded, automatic state conversion is performed which attempts to preserve the state before the reload as much as possible and then the function specified using the conversion option (if present) is invoked. The **Reload** command reloads the configuration module. It is equivalent to **Initialize** on the host slot with the original configuration module name specified.

Variables which are redefined to have different types (either because their type is redefined or because they are given a completely different type) during the reload are reallocated and data is copied from the old to the new versions by matching fields with the same names in structures, corresponding elements in arrays and by

assignment for scalars and pointers. After the reload has completed any pointers to variables which have been redefined are adjusted to point to the corresponding part of the new version if possible. If an inconsistency is found which prevents this, then the reload is aborted leaving the system in the same state as before the reload. Note that pointers which persist across the reload are updated to point to the corresponding part of the new state. This includes port-action assignments. Any slot which is changed gets suspended for the duration of the reload and a synchronization is performed to ensure that any messages in transit between these slots have landed. At this point data in messages is converted and the user defined conversion function is invoked if specified.

User Conversion Functions

The user conversion function specified with the **Conversion** menu option may be any function which will be loaded on the slot at the end of the reload. Any references to state variables will be to the new versions, so the new versions of type definitions should be used when declaring such variables. References to the old versions of variables should be made by declaring pointers of the appropriate old type (which may need to be defined within the module with a different name for consistency with the new versions) and using the old_symbol_ptr function described below. If the module is to be loaded in advance of the reload and needs to refer to new variables which will not be defined until the reload occurs, these can be obtained using the symbol_ptr function also described below.

A.4 Structure of a Testbed Application

Apart from the makefile and the routing file, a Testbed application is bootstrapped entirely from init functions. The application code is divided into modules which contain actions and functions which may be called from these (or each other). These modules fall into the following categories, each of which is discussed in detail in following subsections:

1. Application modules. These are loaded onto non-host slots and may call any of the library functions which are defined for all slots.

2. Host modules. These may call additional library functions to perform such operations as loading modules onto other slots etc. Host modules may contain init functions, declaration functions, X windows callback functions and actions which respond to messages from the target system. See Sec-

tion A.4.6 on customizing the user interface for details on how to add application-specific host modules.

3. Init modules. These should contain a function which has the same name as the module and no parameters. This function is invoked once, immediately after being first loaded. It may call other functions in the init module and standard system functions, but should not call functions in other application modules as these are not linked until after the init function is invoked. The module containing the init remains loaded after the init function has been invoked and so may contain functions which are called by other modules. The purpose of init functions is to perform extra initialization apart from variable and type declarations and port definitions. The latter should be performed by declaration functions described below. The init function will not be reinvoked if the module is reloaded during a reload operation. This is because the initialization which it performs is likely to be inappropriate after the system has started. Any reinitialization or new initialization which is required should be performed by a conversion function.

4. Declaration modules. These are similar to init modules in that each contains a function with the same name as the module which is invoked at load time. Unlike init modules, if the module is reloaded, the declaration function is reinvoked. The declaration function may define types, define and initialize state variables and ports. Initialization of variables should be confined to that allowed by the declaration functions described in Section A.4.4 and no assignments to new variables should be made. If these are necessary then additional init modules should be supplied. Similarly declaration functions should not send initial messages. Declaration functions which setup ports should be loaded after the modules containing actions which are used in the port initializations unless the actions are contained in the same module. There should be at least one declaration function for each slot, which assigns actions to ports, otherwise no user actions may be scheduled at that slot. Declaration functions which define types are often separate from those which define variables and ports. They are generally loaded on all slots, or at least on both the slot which uses the type and the host slot which uses the type information for debugging purposes. If the functions contained in a declaration module need to access state variables defined within the module, they must use static pointer variables, global to the module. These should be set to the address returned by the storage allocation functions (or macros). Static variable declarations are not generally advisable in Testbed modules

as the values of such variables are not preserved across reload operations. However static pointers to state variables are safe in this context as they will be reassigned when the init is reinvoked.

5. The main application configuration module. This is a host init/declaration module which is selected on the command line and loaded and invoked during the host server initialization. Note unlike init functions for other slots, this may contain any declarations and it is reinvoked during a reload operation.

A.4.1 Message Format

Since messages are integral to Testbed applications, the format of messages and their interpretation will be described before the facilities of action functions which send and are invoked in response to these messages. Messages have the type testbed_msg, which has the following definition (in testbed.h):

```
typedef struct testbed_msg
{
  byte node,snode;
  short dest,source;
  u_short size,port;
  rtime_t timestamp;
  union {void *ptr; int val;} m;
} testbed_msg;
```

The timestamp is a real number measured in seconds since system boot time. It is used to delay the actioning of a message, as messages are delayed until after their timestamp. Typically this is set relative to the current time, obtained with the rtime function. The special time value T_INFINITY is defined as the greatest possible time value. If a message has no hard deadline, then a priority may be provided by setting the timestamp using the macro Pri, where Pri(0) is the highest priority. The message timestamp combined with parameters associated with the port determine the scheduling of the action which processes the message.

A.4.2 Actions and Modules

Several actions may be defined in a single source file. This may also contain definitions of functions which are called by the actions and are visible to other modules. Functions which are shared between modules should have unique names. If several functions share the same name then the choice of which one is used is

not defined. There should be no global variables declared outside functions nor static variables within them unless either the values of the statics do not need to be preserved across a reload operation or they will be reassigned during the reload (e.g., pointers to state variables defined in an init). All permanent state variables used by actions should be created using the mechanisms described below. There are two header files: action_lib.h, which should be included by all actions, and host.h, which should be included by host actions.

Actions are simple functions which are called by a special harness process. All actions are invoked with a single parameter which is a pointer to the message data. A prototype for an action would have the form:

```
void action(void *data);
```

Actions will generally be declared with an appropriate type for the data pointer. The values from the fields in the message header are obtainable using functions defined below.

A number of global variables are available to all actions. These are defined in the header file action_lib.h.

Modules are compiled, using the Meiko mcc compiler into object form, with the extension .x8. This is the form in which they are readable by the Testbed, however the Testbed will run make before loading them to ensure that the object file is up to date. The modules will be linked dynamically with those library functions and global symbols which are present in the symbol table at the time they are loaded. They should not contain global or static declarations, which will not work. All state data which persists between invocations of the action must be added to the state.

Macros Defined in action_lib.h

void Send(msg)

Sends the message structure of type testbed_msg pointed to by msg (equivalent to the function route_send).

void Send_Msg(dest,size,port,ptr)

Sends the data of pointed to by ptr to the slot dest with a timestamp of the current time. See also the function route_send_msg.

void Delay_Msg(dest,size,port,delay,ptr)

Sends the data pointed to by ptr to the slot dest with a timestamp of the current time plus delay.

void Display_Obj(size,type,name,ptr)

Displays the object pointed to by ptr using the given size name label and type information in the display window corresponding to that variable if there is one or the main output window otherwise. If size is greater than the size of type, then the object is assumed to be an array of this type. See also the function report_var. Example:

```
char *s = "hello";
Display_Object(strlen(s),"s",char,s);
```

Print_Var(type,var)

Prints the variable, array element or structure field var in the main output window assuming it has the given type. See also the function display_var. Example:

```
Print_Var(int,x);
```

void PrintS(s)

Prints the string s in the output window. Equivalent to the function prints.

void Def_Port(index,action,type,min,max,post,tpriority)

Interface to the def_port function (see below) which converts the action token into a string (prepending the required '_') as well as the type token. The index should be an integer in the range 0..MAX_USER_PORT. Min and max are integers giving the number of reserved schedule entries (buffers) for messages for this port and the maximum number of entries which will be queued for this port. The post parameter is an rtime_t value which may be either a time relative to message timestamp (values less than T_INFINITY), which becomes a deadline or an absolute priority indicated by values greater than T_INFINITY. The higher the value the lower the priority. The macro Pri can be used to generate priority values. It takes positive integer parameters with Pri(0) having maximal priority. The tpriority parameter is a transputer priority value at which the action function executes. It may take values HI_PRI or LO_PRI. Example:

```
Def_Port(MOTOR,motor,int,2,1,Pri(0),HI_PRI);
```

void Data_Port(index,action,type)

Initializes the index port with the named action, which must be in the symbol table, and specifies that messages sent to this port contain data of the given

type. The action may be NULL, in which case any message is processed as a no-op. Example:

```
Data_Port(MOTOR,motor,int);
```

void Port(index,action)

Same as Data_Port but does not set the type.

void Set_Port(index,action)

The macros Data_Port, Port, Def_Port and the general function def_port should be used in declaration modules to define ports, but the macro Set_Port should be used in actions as it is more efficient. It should be noted however that if the action function is declared as static then the port will not be correctly updated during a reload. For this reason all functions which may be used as actions should be global. Equivalent to the function change_port.

void Pri(value)

Returns a priority value, where Pri(0) is the highest priority and Pri(1) is lower etc.

A.4.3 User Defined Types

action_lib.h also contains the following macros for defining types. These may be in ordinary declaration modules but it should be noted that they need to be defined on all slots which make use of these type definitions, including the host slot if values with these types are to be displayed during debugging and monitoring. For this reason it is a good idea to put type definitions in a separate module. It should also be noted that the corresponding types still need to be defined using typedef for the compiler.

void Scalar(primitive,type)

Define a scalar called type which is equivalent to the standard type primitive. Example:

```
Scalar(int,counter);
```

void Pointer(type,base_type)

Define a pointer called type which points to the (already defined) type base_type. Note that when a variable is declared as a pointer (of any order) to a predefined type, the corresponding pointer type(s) are all defined. For example `Var(char ***,fred)` causes the pointer types char*, char**

and char*** to be added to the type table. Similarly if a type is declared based on[4] a pointer which is not defined then it is defined. Example:

```
Pointer(intptr,int);
```

void ArrayType(type,count,base_type)

Define type to be an array of count elements of type base_type. As with pointers, a variable may be declared directly as an array (using macros described in the next section). In this case a type called base_type[n] is created, where base_type is the given base type of the array and n is its dimension. Example:

```
ArrayType(line,256,char);
```

In addition to these macros structure types can be defined using the function define_structure, described in the next section.

A.4.4 Macros for Defining State Variables

The following macros are for defining state variables. The macros Var and Var_I are intended for defining static variables which may be accessed as global variables from actions. These macros are normally used in declaration functions. The macros New and ByteArray are for defining dynamic variables and may be used at any time. It should be noted that unless pointers to dynamic variables exist in the state, they will not persist across a reload operation, even if there are static pointers defined in modules which point to them.

void Var(type,name)

This defines a new state variable called name. It is equivalent to a declaration of the form: type name;[5] in a conventional C program.

void Var_I(type,name,val)

This defines a new state variable called name and initializes it with the value val. It is the equivalent of a declaration of the form: type name = val; in a conventional C program.

void Array(type,name,count)

Similar to Var, in fact if count is a constant, then it is equivalent to Var(type[count],name). It is equivalent to a declaration of the form: type name[count]; in a conventional C program.

[4] e.g. an array of base type char * or a structure with a pointer field

[5] Application modules should not contain global declarations of this form.

void Array_I(type,name,count,val)

Similar to Var_I except that val should be an array of initial values.

void *New(type)

For defining dynamic variables. This is the equivalent of the function call malloc(sizeof(type)) in a conventional C program.

void *NewArray(type,count)

For defining dynamic array variables. This is the equivalent of the function call calloc(count,sizeof(type)) in a conventional C program.

void *ByteArray(size)

Useful for defining temporary objects whose exact size is unknown at compile time. Care should be taken when using this function to define persistent data as the monitoring and dynamic modification mechanisms in Testbed have no information about its structure.

In addition to these there is the macro:

void Free(var)

For releasing storage allocated to dynamic variables. This is the equivalent of the function call free(var) in a conventional C program. Objects freed using Free are available for reuse by any of the storage allocation functions.

A.4.5 Static Variables and Functions

Variables may be declared within modules (either at the top level or within nested blocks) with the storage class static. Such variables will be allocated and initialized as they would be in a conventional program, however they are not part of the slot's state as far as Testbed is concerned and their values will not be preserved during a reload operation. As described in the section on init modules, it is sometimes necessary to use static variables, but the programmer should exercise caution. Global variable declarations (non-static variables declared at the top level in a module) are currently ignored by the Testbed loader and will result in an undefined variable error at the link stage. All global variables should be defined using the functions or macros provided rather than by declaring them at the top level.

Static functions are also allowed and functions which are private to a module should be declared as such. However such functions should not appear as arguments to Set_Port as they do not get included in the Testbed symbol table and so any port which is assigned to a static function will not be updated during a reload. For efficiency reasons the change_port function does not check for the validity of

the action argument. Similarly no state pointers should be set to static functions or static variables.

A.4.6 Configuration

Customizing the User Interface

As described above, there is a box (actually an Athena box widget) reserved for application specific controls in the main root window (called app_box) and in each slot window (called app_slotbox[slot_id]). These may be created and manipulated using the Xlib, Xt and Xaw functions which are available (see Section A.5). When a reload operation is performed these boxes are destroyed (using XtDestroyWidget) and recreated. Unfortunately this results in the loss of any state information contained in them, but it is necessary to avoid ending up with multiple versions of controls only the most recent of which are valid.

Macros for Host Actions

The header file host.h defines the following macros for use in host actions:

void Setup_Slot(id,node,label,stacksize,npools,pool,nmsgpools,msgpool)
Creates a slot with given storage configuration. See the create_slot function.

void Slot(id,node,label)
Creates the slot with given id (in the range 1–255) on given node if it doesn't already exist. The label parameter is used to label the application control box. This version causes the slot to be created with reasonable default values for the stack and global stores.

void Module(target_slot,name) Uses the given name to derive the name of the object module by adding the extension .x8 and loads it onto the target_slot or reloads, if necessary. Any previous module will be deleted.

void Global_Module(name)
As with Module except that the module is loaded onto all slots.

void Init_Slot(target_slot,name)
Downloads and invokes an init module. The module should contain a function called name which takes no parameters. There should be no references to functions in other modules in the init.

void Global_Init(name)

Similar to Init_Slot except that the init module is loaded and invoked on all slots.

void Declare(target_slot,name)

Downloads and invokes a declaration module. The module should contain a function called name which takes no parameters. There should be no references to functions in other modules in the module.

void Global_Declare(name)

Similar to Declare except that the init module is loaded and invoked on all slots.

void HModule(name)

Like Module, but the slot is the host. Useful for loading modules which are invoked in response to messages from the Testbed.

void Prompt(s)

Prints the string s on the host command window and flushes the output.

A.5 Detailed Description of Functions

A.5.1 Library Functions Available to all Slots

Along with the standard C library functions sprintf, atoi, atof, strlen, strcmp, strcpy, fabs and memcpy, the following functions (declared in action_lib.h) are available from all actions.

int define_scalar(int std_type,char *primitive,char *name);

Defines a scalar type which is equivalent to the existing type primitive. Returns the index of the type in the type table. In general the macro Scalar should be used instead of define_scalar.

int define_pointer(int std_type,char *name,int base_type);

Defines a pointer type which points to the type with index base_type. Returns the index of the type in the type table. In general the Pointer macro should be used instead of define_pointer.

int define_structure(int std_type,char *name,int nfields,...);

Defines a structure. The variable part of the parameter list is a series of type, field-name pairs as with init_type. There is no macro form as macros

may not have a variable number of parameters. The std_type field should be FALSE for user types. Returns the index of the type in the type table.

int define_array(int std_type,char *name,int count,int base_type);

Defines an array type of count elements of type with index base_type. Returns the index of the type in the type table. In general the macro ArrayType should be used instead of define_array.

void def_port(int index,char *name,char *type,int min, int max,rtime_t post,int tpriority);

Initializes the port table element index with the action entry point found by looking up _name in the symbol table and associated type. If the port was previously defined, then the initial action assignment is ignored. The type field is used for debugging purposes. The post time is added to the message timestamp and used as a deadline indication to the scheduler if less than the special T_INFINITY value and a priority otherwise. The priority is the transputer priority level (LO_PRI or HI_PRI) at which the action will run. HI_PRI effectively disables interrupts except while sending messages, ensuring that system activities and the activity of other slots on the node cannot delay an action which does not communicate. Such actions also get slightly better performance from system functions such as reading the time and sending messages and the scheduling overhead is slightly lower between successive HI_PRI actions. The cost is that performance of other slots and through routing activities may be impaired. A slot executing a LO_PRI action will be preempted if another slot becomes schedulable with a message whose deadline (priority) is earlier (higher) than that of the current slot's message. They may also be interrupted and delayed at arbitrary points by the system while it receives or through routes messages, or schedules messages when timeouts occur. If messages for this port need not be boxed, have no deadline, are of standard (minimum) priority and do not need to be processed with interrupts disabled then either the Data_Port or Port macros may be used depending on whether the messages carry data.

Min and max are integers giving the number of reserved schedule entries (buffers) for messages for this port and the maximum number of entries which will be queued. A max value of 0 indicates that there is no maximum. If a message arrives at a slot and there is no free storage left for messages, then it will be discarded. Setting a min value greater than 0 ensures that not all messages of a particular type will be lost. When the number of queued messages reaches max, data from new messages is used to update

the data part of the most recent previous message. It should be noted that min includes an entry which is currently being processed, but max does not, thus to ensure that only superseded messages are lost, min should be at least 2. If the size of the data in a new message does not match the size of the type and a reserved schedule entry is being used, then the data part of the schedule entry is reallocated from the free store for messages if possible. If there is insufficient free store for this then the message is discarded. See the Setup_Slot macro above for how to specify the amount of storage for messages.

void def_var(char *name,int size, int type,unknown val);

If the variable _name does not exist it is created and initialized with the value val. If name is an array then val is taken to be a pointer to an array of initial values. If the variable is already defined then the entry in the symbol table for it has its name set to NULL, while the new definition goes into symbol_table and is initialized from the old one by automatic conversion as far as possible. Conversion functions may be applied later, if necessary to complete the conversion. In general one of the macros Array, Array_I, Var or Var_I (defined in init_action.h) should be used instead of def_var.

void route_send(testbed_msg *msg);

Sends the message pointed to by msg. The Send macro is equivalent.

void route_send_msg(int dest,int size,int port,rtime_t timestamp, char *data);

Similar to route_send except that the message structure is filled in by the function. The timestamp is an absolute time in seconds since system startup and acts as both a timeout for scheduling future actions and a reference point for the post time specified at the destination port. The simpler Send_Msg and Delay_Msg macros may be used in place of the route_send_msg function as the timestamp will normally be the current time or an offset from it.

void display_var(char *type,byte *data);

Display data using the given type, which should be defined on the host server, in the main window for the slot. The macro Print_Var may be used instead to print the value of a variable or field.

void report_var(int dest,int port,int size,char *type,char *name, char *data);

Report data using the given type, which should be defined on the dest slot. The report will be sent to the given port at the dest slot which should have been initialized with an appropriate action to process the reported data. The

macro Display_Obj may be used instead to display the data in the window for that variable.

void prints(char *str);

Simple function for outputing a string in the slot's output window. Can be combined with sprintf to achieve the effect of printf. The PrintS macro is equivalent.

void delete_var(void *var);

Removes the variable with address var from the symbol table and adds the storage block to the free. There should be no references to this object which are used after deletion. This is not checked. The macro Free is equivalent.

void *symbol_ptr(char *var);

Returns a pointer to a state variable called var, or NULL if not defined. The name should be preceded by '_' for C variables. This function is intended to be used in user conversion functions for accessing new variables which are not present when the conversion function is loaded.

void *old_symbol_ptr(char *var);

Returns a pointer to the old version a state variable called var, or NULL if not defined. The name should be preceded by '_' for C variables. This function is intended to be used in user conversion functions for accessing the old version of a variable which has been redefined. If no old version has been defined then the current version is returned. Note that the old versions are discarded at the end of a reload operation, after the user conversion function has been invoked.

rtime_t rtime();

Returns the current time in seconds since system startup. The time values returned on the local node should be accurate to approximately $1\mu s$, however clocks are currently only synchronized between nodes to approximately $10\mu s$ accuracy. It should also be noted that there is an overhead associated with reading the clock, which is greater for LO_PRI actions because it requires a priority switch in order to read the high priority transputer timer.

void configure_device(int device_id,int size,int dest,int port);

Can be invoked on any slot to configure a device to input messages of the given size and pass them to the given dest slot and port. The slot need not be on the same node as the device. The messages have the device_id in their source field and are timestamped with the time that they are read from the

link. The same device may be configured more than once but before the first configuration no input will be read from the device.

void read_device(int device_id,int size,int port);

Perform a one off read operation on the device. The data is sent to the given port on the slot which performed the read.

A.5.2 Host Library Functions

In addition to the functions available to all modules the standard functions fprintf and fscanf along with additional standard X library and Testbed functions are available to application code loaded by the host server slot.

Testbed Functions

void print_error(int errcode,int source,char *msg);

Prints an error message appropriate to the given errcode.

void r_display_var(TextWidget ctx,int size,char *type,byte *data);

Displays the data in the given text widget.

char *load_object(char *objname,int *totalsize);

Loads the object file objname, returning a pointer to the result. This may be sent to the LOADM_PORT or LOADI_PORT at a remote slot or linked into the host server slot.

int def_module(int target_slot,int module_type,char *name);

Loads a module from the object file <name>.x8 after performing a make on it and loads it onto the given target_slot. The module_type parameter may be MODULE, INIT_MODULE or DEC_MODULE. The macros Module and Global_Module provide a more user friendly interface.

void global_module(int init,char *name);

Similar to def_module except that the module is loaded onto all slots. The macros Global_Module and Global_Init are more user friendly.

void create_slot(int new_slot_id, int node, char *label, int stacksize, int npools, pool_spec *pool, int nmsgpools, pool_spec *msgpool, int suspend);

Creates a new slot with given id (in the range 1–255) on given node if it doesn't already exist. The id must not have been allocated to a device during the network configuration. The label parameter labels the window opened for this slot on the display. The integer stacksize is the amount of stack (in

bytes) allocated for action function calls. The integer npools is the number of size categories into which the slot's global data store is divided. The pool parameter is an array of type pool_spec, which is defined in testbed.h as:

```
typedef struct
{
    int size,count;
} pool_spec;
```

Each element specifies a pool of count chunks of store at the given size. The nmsgpools and msgpool parameters specify the separate store which is used for storing incoming messages. Note that each message waiting to be processed by a slot there is a schedule entry and a separate data object if the data size is greater than SCALAR_MSG_SIZE. The storage for both parts comes from the msgpool, so this storage should include enough objects large enough for schedule entries (which have size 72 bytes) as well as for the data parts of non-scalar messages. Note that schedule entries including the data area can be reserved for messages addressed to a given port using the min parameter of the Def_Port macro or def_port function. The suspend flag indicates whether the newly created slot should initially be suspended. It is normally set to TRUE, and the macro Setup_Slot defaults to this.

void create_std_slot(int new_slot_id,int node,char *label,int suspend);

Interface to create_slot for creating a new slot with standard stack and store parameters. The macro Slot defaults the value of suspend to TRUE.

X Library Functions and Widget Classes Available

The following X library functions and widget classes are currently available. Include the header file xcode.h to get the appropriate definitions.

XEventsQueued
XFlush
XGetWindowAttributes
XSync
XawDialogAddButton
XawDialogGetValueString
XawScrollbarSetThumb
XawToggleGetCurrent
XtAddCallback
XtAppAddActions

XtAppCreateShell
XtAppNextEvent
XtAppPending
XtAppProcessEvent
XtCallbackExclusive
XtCallbackNone
XtCallbackNonexclusive
XtCallbackPopdown
XtCreateApplicationContext
XtCreateManagedWidget
XtCreatePopupShell
XtDestroyApplicationContext
XtDestroyWidget
XtDispatchEvent
XtDisplay
XtGetValues
XtName
XtNameToWidget
XtOpenDisplay
XtOverrideTranslations
XtParent
XtParseTranslationTable
XtPopdown
XtPopup
XtRealizeWidget
XtRemoveCallback
XtSetSensitive
XtSetValues
XtUnmanageChild
XtUnrealizeWidget
XtVaCreateManagedWidget
XtVaGetValues
XtVaSetValues
XtWindow applicationShellWidgetClass
transientShellWidgetClass
topLevelShellWidgetClass
panedWidgetClass
asciiTextWidgetClass

menuButtonWidgetClass
simpleMenuWidgetClass
labelWidgetClass
boxWidgetClass
commandWidgetClass
toggleWidgetClass
formWidgetClass
dialogWidgetClass
smeBSBObjectClass

Appendix B

Testbed System Ports

B.1 Centre Ports

SYS_REPORT_CLK For non-root nodes causes the current local time to be returned to the root centre. For the root node centre, causes the offset of the sending nodes clock from the root one to be calculated and a SYS_REPORT_CLK message sent to the next node in sequence. When all nodes have reported, the offsets are sent out using SYS_CLK_SYNC messages.

SYS_CLK_SYNC For non-root nodes causes the clock to be incremented by an offset given in the data field. For the root node centre, this message is used to register that the sending node is ready to be synchronized. When such messages have been received from all the other nodes a SYS_REPORT_CLK message is sent to the first.

SYS_CREATES A message from the host server or from a slot migrating from another node asking for a new slot entry to be created on the current node. The message data field contains the slot id.

SYS_DELETES A message from the slot indicating that it is ready to die.

SYS_REROUTE Update information for the routing tables giving the node location of a new slot.

SYS_MIGRATE Depending on whether it contains rerouting information this is either a message from a migrating slot requiring that the node be suspended, or an acknowledgement of suspension from a local slot.

SYS_RESUME A message from a migrated slot allowing slots on the node to resume.

SYS_ABORT Causes the centre to suspend each slot and await SYS_ABORT_ACK acknowledgements.

SYS_ABORT_ACK Once one of these is received from each slot, a message is sent on a special channel to the original startup process which then calls exit(0).

SYS_FLUSH_FORWARD At the end of a migration, the slot asks the centre at its destination node for any messages which have been waiting to be forwarded. These are inserted atomically into the slot's schedule by the centre, without allowing any other messages to arrive in the mean time and then the location table is updated to reflect the fact that the slot is local. This ensures that messages do not arrive out of order.

CENTRE_FORWARD In one of the migration options not all of the nodes are notified of the new location of the slot. In this case messages which arrive at the old node are forwarded by the centre.

MIGRATE_ACK When a message arrives for a slot which has migrated away, the centre sends a SYS_MIGRATE message to the centre on the node from which the message originated. This is acknowledged and subsequently the true destination node is requested to resume the suspended node and update its routing tables. This protocol ensures that messages do not get out of order.

B.2 Host Server Ports

H_PRINTS Prints a string in the output window corresponding to the source slot.

H_DISPLAY_VAR Displays the value of a variable.

MIN_RET_PORT..MAX_RET_PORT Reserved for state variable reports.

H_DEF_VAR Causes a variable to be added to the Variables menu for the source slot.

H_REPORT_LOG Displays a logged event or variable.

INIT Reserved for the host server initialization action contained in the special init module xhost.c.

SYS_ERR Displays an error message.

H_FLUSH Flushes any output waiting to be sent to the display.

XUSER Invokes the polling action which processes outstanding X windows messages from the display server.

B.3 Application Slot Ports

The following ports are not defined for system slots such as the host server or any other centre.

MIGRATE Initiate the migration of the slot. Causes the slot to broadcast a SYS_MIGRATE message to all centre slots.

ALT_MIGRATE Initiate the asynchronous version of migration in which SYS_MIGRATE messages are sent to the centre on the source and destination node only.

ADD_STATE Add a variable during migration.

ADD_TYPE Add a user defined type during migration.

LOADM Load an ordinary module. Used during initialization, dynamic updates and migration.

ADD_PORT Update the user port descriptors during migration.

PORT_ACK Acknowledge the receipt of the user port table to the migrating slot, indicating that it is safe to forward messages.

ADD_PEEK Add a periodic peek request which was being performed on a migrating slot.

MIGRATE_DONE Indicates that all messages have been forwarded and it is safe to flush any messages which the centre was waiting to forward.

FORWARD_RESUME If this message has a node id as data then a SYS_RESUME message is sent to the centre on that node, otherwise the SYS_RESUME is broadcast to all centres.

DEF_CONVERT Define the user conversion function invoked after all automatic conversions during a reload operation.

B.4 Common Ports

NULL_PORT No-op.

TERMINATE Special port code recognized by the executive kernel thread to indicate the termination of an action.

PREEMPT Special port code recognized by the executive indicating that the current action may safely be preempted.

ABORT Special port code recognized by the executive causing it to terminate. Once all executive threads have terminated control is returned to the shell on the Unix host.

SUSPEND_SLOT Cause the slot to suspend processing of user messages.

RESUME_SLOT Cause the slot to either resume processing of user messages immediately (if the message contains no data) or initiate the quiescence protocol setting the group_suspend_total to the supplied value and broadcasting GROUP_SUSPEND.

GROUP_SUSPEND If the data value indicates a reload failure then abort the reload, otherwise increment the group_suspend_count and if equal to group_suspend_total, complete the conversion including converting the data in any waiting messages before broadcasting a GROUP_RESUME message.

GROUP_RESUME Increment the group_resume_count and if equal to group_resume_total, reset group_resume_count and group_suspend_count and resume processing user messages.

LINK Relink all modules in the slot.

MONITOR_REQUEST Request monitoring of some subevent.

ABORT_SLOT Suspend the slot and acknowledge to the centre, indicating that it is safe to send an ABORT message to the executive.

PEEK Request a one-off or periodic peek of a variable value.

POKE Poke the value of a variable.

INITM Load an initialization module and invoke the contained init function.

MIGRATE_ACK For application slots: if migrate_ack_count has reached migrate_total transfer the state of the slot to the new node; for centre slots: forward any messages for the slot followed by a FORWARD_RESUME message.

SHOW_PORT Return the contents of a port entry.

Appendix C

Source Code for ROV

The source code in this appendix is for the test version of the ROV submerssible as described in this thesis, not including the simulation of the cable between the vehicle and the surface controller. See Appendix A for a full description of the format of routing files, and all macros and non-standard library functions used.

C.1 Routing File

```
4
0 0 0 0
1 1 1 0
2 2 2 0
3 0 0 3
r 2 3 4 5
```

C.2 C Definitions

```
/* rov.h - definitions for ROV autodepth controller. */

/* Slot ids: */

#define HIGH_CONTROL    1
#define LOW_CONTROL     2
#define FEEDBACK        3
#define ROV             6

/* Device ids: */

#define ROV_DEV             4
#define CONTROL_DEV         5
```

```
/* Port ids: */

/* HOST: */

#define DISPLAY_STATUS  0

/* HIGH_CONTROL: */

#define H_VELOCITY      0
#define V_VELOCITY      1
#define AUTODEPTH       2
#define HEADING         3
#define AUTOHEADING     4
#define DEPTH_REPORT    5
#define HEADING_REPORT  6

/* LOW_CONTROL: */

#define UPDATE_MOTORS   0
#define SWITCH_ON       1
#define SWITCH_OFF      2
#define POLL_DEPTH      3
#define STATUS          4

/* ROV: */

#define ROV_MESSAGE     0

/* Other constants: */

#define NMOTORS 4
#define ON  1
#define OFF 0

/* Motor indicies: */

#define LEFT_SIDE  0
#define RIGHT_SIDE 1
#define LEFT_REAR  2
#define RIGHT_REAR 3

/* Bits in the low order nyble of a device message byte select the
   motor, while the high order nyble indicate what type of device
   message: */
```

```
#define ROV_SELECT_MASK 0xF0
#define ROV_DATA_MASK   0x0F
#define MOTOR_OFF     0x00
#define MOTOR_ON      0x10
#define MOTOR_PLUS    0x20
#define MOTOR_MINUS   0x30
#define DEPTH_POLL    0x40
#define COMPASS_POLL 0x50

typedef struct
{
  int right,forward;
} horizontal_velocity;

typedef struct
{
  int right,forward,down;
} velocity_vector;

typedef struct
{
  byte speed;
  s_byte direction;
} motor_info;

typedef struct
{
  byte state;
  s_byte direction;
} motor_status;

typedef struct
{
  int depth,compass;
} rov_status;
```

C.3 C Source Code

```
/* rov.c - special module containing the code for loading the
   application system and also special user interface extensions.
   */

#include <action_lib.h>
#include <host.h>
#include <xcode.h>
```

```
#include "rov.h"

#define DRIVE_FORWARD  0
#define DRIVE_SIDEWAYS 1

/* Toggle the ADC on or off. */

static void adc_switch(Widget button)
{
  Boolean adc_on;
  byte adc_mode;

  XtVaGetValues(button,XtNstate,&adc_on,NULL);
  if (adc_on)
    XtVaSetValues(button,XtNlabel,"MANUAL DEPTH",NULL);
  else
    XtVaSetValues(button,XtNlabel,"AUTO DEPTH",NULL);
  adc_mode = adc_on;
  Send_Msg(HIGH_CONTROL,1,AUTODEPTH,&adc_mode);
}

/* Toggle the AHC on or off. */

static void ahc_switch(Widget button)
{
  Boolean ahc_on;
  byte ahc_mode;

  XtVaGetValues(button,XtNstate,&ahc_on,NULL);
  if (ahc_on)
    XtVaSetValues(button,XtNlabel,"MANUAL HEADING",NULL);
  else
    XtVaSetValues(button,XtNlabel,"AUTO HEADING",NULL);
  ahc_mode = ahc_on;
  Send_Msg(HIGH_CONTROL,1,AUTOHEADING,&ahc_mode);
}

/* Send message to ROV, commanding it to go up. */

static void rov_up()
{
  static int depressed = FALSE;
  int motor_speed;

  if (depressed)
    motor_speed = 0;
```

```
  else
    motor_speed = -100;
  Send_Msg(HIGH_CONTROL,sizeof(motor_speed),V_VELOCITY,
           &motor_speed);
  depressed = !depressed;
}

/* Send message to ROV, commanding it to go down. */

static void rov_down()
{
  static int depressed = FALSE;
  int motor_speed;

  if (depressed)
    motor_speed = 0;
  else
    motor_speed = 100;
  Send_Msg(HIGH_CONTROL,sizeof(motor_speed),V_VELOCITY,
           &motor_speed);
  depressed = !depressed;
}

static Widget rov_panel;

/* Actions which display the latest depth and heading from the ROV
   in a special panel. */

void display_status(rov_status *status)
{
  char display[21];

  sprintf(display,"DPTH:%6d HDNG:%3d",status->depth,
          status->compass);
  XtVaSetValues(rov_panel,XtNlabel,display,NULL);
  XFlush(thedisplay);
}

/* The turn control is a scroll bar which snaps back to the zero
   position when released.  Only the middle button has any effect.
   On the real ROV control panel it is a knob which has this
   behaviour. */

void snap_back(Widget turnscroll)
{
  int differential = 0;
```

```
  XawScrollbarSetThumb(turnscroll,0.45,0.1);
  Send_Msg(HIGH_CONTROL,sizeof(int),HEADING,&differential);
}

float adjust_thumb(float pos)
{
  pos -= 0.05; /* The middle of the thumb tracks the pointer. */
  /* Keep the whole thumb in view: */
  if (pos < 0)
    pos = 0;
  else if (pos > 0.9)
    pos = 0.9;
  return(pos);
}

void jump_turn(Widget turnscroll,XtPointer client_data,
               XtPointer call_data)
{
  int differential;
  float pos = *(float *)call_data;

  differential = 100 - 200 * pos;
  pos = adjust_thumb(pos);
  XawScrollbarSetThumb(turnscroll,pos,0.1);
  Send_Msg(HIGH_CONTROL,sizeof(int),HEADING,&differential);
}

Widget create_scroll(Widget box,XtArgVal orientation,
                     XtArgVal length,float topofthumb,float shown)
{
  Arg args[4];

  XtSetArg(args[0],XtNorientation,orientation);
  XtSetArg(args[1],XtNlength,length);
  if (sizeof(float) > sizeof(XtArgVal))
    {
      /* If a float is larger than an XtArgVal then pass these
      resources by reference. */
      XtSetArg(args[2],XtNtopOfThumb,&topofthumb);
      XtSetArg(args[3],XtNshown,&shown);
    }
  else
    {
      /* Convince C not to perform an automatic conversion. */
      XtArgVal *l_top = (XtArgVal *)&topofthumb,
```

```
                *l_shown = (XtArgVal *)&shown;

        XtSetArg(args[2],XtNtopOfThumb,*l_top);
        XtSetArg(args[3],XtNshown,*l_shown);
      }
  return(XtCreateManagedWidget("scroll",scrollbarWidgetClass,box,
                               args,FOUR));
}

void setup_turn_control()
{
  Widget turnbox,turnscroll;
  static XtActionsRec turn_actions[] = {{"snap_back",snap_back}};

  XtAppAddActions(app_con,turn_actions,1);
  turnbox = XtCreateManagedWidget("turnbox",boxWidgetClass,
                app_slotbox[HIGH_CONTROL],NULL,ZERO);
  XtCreateManagedWidget("Turn Control:",labelWidgetClass,turnbox,
                        NULL,ZERO);
  turnscroll = create_scroll(turnbox,XtorientHorizontal,100,0.45,
                             0.1);
  XtAddCallback(turnscroll,XtNjumpProc,jump_turn,NULL);
  XtOverrideTranslations(turnscroll,
            XtParseTranslationTable(
"<BtnUp>: NotifyScroll(Proportional) EndScroll() snap_back()"));
}

/* Two scroll bars control the horizontal movement of the ROV.
   The real ROV control panel has a joystick for this. */

static horizontal_velocity horiz_v = {0,0};

void jump_horizontal(Widget scroll,XtPointer client_data,
                     XtPointer call_data)
{
  float pos = *(float *)call_data;
  int direction = (int)client_data;

  if (direction == DRIVE_FORWARD)
    horiz_v.forward = 100 - 200 * pos;
  else
    horiz_v.right = 200 * pos - 100;
  pos = adjust_thumb(pos);
  XawScrollbarSetThumb(scroll,pos,0.1);
  Send_Msg(HIGH_CONTROL,sizeof(horiz_v),H_VELOCITY,&horiz_v);
}
```

```
void setup_drive_control()
{
  Widget drivebox,drivescroll;

  drivebox = XtCreateManagedWidget("drivebox",boxWidgetClass,
                  app_slotbox[HIGH_CONTROL],NULL,ZERO);
  XtCreateManagedWidget("Drive Control:",labelWidgetClass,
                        drivebox,NULL,ZERO);
  drivescroll = create_scroll(drivebox,XtorientVertical,100,0.45,
                              0.1);
  XtAddCallback(drivescroll,XtNjumpProc,jump_horizontal,
        (XtPointer)DRIVE_FORWARD);
  drivescroll = create_scroll(drivebox,XtorientHorizontal,100,
                              0.45,0.1);
  XtAddCallback(drivescroll,XtNjumpProc,jump_horizontal,
        (XtPointer)DRIVE_SIDEWAYS);
}

static char *motor_name[NMOTORS] = {"Left Side","Right Side",
                                    "Left Rear","Right Rear"};

void rov()
{
  Widget control,button;
  int motor;

  Slot(HIGH_CONTROL,1,"HIGH_CONTROL");
  Slot(LOW_CONTROL,2,"LOW_CONTROL");
  if (FEEDBACK != LOW_CONTROL)
    Slot(FEEDBACK,2,"FEEDBACK");
  Slot(ROV,3,"ROV");

  /* Global type definitions: */

  Global_Declare(rov_types);

  /* Setup high level control, selecting speed for motors based on
     user input or autodepth/heading: */

  Module(HIGH_CONTROL,high_control);
  Declare(HIGH_CONTROL,high_decs);

  /* Setup low level control: */

  Module(LOW_CONTROL,motor_control);
```

```
  Declare(LOW_CONTROL,motor_decs);
  Init_Slot(LOW_CONTROL,motor_init);
  Module(FEEDBACK,poll);
  Declare(FEEDBACK,poll_decs);
  Init_Slot(FEEDBACK,poll_init);

  /* Setup simulated ROV: */

  Module(ROV,rov_actions);
  Declare(ROV,rov_decs);
  Init_Slot(ROV,rov_init);

  /* A control panel containing up, down, autodepth and
     autoheading buttons is added to the application box in the
     HIGH_CONTROL slot output window. */

  control = app_slotbox[HIGH_CONTROL];

  /* Pressing either of the up or down buttons (by pointing to the
     button and pressing mouse button 1) causes the ROV to drive
     in that direction.  Releasing the button (by releasing the
     mouse button) causes it to stop. */

  button = XtCreateManagedWidget("UP",toggleWidgetClass,control,
                                 NULL,ZERO);
  XtAddCallback(button,XtNcallback,rov_up,NULL);
  XtOverrideTranslations(button,
            XtParseTranslationTable(
"<Btn1Down>: toggle() notify()\n <Btn1Up>: toggle() notify()"));
  button = XtVaCreateManagedWidget("DOWN",toggleWidgetClass,
                                   control,XtNfromVert,button,
                                   XtNradioGroup,button,NULL);
  XtAddCallback(button,XtNcallback,rov_down,NULL);
  XtOverrideTranslations(button,
            XtParseTranslationTable(
"<Btn1Down>: toggle() notify()\n <Btn1Up>: toggle() notify()"));

  button = XtCreateManagedWidget("AUTO DEPTH",toggleWidgetClass,
                                 control,NULL,ZERO);
  XtAddCallback(button,XtNcallback,adc_switch,NULL);

  button = XtCreateManagedWidget("AUTO HEADING",toggleWidgetClass,
                                 control,NULL,ZERO);
  XtAddCallback(button,XtNcallback,ahc_switch,NULL);

  setup_turn_control();
```

```
  setup_drive_control();

  /* Panel for showing the latest depth and heading. */

  rov_panel = XtCreateManagedWidget("                    ",
                    labelWidgetClass,
                    control,NULL,ZERO);

  /* Actions which update the panel. */

  Def_Port(DISPLAY_STATUS,display_status,rov_status,2,1,LO_PRI);
}

/* rov_types.c - User defined types for the rov application. */

#include <action_lib.h>

void rov_types()
{
  define_structure(FALSE,"horizontal_velocity",2,"int","right",
           "int","forward");
  define_structure(FALSE,"velocity_vector",3,"int","right",
                   "int","forward","int","down");
  define_structure(FALSE,"motor_info",2,"byte","speed",
                   "s_byte","direction");
  define_structure(FALSE,"motor_status",2,"byte","state",
                   "s_byte","direction");
  define_structure(FALSE,"rov_status",2,"int","depth",
                   "int","compass");
}

/* high_control.c - Actions for the HIGH_CONTROL slot.  These are
   responsible for all the high level control of the ROV,
   including autodepth, autoheading and maintaining safety
   conditions. */

#include <math.h>
#include <action_lib.h>
#include "rov.h"

#define MAX_MOTOR_CHANGE 20
#define MOTOR_SAFETY_LIMIT 180

extern int target_depth;
extern int target_heading;
extern int autodepth_tolerance;
extern int autoheading_tolerance;
```

```
extern motor_info motor_settings[NMOTORS];
extern s_byte motor_target[NMOTORS];
extern velocity_vector velocity;
extern int manual_v;
extern int manual_diff;
extern int rear_differential;
extern byte autodepth_mode;
extern byte autoheading_mode;
extern int autodepth_factor;
extern int autoheading_factor;
extern rov_status status;
extern int report_settings;

/* Utility functions: */

static void compute_new_targets()
{
  int motor;

  /* We could do bounds checking on the velocities, but we just
     assume that the interface restricts them to the feasible
     region. */

  motor_target[LEFT_SIDE] = (velocity.down + velocity.right)/2;
  motor_target[RIGHT_SIDE] = (velocity.down - velocity.right)/2;
  motor_target[LEFT_REAR] = (velocity.forward
                             - rear_differential)/2;
  motor_target[RIGHT_REAR] = (velocity.forward
                              + rear_differential)/2;

  /* Safety condition: the sum of the motor speeds on either side
     must be within a limit. */

  while (abs(motor_target[LEFT_SIDE])
         + abs(motor_target[LEFT_REAR]) > MOTOR_SAFETY_LIMIT ||
         abs(motor_target[RIGHT_SIDE])
         + abs(motor_target[RIGHT_REAR]) > MOTOR_SAFETY_LIMIT)
    {
      /* Reduce all motor speeds by 75% until within bounds. */
      for (motor = 0; motor < NMOTORS; motor++)
        motor_target[motor] *= 0.75;
      PrintS("SAFETY: motors cut to 75%\n");
    }
}

static int adjust_motor_speeds()
```

```
{
  int motor,changed = FALSE;
  s_byte motor_state,new_state,delta;
  motor_info settings;

  for (motor = 0; motor < NMOTORS; motor++)
    {
      motor_state = motor_settings[motor].speed
    * motor_settings[motor].direction;
      delta = motor_target[motor] - motor_state;

      /* Motor speed change is limited.  If the change exceeds
         this limit, it will need to happen in several steps. */

      if (delta > MAX_MOTOR_CHANGE)
        delta = MAX_MOTOR_CHANGE;
      else if (delta < -MAX_MOTOR_CHANGE)
        delta = -MAX_MOTOR_CHANGE;

      new_state = motor_state + delta;
      if (new_state >= 0)
        {
          settings.speed = new_state;
          settings.direction = 1;
        }
      else
        {
          settings.speed = -new_state;
          settings.direction = -1;
        }

      /* The motor must stop for one cycle in order to change
         direction. */

      if (motor_settings[motor].speed > 0 &&
          settings.direction != motor_settings[motor].direction)
        settings.speed = 0;
      motor_settings[motor] = settings;
    }
}

static void adjust_motors()
{
  compute_new_targets();
  adjust_motor_speeds();
  Send_Msg(LOW_CONTROL,NMOTORS * sizeof(motor_info),UPDATE_MOTORS,
```

```
            motor_settings);
  if (report_settings)
    {
      name_string buf;

      sprintf(buf,"%g: motor_settings = ",rtime());
      PrintS(buf);
      Print_Var(motor_info[4],motor_settings);
    }
}

#define Bound(v,b) (v < -b ? -b : v > b ? b : v)

/* Compute vertical motor setting required to make up for depth
   discrepancy. */

static int auto_vertical(int delta)
{
  if (abs(delta) > autodepth_tolerance)
    {
      int v = Bound(autodepth_factor * delta,100);

      return(v);
    }
  else
    return(0);
}

/* Compute rear motor differential required to make up for heading
   discrepancy. */

static int auto_turn(int delta)
{
  if (delta > 180)
    delta -= 360;
  else if (delta < -180)
    delta += 360;
  if (abs(delta) > autoheading_tolerance)
    {
      int d = Bound(autoheading_factor * delta,100);

      return(d);
    }
  else
    return(0);
}
```

```
/* Actions: */

void h_velocity(horizontal_velocity *h)
{
  velocity.right = h->right;
  velocity.forward = h->forward;
  adjust_motors();
}

void zero_depth(int *depth);

void v_velocity(int *rate)
{
  manual_v = *rate;
  velocity.down = manual_v;
  adjust_motors();
  if (manual_v == 0 && autodepth_mode)

    /* This is the end of a temporary manual override of
       autodepth.  Try to maintain the next reported depth. */

    Set_Port(DEPTH_REPORT,zero_depth);
}

void autodepth(byte *status)
{
  autodepth_mode = *status;
  if (autodepth_mode) /* try to maintain the depth at next
                         reported value */
    Set_Port(DEPTH_REPORT,zero_depth);
}

void hold_depth(int *depth);

/* Reset the target depth for autodepth control to the reported
   value, then go back into the hold state. */

void zero_depth(int *depth)
{
  target_depth = *depth;
  Set_Port(DEPTH_REPORT,hold_depth);
  status.depth = *depth;
}

void hold_depth(int *depth)
```

```
{
  if (manual_v == 0 && autodepth_mode)

    /* Autodepth is engaged and not being temporarily overridden
       by the up and down controls.  Compute a correction to try
       and achieve the target depth. */

    velocity.down = auto_vertical(target_depth - *depth);
  status.depth = *depth;
}

void zero_heading(int *compass);

void heading(int *differential)
{
  manual_diff = *differential;
  rear_differential = manual_diff;
  adjust_motors();
  if (manual_diff == 0 && autoheading_mode)

    /* This is the end of a temporary manual override of
       autoheading.  Try to maintain the next reported heading. */

    Set_Port(HEADING_REPORT,zero_heading);
}

void autoheading(byte *status)
{
  autoheading_mode = *status;
  if (autoheading_mode)
    Set_Port(HEADING_REPORT,zero_heading);
}

void hold_heading(int *compass);

void zero_heading(int *compass)
{
  target_heading = *compass;
  Set_Port(HEADING_REPORT,hold_heading);
  status.compass = *compass;
  Send_Msg(HOST_ID,sizeof(rov_status),DISPLAY_STATUS,&status);
}

void hold_heading(int *compass)
{
  if (manual_diff == 0 && autoheading_mode)
```

```
    /* Autoheading is engaged and not being temporarily overridden
       by the manual heading control.  Compute a correction to try
       to achieve the target heading. */

    rear_differential = auto_turn(target_heading - *compass);

  adjust_motors();
  status.compass = *compass;
  Send_Msg(HOST_ID,sizeof(rov_status),DISPLAY_STATUS,&status);
}

/* high_decs.c - Declarations for HIGH_CONTROL slot. */

#include <action_lib.h>
#include "rov.h"

void high_decs()
{
  motor_info init_motor_settings[NMOTORS]
                                  = {{0,1},{0,1},{0,1},{0,1}};
  velocity_vector init_velocity = {0,0,0};

  Def_Port(H_VELOCITY,h_velocity,horizontal_velocity,0,1,LO_PRI);
  Def_Port(V_VELOCITY,v_velocity,int,0,1,LO_PRI);
  Def_Port(AUTODEPTH,autodepth,byte,0,1,LO_PRI);
  Def_Port(HEADING,heading,int,0,1,LO_PRI);
  Def_Port(AUTOHEADING,autoheading,byte,0,1,LO_PRI);
  Def_Port(DEPTH_REPORT,hold_depth,int,0,1,LO_PRI);
  Def_Port(HEADING_REPORT,hold_heading,int,0,1,LO_PRI);

  Var(int,target_depth);
  Var(int,target_heading);
  Var_I(int,autodepth_tolerance,1);
  Var_I(int,autoheading_tolerance,1);
  Array_I(motor_info,motor_settings,NMOTORS,init_motor_settings);
  Array(s_byte,motor_target,NMOTORS);
  Var_I(velocity_vector,velocity,init_velocity);
  Var_I(int,manual_v,0);
  Var_I(int,manual_diff,0);
  Var_I(int,rear_differential,0);
  Var_I(byte,autodepth_mode,FALSE);
  Var_I(byte,autoheading_mode,FALSE);
  Var_I(int,autodepth_factor,5);
  Var_I(int,autoheading_factor,5);
  Var(rov_status,status);
  Var_I(int,report_settings,FALSE);
```

```
}

/* low_control.c - Actions for the low level motor control. */

#include <action_lib.h>
#include "rov.h"

extern rtime_t duty_cycle;
extern rtime_t next_duty_cycle;
extern rtime_t safety_margin;
extern int on_count;
extern byte motor_state[];
extern motor_info motor_settings[];
extern rtime_t fudge_factor;

static const byte motor_bit[NMOTORS] = {0x01,0x02,0x04,0x08};

/* Switch one of the motors on or off.  Each motor is switched on
   for a percentage of the duty cycle given by the speed.  A
   switch_motor action is scheduled for each motor at the start of
   each duty cycle and at a point into the cycle given by the
   speed. */

void switch_on()
{
  int i,j;
  rtime_t now = rtime();
  byte different_speed[NMOTORS] = {TRUE,TRUE,TRUE,TRUE},
  motors_on = MOTOR_ON,
  motors_plus = MOTOR_PLUS,
  motors_minus = MOTOR_MINUS;

  while (next_duty_cycle < now)
    next_duty_cycle += duty_cycle;

  /* Motors which have the same speed are scheduled to be turned
     off at the same time.  All motors which are to be turned on
     are grouped and turned on together.  Similarly all motors
     rotating in the positive direction are grouped separately
     from those rotating in the negative direction. */

  for (i = 0; i < NMOTORS; i++)
    {
      if (motor_settings[i].speed > 0) /* Don't turn on motors
                                          with speed 0. */
        motors_on |= motor_bit[i];
      if (motor_settings[i].direction == 1)
```

```
        motors_plus |= motor_bit[i];
      else
        motors_minus |= motor_bit[i];
    }

  Send_Msg(ROV_DEV,1,10,&motors_plus);
  Send_Msg(ROV_DEV,1,10,&motors_minus);
  Send_Msg(ROV_DEV,1,10,&motors_on);
  now = rtime();
  for (i = 0; i < NMOTORS; i++)
    if (different_speed[i])
      {
    byte motors_off = MOTOR_OFF,
    speed = motor_settings[i].speed;

    for (j = i; j < NMOTORS; j++)
      if (motor_settings[j].speed == speed)
        {
          motors_off |= motor_bit[j];
          different_speed[j] = FALSE;
        }
    /* Don't schedule a switch_off for motors which on all the
       time. */

    if (speed < 100)
      {
        route_send_msg(slot_id,1,SWITCH_OFF,
                now + duty_cycle * speed/100.0 + fudge_factor,
                &motors_off);
        on_count++;
      }
      }

  if (on_count == 0)
    reschedule_for(next_duty_cycle);
}

/* Action which switches off a set of motors.
   If these are the last motors to be switched off in this duty
   cycle the switch_on action is rescheduled. */

void switch_off(byte *motors_off)
{
  Send_Msg(ROV_DEV,1,10,motors_off);
  if (--on_count == 0)
    route_send_msg(slot_id,0,SWITCH_ON,next_duty_cycle,NULL);
```

```
}

/* Action which adjusts the motors with the value sent from the
   higher level control slot. */

void update_motors(motor_info settings[])
{
  int motor;

  for (motor = 0; motor < NMOTORS; motor++)
    motor_settings[motor] = settings[motor];
}

/* motor_decs.c - Declarations for low level motor control
   actions. */

#include <action_lib.h>
#include "rov.h"

static motor_info init_motor_settings[NMOTORS]
= {{0,1},{0,1},{0,1},{0,1}};

void motor_decs()
{
  Var_I(rtime_t,duty_cycle,1.0);
  Var_I(rtime_t,next_duty_cycle,rtime());
  Var_I(rtime_t,safety_margin,0.001);
  Var_I(int,on_count,0);
  Var_I(rtime_t,fudge_factor,0);
  Array_I(motor_info,motor_settings,NMOTORS,init_motor_settings);
  Def_Port(UPDATE_MOTORS,update_motors,motor_update[4],0,1,
           LO_PRI);
  Port(SWITCH_ON,switch_on);
  Def_Port(SWITCH_OFF,switch_off,byte,0,0,HI_PRI);
  set_post(Pri(0));
}

/* Motor_init.c - initialisation for the low level motor control
   actions. */

#include <action_lib.h>
#include "rov.h"

extern rtime_t next_duty_cycle;

void motor_init()
{
```

```
  /* Start duty cycle: */

  route_send_msg(slot_id,0,SWITCH_ON,next_duty_cycle,NULL);
}

/* poll.c - Actions for performing the polling for depth and
   compass values which are reported back to the HIGH_CONTROL
   slot. */

#include <action_lib.h>
#include "rov.h"

extern rtime_t duty_cycle;
extern rtime_t next_duty_cycle;

void report_depth(int *status_value);
void report_compass(int *status_value);

void poll_depth()
{
  byte depth_request = DEPTH_POLL;
  rtime_t now = rtime();

  if (FEEDBACK != LOW_CONTROL)
    while (next_duty_cycle < now)
      next_duty_cycle += duty_cycle;
  Send_Msg(ROV_DEV,1,11,&depth_request);
  Set_Port(STATUS,report_depth);
}

void report_depth(int *status_value)
{
  byte compass_request = COMPASS_POLL;

  Send_Msg(HIGH_CONTROL,sizeof(int),DEPTH_REPORT,status_value);
  Send_Msg(ROV_DEV,1,12,&compass_request);
  Set_Port(STATUS,report_compass);
}

void report_compass(int *status_value)
{
  Send_Msg(HIGH_CONTROL,sizeof(int),HEADING_REPORT,status_value);
  route_send_msg(slot_id,0,POLL_DEPTH,next_duty_cycle,NULL);
}

/* poll_decs.c - Declarations for depth and compass polling
   actions. */
```

```
#include <action_lib.h>
#include "rov.h"

void poll_decs()
{
  if (FEEDBACK != LOW_CONTROL)
    {
      Var_I(rtime_t,duty_cycle,1.0);
      Var_I(rtime_t,next_duty_cycle,rtime());
    }
  Def_Port(POLL_DEPTH,poll_depth,void,0,1,LO_PRI);
  Data_Port(STATUS,report_depth,int);
}

/* poll_init.c - initialisation for the LOW_CONTROL slot. */

#include <action_lib.h>
#include "rov.h"

void poll_init()
{
  int motor;

  configure_device(ROV_DEV,sizeof(int),slot_id,STATUS);

  /* Start polling: */

  Send_Msg(slot_id,0,POLL_DEPTH,NULL);
}

/* rov_actions.c - Code for the ROV simulation. */

#include <action_lib.h>
#include "rov.h"

#define BOTTOM_DEPTH 1000

/* Constants which determine the rate at which the depth changes
   with the combined effect of the side motors and the heading
   with the difference between the rear motors. */

#define DRIVE_RATE 10.0
#define ANGULAR_DRIVE_RATE 10.0

extern double depth;
extern double compass;
```

```
extern double env;
extern rtime_t old_t;
extern motor_status motor[];
extern double heading_drift_factor;
extern rtime_t on_time[];
extern int report_durations;

static const byte motor_bit[NMOTORS] = {0x01,0x02,0x04,0x08};

#define Motor_Val(i) (motor[i].state * motor[i].direction)

void update_depth(rtime_t now)
{
  /* Assume very high damping, instant acceleration and terminal
     velocity proportional to net force. */
  double down_v = (Motor_Val(LEFT_SIDE) + Motor_Val(RIGHT_SIDE))
                  * DRIVE_RATE + env;

  depth += down_v * (now - old_t);
  if (depth < 0) depth = 0;
  else if (depth > BOTTOM_DEPTH) depth = BOTTOM_DEPTH;
}

void update_compass(rtime_t now)
{
  /* Assume very high damping, instant acceleration and terminal
     angular velocity proportional to net torque. */
  double angular_v = (Motor_Val(RIGHT_REAR)
                      - Motor_Val(LEFT_REAR)) * ANGULAR_DRIVE_RATE
                      + heading_drift_factor
                        * (2*rand()/32767.0 -1);
  int intcompass;

  compass = compass + angular_v * (now - old_t);
  intcompass = compass;
  compass = intcompass % 360 + compass - intcompass;
  if (compass < 0) compass += 360;
}

/* There is only one action as this is a simulated device.  All
   messages come to the same port and the contents must be
   decifered to distinguish the type of message. */

void rov_message(byte *data)
{
  rtime_t now = rtime();
```

```
int value,i;

update_depth(now);
update_compass(now);
old_t = now;

switch (*data & ROV_SELECT_MASK)
    {
    case MOTOR_OFF:
  for (i = 0; i < NMOTORS; i++)
    if (*data & motor_bit[i])
      {
        byte old_state = motor[i].state;
        motor[i].state = OFF;
        if (report_durations && old_state)
      {
        name_string buf;

        sprintf(buf,"%g: motor %d, %g seconds\n",now,i,
            now-on_time[i]);
        PrintS(buf);
      }
      }
  break;
    case MOTOR_ON:
  for (i = 0; i < NMOTORS; i++)
    if (*data & motor_bit[i])
      {
        motor[i].state = ON;
        on_time[i] = now;
      }
  break;
    case MOTOR_PLUS:
  for (i = 0; i < NMOTORS; i++)
    if (*data & motor_bit[i])
      motor[i].direction = 1;
  break;
    case MOTOR_MINUS:
  for (i = 0; i < NMOTORS; i++)
    if (*data & motor_bit[i])
      motor[i].direction = -1;
  break;
    case DEPTH_POLL:
  value = depth; /* convert double to int */
  Send_Msg(CONTROL_DEV,sizeof(int),0,&value);
  break;
```

```
      case COMPASS_POLL:
    value = compass; /* convert double to int */
    Send_Msg(CONTROL_DEV,sizeof(int),1,&value);
    break;
      }
}

/* rov_decs.c - declarations of state variables and ports for rov
   simulation slot. */

#include <action_lib.h>
#include "rov.h"

static motor_status motor_init[NMOTORS]
                          = {{0,1},{0,1},{0,1},{0,1}};

void rov_decs()
{
  Def_Port(ROV_MESSAGE,rov_message,byte,0,6,LO_PRI);

  Var_I(double,depth,0.);
  Var_I(double,compass,0.);
  Var_I(double,env,-0.1);
  Var_I(double,heading_drift_factor,1);
  Var_I(rtime_t,old_t,rtime());
  Array_I(motor_status,motor,NMOTORS,motor_init);
  Array(rtime_t,on_time,NMOTORS);
  Var_I(int,report_durations,FALSE);
}

/* Rov_init.c - initialisation for the ROV simulation slot. */

#include <action_lib.h>
#include "rov.h"

void rov_init()
{
  configure_device(CONTROL_DEV,1,ROV,ROV_MESSAGE);
}
```