1987

A report on debugging networked real-time multi-microprocessor systems

Anandi Krishnamurthy

Rochester Institute of Technology
School of Computer Science and Technology

A report on

Debugging networked real-time multi-microprocessor systems

by
Anandi Krishnamurthy

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: _____
Prof. Wiley R. McKinzie, Chairman

_____
Prof. Peter G. Anderson

_____
Dr. Andrew T. Kitchen

October 1, 1987

I, Anandi Krishnamurthy, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

1 October 1987

Acknowledgements

I am greatly indebted to Xerox Corp. for financing and otherwise supporting my graduate education. I wish to thank Prof. Wiley McKinzie for his comments and suggestions about the form and contents of this report.

Abstract: Software development in networked real-time multi-microprocessor control systems is made difficult by the real-time constraints and parallel processing inherent in such systems. Software development in general, and debugging, in particular, of networked real-time multi-microprocessor systems is discussed. A tool that is a part of a debugger for such a system is described. The tool runs on a workstation with a passive user-interface and provides high-level access to the control software. The commands supported by the tool are analyzed. The software architecture of the tool is documented using data flow diagrams, a dictionary, file formats and pseudo-code.

Key Words and Phrases: networked real-time multi-microprocessor systems, high-level debugging, non-invasive debugging.

Computing Review Subject Codes: D.1.3 Concurrent Programming; D.2.2 Tools and Techniques; D.2.5 Testing and Debugging; D.3.3 Language Constructs;

Debugging networked real-time multi-microprocessor systems

1. Intoduction

Today, many industrial real-time control systems use multiple networked microprocessors to control and monitor their processes. Software development for such systems is made difficult by the real-time constraints and parallel processing inherent in such systems. This report explores software development, in general, and debugging, in particular, of networked real-time multi-microprocessor systems. It describes a debugger for such systems, discusses its functional characteristics and system design and suggests possible enhancements. In the rest of this report the microprocessors in the control loop are referred to as *target microprocessors* or *target processors*.

1.1 Multi-microprocessor systems

Software development in networked real-time multi-microprocessor systems is greatly influenced by the following characteristics of such systems:

1. absence of peripherals. The processors are embeddded in other machinery and lack conventional IO and secondary storage devices like keyboards, display terminals and disk drives. The code they execute is stored on ROM.
2. real-time constraints. These systems must respond reliably within guaranteed response times to inputs of varying priorities. The microprocessors must temporarily suspend current activity to service inputs of higher priority. This requirement implies that a priority driven multi-tasking environment is the most natural choice for real-time control systems. Polling is also used to service requests in real-time control systems; however, this technique is somewhat inflexible and only applicable for selected applications. This report concentrates on multi-tasking environments.
3. extensive IO and interprocessor communications. These systems interface with a variety of external interfaces like AD & DA converters and various sensors and instruments. Interprocessor communications also have to be serviced.
4. parallel processing or concurrency. Two levels of parallelism are present in such systems: (i) multiple processors on the network run in parallel; (ii) multiple tasks run concurrently within a single processor. Concurrency raises problems of resource sharing and deadlock management.
5. error handling. Real-time systems must not only have the capability to detect errors but to recover from most error conditions and continue operating as best

they can. For example, it is unacceptable for an aircraft guidance system to terminate on receiving a bad coordinate input. Instead, it should log the error condition, take corrective action, e.g., reject the bad coordinate and wait or request for another, and continue processing. The degradation in performance and reliability caused by an error condition has to be minimal.

Debugging networked real-time multi-microprocessor systems is complicated by the following:

1. Multiple target processors may need simultaneous debugging. Attaching independent debuggers, each with its own monitor, to the multiple targets makes it difficult to observe and synchronize the activities of the targets.
2. The debugging aids cannot run on the target processor. The target processors, typically, do not have sufficient resources to support a debugger and even if they do, running the debugger on the targets may alter the nature of timing related bugs, common in real-time systems.
3. Breakpoints are of limited use in isolating difficult to reproduce, timing related bugs. Extensive tracing is often the only way to locate such bugs, particularly when multiple processors are involved. Tracing requires large amounts of disk storage, and / or dynamic memory.
4. The software for such systems is normally developed on a separate computer and down loaded to the target processor. To fix a bug and test the fix, the programmer has to edit the source file, compile and bind the program on the software development computer and then download the object file to the target processor. Most programmers find it much simpler to directly patch the target memory with the fix. This leads to frequent patching of the target processor memory during testing, thus, creating a major problem in source files maintenance.

1.2 Networked real-time multi-microprocessor debuggers

Until recently, most real-time control systems were debugged using logic analyzers and oscilloscopes. This is becoming more difficult as the programs become larger and more complex. Currently, there is a growing interest in developing debugging tools for networked real-time multi-microprocessor debuggers.

[Lesh 78] describes a system made up of multiple minicomputers communicating through a set of global buses. The distinguishing feature of this system is a special

synchronized interrupt that occurs every 2.5 milliseconds on all the processors. This interrupt, known as the RTI (Real Time Interrupt), causes each processor to begin executing the tasks in its queue. The underlying assumption is that no set of queued tasks take more than a fraction of 2.5 ms. Messages originating during a *tick*, or 2.5 ms step, are not consumed until the next tick, thus, simplifying the timing of message transmissions. The machines can be synchronously halted at the end of each *tick*, after which interprocessor results can be examined, bugs detected, processor states modified, if necessary, and processing resumed. All bus transfers are recorded in both the originating and consuming processors; this assists in detecting communications related bugs. A copy of the debugger resides on each processor. In addition, the debugger provides conventional breakpoints for analyzing problems on a single processor.

[Greg 82], [Conte & Greg 80] describe a debugger for a very general purpose, message passing, multi-processor system. The target system is made up of multiple clusters, each having multiple processors. Each processor in a cluster has local memory, divided into private and global parts. The global memory is accessible to all other processors in the cluster via a global bus. Inter-cluster communications are through serial links. Debugging in this system takes place at two levels. The upper level is a single debugger task that runs on one of the target processors and provides a central user-interface. At the lower level, identical debugger tasks run on each target processor. The debugger, as a whole, provides commands to load, run, suspend and delete tasks and to display and patch memory and registers. It also has commands to simulate message transmission between any two tasks. There is no symbolic debugging.

The debug system described by [Haney 80] is also double layered. The top debugging layer runs on a single minicomputer with an interface to the lower debugging layer. This layer provides symbolic debugging and a common user-interface. The lower layer is made up of multiple debug hardware modules, each with its own console for user interaction. Each debug module can be connected to a maximum of 8 microprocessors. The debugger provides high speed system loading, high level symbolic debugging, command file capability, breakpoints and tracing with triggering between processors. When a breakpoint is hit, a single processor or all the processors may be halted. The lower level debug modules provide RAM to simulate the ROM in the target processors.

The preceding examples illustrate three vastly different approaches for debugging networked real-time multi-microprocessor systems. In each case, the debuggers are tailored to a specific target system and are not general purpose. This report attempts to define a more general purpose debugger for networked real-time multi-microprocessor systems.

1.3 Project overview

DEBUG, the debugger, described in this report, was conceived as a part of a workstation based software development environment for networked, real-time multi-microprocessor systems. The project included the definition of a high level language and the development of a compiler and a binder for that language. Most of the debug hardware was designed earlier and was available at that time.

The definition, design and implementaion of the debugger lasted two years. There were four programmers on the project during most of the definition and design phases. I joined the group during the early stages of the project. I played a passive role during most of the definitions phase, attending numerous long meetings with potential users, adjusting to the company jargon and learning a new programming environment and language. My contributions did not come until the design phase when I designed the interface between the high level language and the debugger and participated in the design of the other pieces. One other programmer and I did most of the implementation, testing and initial product release. The debugger was implemented in 1983 and an improved version is in use today.

1.4 Objective and outline

This report discusses the problem of software development and debugging in multi-microprocessor systems. It describes a debugger implemented for such a system and discusses its merits and demerits. The report is organized as follows: Chapter 2 describes programming language features for multi-microprocessor systems. The relationship between programming languages and debugging is aso discussed. Chapter 3. lists all the features that are desirable for an "ideal" debugger. Chapter 4. gives a functional description of DEBUG and compares it to the ideal debugger of Chapter 3. Chapter 5 gives an architectural description of DEBUG to demonstrate its

algorithmic and data-structural complexity. Chapter 6 contains concluding comments and areas of future work.

2. High Level Language and Debugging

A good high level language is a great asset in writing, debugging and maintaining most software systems. The functions that a system is to perform and other design level decisions are often affected by the language in which it is to be written. This is especially true in the development of distributed real-time systems. A language suitable for distributed real-time systems should have all the features of a good conventional programming language. In addition, it should have special constructs to represent the distributed and real-time aspects of the system.

This chapter explores the language characteristics that aid in the design, development and testing of networked real-time multi-microprocessor systems.

2.1 Definitions

High Level Language: A high level language is an abstract model in which data and operations on that data can be specified in problem-oriented rather than machine-oriented terms. Data objects are referred to by user-defined names rather than by memory addresses. Operations on the data objects are specified in terms that are more easily understood by the human user. Each such high level operation, typically, breaks down into multiple computer instructions.

High Level Debugging: High level debugging is more difficult to define. In low level debugging, the user is provided with enormous quantities of machine level data from which meaningful information must be gleaned. In high level debugging, this meaningful information is extracted by the debugger, thus, the user is able to directly debug the problem rather than sifting through mounds of machine details to reconstruct it. That is, high level debugging is using tools to filter and interpret the data provided by a low level debugger and to present it to the user in a format such that the user can work in the problem domain rather than the solution domain. Ideally, when carrying out high level debugging, the user is not concerned with anything below the programming language level.

2.2 Language requirements for networked real-time multi-microprocessor systems

In real-time systems, unlike other types of computer systems, the time interval for an input to be serviced is of prime importance. A typical system will have different categories of inputs, each having to be serviced with different degrees of urgency. Consider, for example, a distillation column controlled by a real-time computer system, where one of the inputs to the computer is the temperature of the mixture. If the temperature rises above safe levels, the computer has to respond very quickly by turning off the heater and/or turning on the cooling system. The response has to be almost immediate. However, if the temperature was still within the normal range, the system can take longer to respond. This requirement for servicing different inputs within different fixed time periods largely determines the language requirements for a good real-time language.

The task of monitoring different inputs and responding appropriately is inherently non-sequential. When the system is responding to an input, another one of a higher priority may occur, causing the system to suspend the current process, assign it a lower priority and service the new input. In addition to controlling inputs, real-time systems may perform such functions as managing an operator interface, logging data for future analysis and providing statistical information from the logged data. Therefore, more than one task can be active at any time. Such systems are best conceptualized as multi-tasking systems in which each task runs as a process with a given priority. A language suitable for programming real-time systems should then have constructs to specify independent processes, possibly running in parallel, and the ability to assign priorities to them.

Another important language criterion is the ability to handle exceptions effectively, without comsuming excessive resources for exception checking during normal operation. Since timely handling of IO is the single most important function of real-time systems, a good real-time language should provide either high level constructs or pre-written interfaces to support IO.

Distributed systems must provide for inter-processor access. In tightly-coupled systems, this access is acheived with shared memory and inter-processor task syncronization. In loosely-coupled systems, access is done with message passing between inter-processor tasks. Hence, a language suitable for distributed real-time

systems should provide constructs that allow processes running on different processors to communicate with each other without compromising data-integrity.

Many of the language requirements mentioned above are common to both single processor as well as distributed real-time systems. In addition, the language must possess the qualities of any good programming language. It should have extensive compile time error checking, including strict data-type checking and extensive run time error checking with extra code inserted by the compiler. It should be well-structured and readable, thereby, making it easier to locate and fix bugs. The constructs provided by the language should be flexible enough to naturally express the actions desired by the programmer. The constructs and features of the language should be simple and unambiguous. The compiler should generate code that takes maximum advantage of the features of the hardware on which the code is executed. At the same time, the language should be fairly independent of the hardware so that programs can be easily ported from one machine to the other.

Some of the requirements in the previous paragraph are contradictory. For example, compiler inserted run time error checking code decreases the run time efficiency of the program. As usual, programming language design amounts to making trade-offs between desirable features to maximize the useability of the language.

2.3 Language features that aid in debugging

In the previous section, it was shown that real-time programming is characterized by multi-tasking, specialized exception handling and IO programming and, a high level language for such systems should have special constructs to express these features naturally and concisely. Such a language also has a positive impact on debugging, since implementing a program that closely resembles the design results in easier understanding of the program during debug time. This section discusses language features particularly relevant to debugging distributed real-time systems.

Data Typing: Until now, real-time programs have been written in low level, weakly typed languages that allowed the programmer to take full advantage of the machine's capability. The disadvantage in this approach is the difficulty in insuring reasonably bug-free programs. As the programs become larger, the machines become more powerful and faster, and more effective structured high level

languages are developed, the justification for low level, weakly typed languages diminishes. New programming languages for real-time systems are strongly typed and their compilers check for data type conformance. ADA and Mesa [Geschke 77] are two such languages.

Modularization: This feature allows a large programming problem to be split into smaller parts, or *modules*, which share data and operations on that data. The modules can be compiled separately and put together, or they can be written as one unit with each part maintaining its properties. Here we will concentrate on modules that compile separately. Typically a module has two parts - the interface specification and the module implementation. Modules communicate through their interfaces. These are, essentially, templates made up of type, data and procedure declarations, but not executable code. The interface declarations are implemented by one or more program modules that *export* the interface. Other modules use items declared in the interface by *importing* it. Modularization aids debugging in many ways.

A system implemented as modules can closely resemble the design level breakdown of tasks and functions. That is, a top-down design that progressively breaks down a large problem into smaller and smaller hierarchical tasks can be implemented by programming a module for each task. That is, design is done in the problem domain, yet the design level relationships between tasks are preserved in the module implementations and the shared objects in the design are defined in the module interfaces. Therefore, implementation decisions are based on the design rather than design decisions being based on the implementation. Separation of the interface from the implementation makes the high level functions of the modules apparent without having to know extraneous implementation details.

Modularization helps a group of programmers in building large systems. Once the module interfaces are specified, the programmers can independently implement and partially test the modules. That is, the effects of implementation details and changes are largely localized. When an error is detected, the fix often affects only one module or a group of closely related modules. Modules can be tested one, or a few, at a time. The large system can be integrated with a higher degree of confidence since the individual modules have already been tested. In most cases, a module's state cannot be corrupted from the outside since access to it is restricted.

To some extent, modularization organizes error handling. All the error conditions anticipated by the caller of a public procedure are declared in the interface. All other error conditions are handled locally within the public procedure. This formally breaks down the error handling responsibilties between the implementing and the importing modules and leads to fewer instances of neither modules handling an error condition or both modules handling the same error condition.

Modules that implement often-used programming objects such as stacks, lists, string to number conversions and input parsers, can be written and debugged and then used repeatedly in many projects. This also ensures that different projects exhibit uniform behavior.

The above points are quite general and are relevant to both conventional and real-time programs. More specific to real-time programs, modularization can provide mutual exclusion for sensitive data. (Refer to the next section for more on mutual exclusion.) Access to sensitive data is restricted by declaring it within a module rather than its interface. Thus, only procedures declared in the module have access to the data. Restrictions can then be imposed on those procedures to control access to the data. Modularization also aids communications handling and IO programming. Device specific, special purpose modules can be written for IO and communications devices. Once implemented and tested for a device, a module can be used by multiple clients. The interfaces exported by these devices can be standardized, thereby, reducing the time taken by a programmer to become familiar with a device.

Multi-tasking: A computer system is said to be capable of multi-tasking if it can handle more than one task at a given time. A task here refers to a chain of program control - a process. The tasks in a multi-tasking system execute in parallel, interacting, where necessary, with other tasks. There are two types of multi-tasking. In one, the tasks run on multiple physical processors sharing common memory. The other type has only one processor, one currently active task and multiple suspended tasks. The following discussion applies to both types of multi-tasking. Real-time distributed systems almost always have multi-tasking at the single processor level. Multi-tasking on multiple processors, sharing common memory, is less common. Many recently developed languages support multi-tasking. These include ADA, Modula [Wirth 77] and Mesa [Lampson, Redell 80].

There are two types of interaction between tasks. One is the sharing of data between tasks and the other is the synchronization of execution. Consider two tasks A and B, where A is the producer and B is the consumer of some data *x*; then x is the data shared by A and B. For the producer-consumer relation to function correctly B has to be told when a new x has been produced, so it can use x. And A has to be told when B has finished consuming x so it can fetch or generate the next x. Therefore, A and B have to communicate to each other their execution states in order to function correctly, i.e. synchronization of execution. Further, the access by A and B to the common memory area where x is stored has to be mutually exclusive, i.e, mutual exclusion of shared data. Ensuring mutual-exclusion for data access and providing process synchronization are two very important features of multi-taking systems.

One major area of concern in multi-tasking system is error-handling. In single thread control, if some error situation occurs, it is possible to take corrective action and proceed or stop as appropriate. In multi-tasking systems, either all processes have to detect all errors or a process has to have a way of aborting other processes when it detects an error (that is, it must kill its children before dying). Also, to ensure data integrity, processes waiting on conditions have to take special action. Most languages provide features for aborting processes but their use and implementation are torturous.

Mutual exclusion for shared data access, process synchronization and error-handling make multi-tasking extremely complex. The manner in which a language implements these features has great impact on debugging programs written in that language. Inadequate support for these constructs will result in more bugs while coding and, thereby, additional time to fix them.

Exception Handling: This is the detection of errors in computer systems and the actions taken to deal with them. Errors in software systems can be broadly classified into three categories: (i) low level hardware errors such as a parity error while reading from disk/ memory, (ii) low level software errors such as a divide by zero and (iii) programming or logic errors such as array-index range violation.

In conventional programs, it is adequate to output an error message and abort execution of the program when an error is detected. A real-time system, which controls other machinery and physical processes, cannot just abort execution. It

must take actions to minimize the effects of the error and carry on, retaining as much functionality as possible. In many cases of computer-controlled systems, a temporary degradation of performance is preferable to a complete shut-down. In multi-processor systems, the other processors may need to be made aware of the error condition.

The traditional way of handling errors is to check for exception conditions explicitly, where ever they may occur and take action, such as, simply output a message and abort the program. However, to recover from the error, other procedures in the calling sequence have to be told about the error so they, in turn, can take corrective action. Each procedure call must, therefore, anticipate all possible errors that the called procedure may encounter, as well as pass notification of errors back to its calling procedure. Another disadvantage is the time spent checking for all the error conditions possible, to ensure that they did not occur.

The newer languages allow more flexible error-handling. When an error is detected, the procedure can attempt to rectify it and proceed, or it can decide to propagate the error to its calling procedure. The error handling code is separated from the main-line code. Once an error has been handled, procedures that propagated the error are given an opportunity to clean up, for example, free any locally allocated dynamic storage. This kind of exception-handling allows the main-line code to concentrate on the more probable situations. Exceptions are dealt with separately. The written program more closely resembles the real problem, thus, making it easier to debug.

IO programming: IO plays a very important part in real-time programming. Real-time programs drive unusual IO devices such as analog/digital converters, digital IO ports and various instrument interfaces. Although most general purpose high level languages have IO libraries or modules to support the standard devices like disks, keyborads and printers, until recently, the most common way to program a non-standard IO device was to use assembly language. Many older real-time languages allowed assembly level statements to be interspersed among high level statements. This is a time consuming and frustrating way to program IO; therefore, a language, to be suitable for real-time programming, must provide special high level IO constructs.

Communicating with IO devices often involves writing values to IO ports. Often, a certain bit in a certain port has to be set or reset. This code usually looks something like #5A.3 : = 1. For values that are more than 1 bit long this is even more complicated since a mask has to be used to clear the current value of the bits and then the new value set. Instead, a programmer should be able to define ports as variables and bits as qualifiers with a user-defined record type. Values can then be assigned to one or more bits using the field names defined in the record type. Using such a feature, #5A.3 : = 1 would become tempTimer.counterSelect : = 1 where "tempTimer" is defined as a port variable at #5A and "counterSelect" is the qualifier for bit 3.

Most computer systems perform IO by either continuously polling each IO device or by having the IO device interrupt the main processing when needed. Interrupt driven systems are more time efficient and more commonly used in real-time systems. The priorities of IO are also more naturally implemented with interrupt-driven IO. In such systems, the IO hardware interrupts the CPU when it has valid input data or has completed an output operation. These interrupts have different priorities attached to them. If an interrupt of a higher priority occurs, the CPU suspends its current task and services the interrupt. Once the interrupt is serviced, the CPU resumes the suspended task. The various tasks also need to be synchronized correctly. This model meshes closely with the concept of processes discussed earlier. A language supporting processes is useful in IO programming.

The concept of modules and interfaces can also be used to make IO programming more understandable. A programmer can provide an interface to a class of IO devices that is largely device-independant. The implementing module can then contain the device-dependant portions. This has many benefits. The module can be used in more than one application using similar hardware devices. New devices can be added with minimum changes to the interface module. A programmer can communicate with IO devices through the interfaces without knowing all the detailed working of the devices.

In addition to all external IO, multi-processor systems need to communicate with each other. This communication may be through shared access to the same memory or by exchanging messages through communication lines. It is more common to

write library routines or an interface and exporting modules to implement communications. Languages rarely have constructs for communications.

High level IO programming makes it easier to implement IO intensive real-time systems. This reduces the number of IO errors made while coding and the time taken to detect them.

2.4 Language and Debugging. How the two are related.

This section describes and binds the relationship between a high level language and a debugger. It also introduces the tables that must be generated by the high level language compiler for a high level debugger. These tables are referenced in the next three chapters.

After a module is written in a high level language, it is compiled, perhaps bound with other system modules and then debugged. An integrated development system in which the editor, the compiler, the binder and the debugger, all run on the same machine is ideal. This provides for bug fixes at the source level rather than patches to target memory. Incremental compilers and binders are also desirable for the same reason. They reduce the time it takes to make a source level change and get it running on the target system, ready to debug.

The speed and efficiency of debugging can be improved if the abstractions provided by the high level language are available at debug time. Otherwise, after programming in a high level language, the user still has to depend on memory dumps and detailed knowledge of what resides where and how in memory to debug his program.

Symbolic or *high level* debuggers preserve the abstractions at debug time. They use the source and object files of the high level language to provide special debug features, such as: (i) the ability to specify a break point by pointing to a line of high level code, (ii) once the break is taken, the ability to look at data by referencing the high level names of variables and (iii) the ability to display the execution path in high level language statements using data from the trace buffer. To provide these features, the compiler and the binder must generate special tables, for the debugger. Two tables are commonly generated: the *correspondence table* which contains the object file offset for each source statement; and the *symbol table*

which contains the in the object file offset, size and type information for each symbol declared in the source file. The compiler and the binder calculate relative offsets and the loader converts these to absolute memory locations on the target.

An *interpreter* is a tool that runs in the debugger and allows a programmer to specify a sequence of actions much like program statements. These actions are directly executed in the program domain, without being compiled. Interpreted actions include the execution of procedures and the evaluation of high level language expressions. Thus, a programmer can experiment by executing portions of his source code with an interpreter without recompiling, rebinding and reloading his program.

The points enumerated in this section make it clear that, to be useful, a debugger has to work closely with the high level language in which the programs being debugged are written.

2.5 Conclusion

High level languages allow programs to be written in a form that is more easily understood by human beings than raw machine code or assembly language. Languages that support real-time systems have to provide additional features to cover the special needs of such systems. Exception handling, multi-tasking and IO programming are a few such special features. A good debugger must interface with the language in which the programs are written. A programmer can then use the source and object files in conjunction with the debugger tools to test his program. Like languages, real-time debuggers have to be different from conventional debuggers. This aspect is covered in the next chapter.

3. Debugger Features

The previous chapter dealt with the impact high level programming languages, tailored to implement real-time systems, have on debugging such systems. This chapter concentrates on the debugging side of the problem. Real-time systems, especially ones that run on multiple processors, need special debuggers. Multi-tasking and interprocessor communications introduce timing related bugs that are difficult to duplicate and debug. The target configuration may consist of more than one type of processor with varying debug requirements. Debugging has to be done from a remote host for two reasons: (i) the targets often do not have the resources to support a debugger; and (ii) it is convenient to debug multiple processors from a single user console.

There are no standard procedures for developing and testing multi-microprocessor systems. Installations evolve their own techniques over the years. This chapter describes a general debug hardware configuration for networked real-time multi-microprocessor systems. This is followed by a discussion of the software features that assist in debugging such systems. Real-time tracing of processor state, a very important debugging aid for multi-processor systems, is covered in detail. User-interface features that aid debugging are also discussed.

3.1 A sample multi-microprocessor system

Real-time systems are designed to control and monitor real life processes. They control processes like space craft navigation and landings, weapons control and medical instrumentation. There are three relevant parts to such systems: (i) the system being controlled - the space craft, the weapon or the medical instrument, (ii) the control hardware and (iii) the control software. These parts are usually developed in parallel. The system being controlled may not be ready in time to start software testing. Even if it were ready, it is not advisable to start testing on it, as the untested software may seriously damage the system. Simulators are used to fill the gap. A simulator is a general purpose system that can be programmed to respond with prespecified output values to a set of prespecified inputs. Typically, software debugging begins on a simulator as soon as the control hardware, in the form of printed circuit boards (pcbs), is ready.

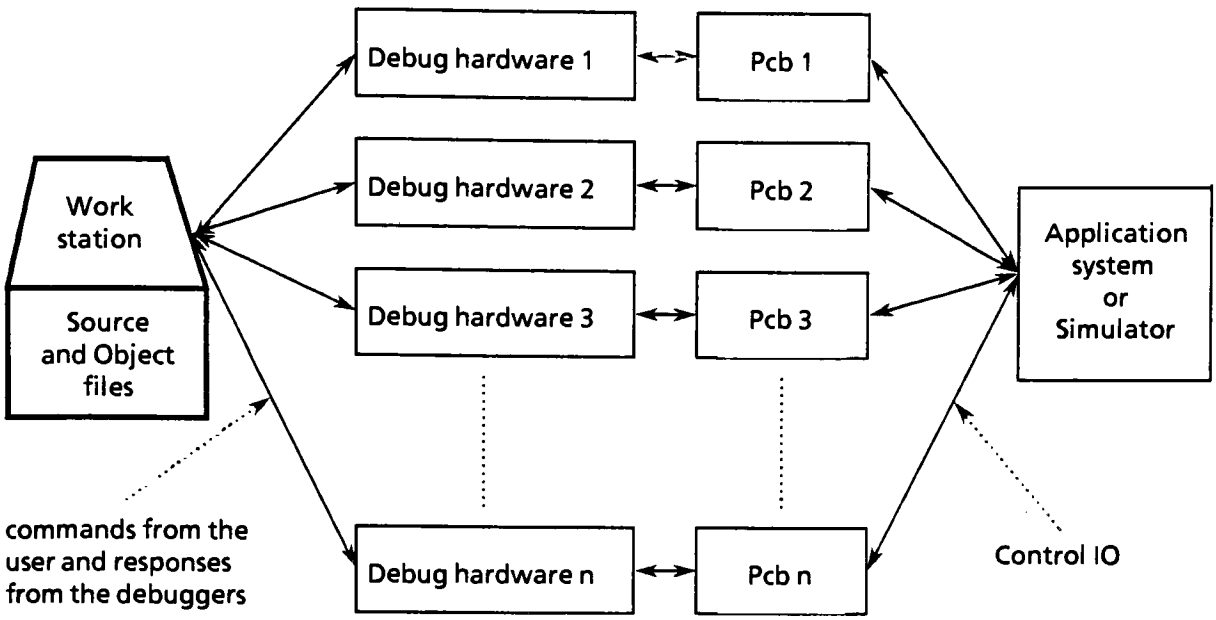

Figure 1. The debug hardware configuration

The microprocessors on the pcbs are replaced with in-circuit emulators (ICE) during the debug phase. In-circuit emulators simulate all the operations of the CPU that they replace with additional circuitry to monitor the CPU. The additional circuitry is capable of (i) detecting the beginning of each instruction sequence (ii) suspending instruction execution at a given point from where execution can begin later (iii) performing simulated memory and IO read and write operations and (iv) recovering the contents of CPU registers without destroying the contents. Before ICE modules became popular the debugger and the code being debugged had to reside in the same microprocessor. They had to share the same resources processors, memory, registers and IO - and take special care to keep out of each others way.

A typical debug hardware configuration is shown in figure 1. The pcbs for the system under design are hooked up to a simulator or the actual system being controlled. The pins of the processors or ICE modules are connected to processor specific debug hardware such as logic analyzers. There is one processor specific debug hardware module for each processor. These interpret the values passing through the pins of the processor. All the debug hardware modules are hooked up to a host computer, most often a workstation. The workstation controls the debug

hardware which, in turn, controls the target processors. Once the hardware system is set up, the user sits at the workstation and directs the debug process.

The debugger running on a remote processor has several advantages: (i) the target processors need not have additional resources to support a debugger, (ii) the debugger cannot alter the problem being debugged and (iii) the code under test cannot corrupt the debugger. The other advantage, from the user perspective, is the presence of a central console to monitor all the targets.

The workstation has the source and object files necessary for high level debugging. The object files are loaded into target memory using either the debugger or a separate loader. If a loader is used, the names and load addresses of the object files are input to the debugger. In addition to the applications code, each target is loaded with a run time executive. The run time executive provides system level functions, such as ensuring mutual exclusion and managing communications, which are usually common across many projects. As a result, once an executive is developed and tested, it is used in many projects.

The next step, after down-loading the programs into the targets, is to start the targets. Depending on how a multi-processor system is coupled, all the targets may be started simultaneously with their clocks synchronized, or a "master" processor may be started first. In a "loosely coupled" system, the processors are independent of each other and can be started in any order.

3.2 Control of multiple processes on multiple processors

Once the targets are running, the programmer sets up test conditions on the simulator or the actual hardware and observes the results. Actual debugging starts when the observed output is not the desired one . Breakpoints are commands to the debugger to halt a target when a set of conditions are true and are valuable for analyzing aberrant program behavior. The program state information is preserved when the target halts so that the user can (i) analyze the current state and (ii) resume from the breakpoint after analysis.

Breakpoints are most useful in analyzing "single path" bugs, where it is possible to recreate the sequence of events exactly. They are not directly useful in solving timing related problems since the time taken to break and examine the registers

alters the problem. Despite this deficiency, they are used extensively in debugging real-time systems. By setting breakpoints and examining the current state information, the user can examine probable causes for even timing related bugs and, thereby, make more selective decisions on how to proceed with the debugging.

Breakpoints are usually set at the beginning of machine level instructions and cause the targets to halt just before executing the specified instructions. There are two ways of specifying a machine level instruction: the actual hardware address where the instruction is stored or its location in a source module which is then translated into a hardware address. The latter method is superior since it eliminates the need for the user to look up the hardware addresses of instructions in load tables. In machines with paged memory, this can be quite tedious. Other conditions for breaking can be an attempt to write to/ read from a specified memory location, register or IO port.

User interface is a key factor in setting breakpoints. In traditional TTY type tools the user had to type in a source module name along with a line number or a procedure name to specify a source level break. Some debuggers accepted breaks only at entry to procedures. The newer window format tools are more natural to use. To set a breakpoint, the user loads the source code in a window and points, with a device such as the mouse, to an instruction in that window and selects a command from a menu.

Sometimes more elaborate conditions for setting a breakpoint are needed. Some of these are (i) break after cycling through a loop *n* times (ii) break at a source position only if specified procedures are in the call stack in a specified order or (iii) break at a source position only if a relational condition is true. Some multi-processor debugging systems provide cross processor breaks where it is possible to stop and examine target A when the break conditions specified for target B are true. This feature is needed for systems where all the other targets in the network continue running when a target halts on a break. On the other hand, this is the default in systems where all the targets are halted when one target reaches a breakpoint.

3.3 Current state information

On reaching a breakpoint, the user is interested in the current state of the program, that is, the code location where execution stopped and the state of all the variables, registers and other processes at that time. This section discusses debugger features that aid in analyzing the current state and provide ways of proceeding from the break. The commands that follow demonstrate these features and may be implemented on a TTY or a window oriented interface. User interface is covered in the next section.

Select processor: In a multiprocessor system the user must first select a target to concentrate on. In a window oriented debugger it may be possible to work on multiple processors simultaneously by bringing up multiple windows, one for each target.

List processes: This command lists all the processes that are in the process queue, including those waiting on conditions. Each entry consists of the process number, the source module, the process name and its current state (active or waiting). The user selects a process from the list and proceeds to the next step in debugging. In a window oriented system it may be possible to select more than one process by bringing up a window for each process.

Display stack: This command displays the exact address and/or source location where the process halted and sets the current context. The current context, which is very similar to scoping in high level languages, defines the variables accessible to the process at the source location at which it halted. The debugger follows the same scoping rules as the high level language used to write the program. Therefore, it is possible to travel up the stack of procedures. The current context changes at each level to include variables available at that level and to exclude those declared in the lower level. Window oriented debuggers can load the source module in which the process is defined, select the statement and set the current context where the process is currently waiting.

Display and patch memory: This is a basic feature in any debugger. The user can access any memory location by specifying its memory address. Symbolic debuggers allow accessing of variables as they are defined in a source module. Therefore, the

user does not have to know the hardware addresses or the machine representations for the data items. Only variables that are defined in the current context are available. Also, the user can assign values to data items using a subset of the syntax provided by the source language, as well as "patch" or modify code in memory. The latter case, however, may require the user to be familiar with the machine representations of various instructions.

Attach keystrokes: Another useful feature in a debugger is the ability to specify an instruction stream or command file, to be executed by the debugger, after a breakpoint is hit. Each breakpoint or cycle step can have its own command file. This feature is often used in test situations where a breakpoint is expected to be hit many times. When the breakpoint is hit, the debugger executes the instruction stream which may include displaying/patching data items and proceeding or cycle stepping through the code automatically. This allows the user to set up the instruction streams, start the system and return later to analyze the information displayed by the debugger.

After analyzing the current state, the user can (i) restart the program at the breakpoint, thus, executing it to another breakpoint or (ii) cycle step the program one instruction at a time where, after each step, control is returned to the debugger or (iii) invoke a procedure call from the debugger as if the procedure call was coded in the source module at the breakpoint. Another option is to discontinue the debug session. Before discontinuing a session, the user may save patches to target memory for later use. The patches are saved as files on the remote host.

3.4 Tracing

Bugs that cannot be consistently reproduced are often timing-related. They are unique to multi-tasking systems and show up only when a set of events, controlled by two or more parallel tasks, happen in a certain time sequence. A program with a timing-related bug may work correctly some of the time and fail at others. These are difficult to locate since (i) they are hard to reproduce and (ii) they involve interaction between multiple tasks which need to be analyzed simultaneously. Tracing is the most effective tool in analyzing such bugs. Tracing is also useful in debugging the more common "single path" bugs.

Tracing is the periodic monitoring of the status of one or more processor buses, usually the address and the data buses, and storing the collected information in a buffer. Trace buffers are finite in size. Once the buffer fills up, either the tracing stops or the data is lost on a FIFO basis. A *trace entry* is a set of bus values captured in one period of tracing. A trace buffer contains the history of traffic on the buses for some period of time. This data can be used to investigate bugs.

Tracing is usually done using logic-analyzers. These are external devices that are connected to the pin-leads of the microprocessor under test. They have a set of keys and a small display panel to accept commands and display results to the user. Logic analyzers do not have the software and the disk space necessary to support high level debugging. To provide source level symbolic access, the logic analyzer must be interfaced to a host machine.

Three command parameters are required to set up a trace: when to start the trace, when to stop the trace and what to store in the trace buffer. Start and stop conditions vary widely in complexity, depending on the trace hardware. They can be as simple as physical switches on the hardware. At the other end of the spectrum, they can be complex logical conditions with several levels of nesting.

Examples of start and stop conditions are: (i) the target being traced is started or stopped; (ii) physical switches are turned on or off; (iii) something is written to a specified location on the target; (iv) a specified value is written to a location on the target; (v) several instructions, identified by their memory addresses, have all been executed in a given order and (vi) the trace buffer is full. The last is a stop condition only.

At times it is desirable to know the events before and after a condition becomes true. Some trace hardware allow for this in the following way. Tracing begins immediately after the condition is specified and continues even after the condition is detected. It stops when the target stops or when there is half a buffer of data collected *after* the condition was detected.

The third parameter determines what is stored in the trace buffer. Often the user is interested in a limited area of program execution and would prefer the trace buffer contain only data relating to this area of interest. Some trace devices use hardware filters to accomplish this. These are somewhat inflexible since they are

implemented in hardware and must work in real time. Typically, hardware filters can be set up to trace selected types of bus cycles and accesses to specified memory ranges. An example for the former is tracing all memory writes and output cycles, ignoring all the others. Very often the memory ranges are restricted to a single memory location or multiple locations beginning and ending on a block boundary.

The trace entries in the buffer are in raw binary form. However, the trace devices usually have software to translate the binary addresses and instructions into assembly language form. Thus, the user can view the data in hex or assembly level on the display panel of the device.

As mentioned earlier, the logic analyzers must interface to a host computer to provide source-level symbolic access. The user can then specify trace parameters, at the source level, from the host machine and the host machine, in turn, can read the data from the trace buffer and translate them into high level form.

The software in the host machine references the symbol tables for the module names, variable names and type information for data items in the trace buffer. These are then displayed as defined in the source module. The host software uses correspondence tables to translate instructions in the trace buffer to module names and line numbers of the source files.

The source level data may be displayed a number of ways. The easiest is to display the module names and line numbers for the instructions in the trace buffer; the contents of the line may also be included in the display. This information is more effectively displayed with a window oriented system where the affected source files are loaded into windows and the executed statements are highlighted. The user, scrolling through the source file displayed in the window, has a visual picture of the execution path. It is also possible to have a query feature where a programmer types in a module name and a line number to determine if the line was executed and, if so, how many times.

In time tracing, each trace entry includes the time the information was captured. This feature is very useful in solving timing related bugs. A time trace can be done simultaneously on each processor under test. The relative timings of events recorded in the different trace buffers can provide valuble insight into the problem.

Cross processor tracing is supported by trace hardware that initiates tracing on another processor when a trace condition occurs on its processor. This feature is very useful in tracing communications bugs. For example, just before sending a message on the network, a processor can start a trace on the receiving processor. In cross tracing, the trace hardware connecting the two targets must communicate directly.

In conclusion, tracing is a very powerful tool in solving real-time problems. To be particularly useful, the trace device should have flexible ways of specifying the start and stop conditions and powerful hardware filters. To provide tracing at the source language level, the trace device must interface to a host computer running the appropriate software.

3.5 User interface

User interface is a very important aspect of a real-time debugger. The user should be able to set up and control the system with minimum effort. This includes minimum training and set up times and ease and flexibility of use. At the output end, the debugger collects large amounts of data about multiple tasks running in multiple processors. One of the major tasks of the debugger is to analyze the data collected and present it to the user in a form that is easy to understand. Some other desirable user-interface characteristics are: (i) it should be easy to learn; (ii) it should require minimum type-in, mouse-clicks or other forms of user input; (iii) it should be flexible; (iv) a user should not have to remember syntactic details of debugger commands and (v) it should not force the user into a set pattern of operation.

Though it does not have any direct bearing on real-time multi-processor debugging, the last item is interesting enough to deserve further explanation. Until recently, most tools had a serial TTY-type of user-interface where the user typed a command which locked up the user-interface till the command had completed processing, even if the command was just forking a background process. The user had two choices regarding the input parameters for a command. They were typed in with the command or entered upon system request. In either case, the inputs had to be entered in a predetermined order with no ability to suspend, and later resume, the entering activity. That is, the tools, and not the user, determined the order of parameter input.

In contrast, the window driven systems of today are more non-intrusive. A user first brings up a tool, enters the parameters and then selects the command to be performed. The parameters can be entered in any order and the commands selected at will. The user is free to enter parameters, move to another widow, do something else and then come back to the original window and complete the command. The tools are more passive. In general, the window approach is preferable when the user is sitting at the terminal and the TTY approach is better for long jobs that can be put in a command file and left running without user-intervention. Ideally, tools should support both types of interfaces.

The user interface should support the loading of multiple tools. That is, a user should be able to perform multiple tasks within the debugger as well as other tasks, such as retrieving a file from a remote location, sending mail and editing and compiling source files without having to unload and reload tools between tasks. Tools, put aside for later use, should save their state and wait for the next input. Window oriented debuggers can use multiple windows to represent multiple processes or processors; the windows can be in active or inactive states.

The best way to present status information about multiple tasks and processors is to display global level information about the state of the entire system and more detailed data about a few user-specified tasks. Global data could be the status of targets (run/stop), their IO status and the status of their communication channels. The type of global data desired by the user will vary from time to time. For this reason, the user should be able to specify a template for the debugger to display global data. The detailed data could be the number of tasks on a target, the current breakpoints and values of certain variables. It is even more important that detailed data be displayed with a template, since the amount and types of data can be quite large.

3.6 Conclusion

This chapter describes an ideal debugger. Most debuggers provide a subset of the features enumerated here. DEBUG, a debugger for networked real-time multi-microprocessor systems, is described in the next chapter. A significant portion of that chapter is devoted to how DEBUG compares against the ideal tool.

4. Functional specification for DEBUG

This chapter describes, DEBUG, the front end and user interface for a networked real-time multi-microprocessor systems' debugger. DEBUG consists of five tools and runs on a host workstation with a window oriented user interface that allows any combination of multiple tools, tasks and windows to be active simultaneously. The parallel nature of the user interface aids in conceptualizing a multi-processing environment. This chapter begins with an overview of the debug configuration. A discussion of the workstation and its software environment follows. The five DEBUG tools are described in the later half of the chapter.

4.1 Debug configuration

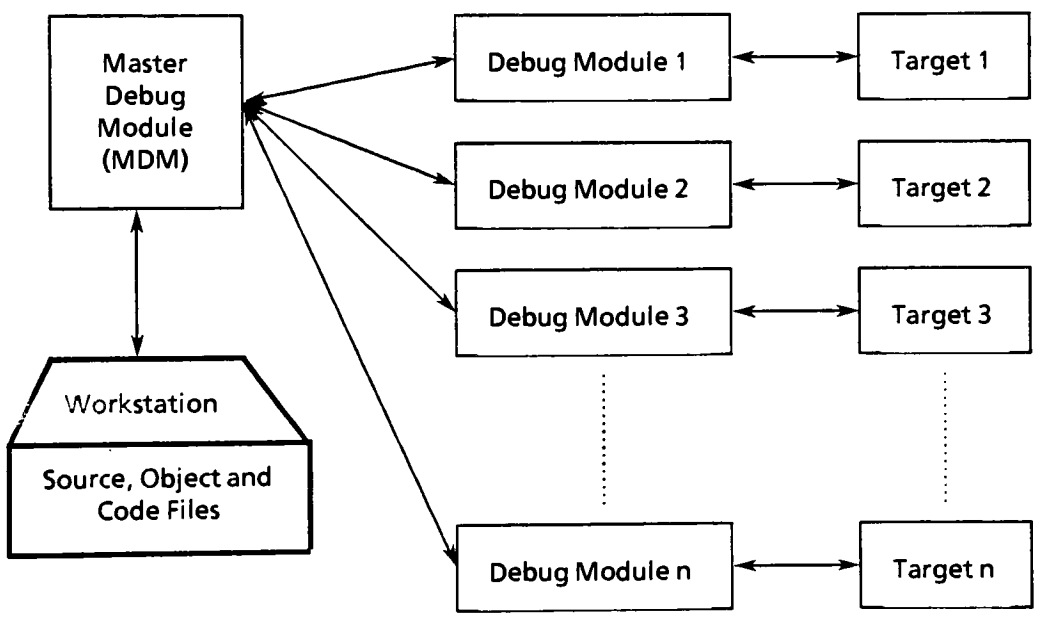

Figure 1. The debug configuration

The target system consists of multiple loosely coupled microprocessors, which may be of different types. Tasks on the same processor share memory, while tasks running on different processors communicate by passing messages on a serial network. Each processor has a run-time executive that handles the scheduling of tasks and the sharing of resources. Application programs are written in a high level

language with calls to the run-time executive for inter-processor communications and inter-task access to shared memory. The code resides in ROM in the final version of the system. RAM is substituted during testing. No debug software resides on the targets and debugging is done completely from the outside by monitoring the pin leads of the targets.

Figure 1 illustrates the interconnections between the targets and the debug hardware. Each target processor is connected to its own debug hardware, tailored for a specific type of target microprocessor. The debug hardware modules communicate with each other, as well as with the host workstation, through a master debug module (MDM). The host workstation, with its window oriented multi-tasking operating system, provides a centralized user interface for debugging the multiple target processors. It parses user commands and communicates instructions to the debug hardware. In turn, it receives messages from the debug hardware and interprets and displays them to the user.

The target application programs are written in a high level language that supports extensive type checking, multi-tasking and modularization with PASCAL like scope rules. The compiler produces a symbol table and a correspondence table as a part of the object file for each source module. The symbol table contains the address and type information for all the data items declared in the source module; the correspondence table contains the start address of the object code for each executable statement at the source level. The code generated by the compiler is relocatable; hence, the addresses stored in the correspondence and symbol tables are relative to the object module. The binder generates a table (as a part of its output code file) that is used in conjunction with the target load addresses (maintained by the loader) to convert the relative addresses to absolute addresses on the target processors.

The symbol table makes it possible to use source level names instead of absolute addresses when refering to data items in the debugger. In the same vein, the correspondence table allows the user to specify a code location in target memory by either specifying the source module name and the statement number or loading a source file in a window and using the mouse to point to the desired statement. In effect, these two tables provide source level access to the software on the targets, as

well as the ability to display information received from the debug hardware in source form whenever possible.

4.2 The workstation

DEBUG is implemented on a workstation with a high resolution bit-mapped display and an extended keyboard with several special function keys for editing text. The operating system on the worstation is multi-tasking and provides window management procedures. It also provides higher level interfaces, using the window package, for tool and menu creation and manipulation. The workstation supports the simultaneous display of many windows. Windows may overlap either partially or completely. Overlapping window areas are layered top to bottom with only the data in the top window displayed. A mouse is used to bring a window to the top or to hide it at the bottom of the window stack. Windows can be moved anywhere on the display and their sizes can be changed.

A tool is a set of one or more programs that interacts with the user to perform some useful task. In this operating system, tools interact with the user via windows of a special type known as tool windows. A tool window once created can be in one of three states: active, tiny or inactive. In the active state, the window is displayed on the screen and the user has access to data displayed in it. When the tool window is tiny, it is represented on the screen by a small fixed size brick or icon. When it is inactive, the tool window disappears completely from the screen and its name is entered in a menu list of inactive windows. Tools are taken from one state to the other using the mouse.

Tool windows are made of one or more subwindows. One of these is used to log tool activity. The others contain commands and fill-in fields for the command parameters. The parameters can be booleans, numbers, strings or user defined enumerated variables. To execute a command, the user fills in the parameters for the command, moves the mouse to the command name and clicks it. Tools may have additional command menus that provide auxilliary functionality.

This type of user interface has many advantages. All the commands are displayed in windows or in menus. The parameters for the commands can be entered in any order. The user does not have to remember the exact command syntax, thus making the system easy to learn. Type-in is kept to a minimum by having to enter

only the parameter values at the prompts. Users can tailor the display layout according to their needs, with minimal effort, by changing the number, size and position of tools and tool windows. This makes the user interface very flexible and non-intrusive in that users are not forced into a set order of actions.

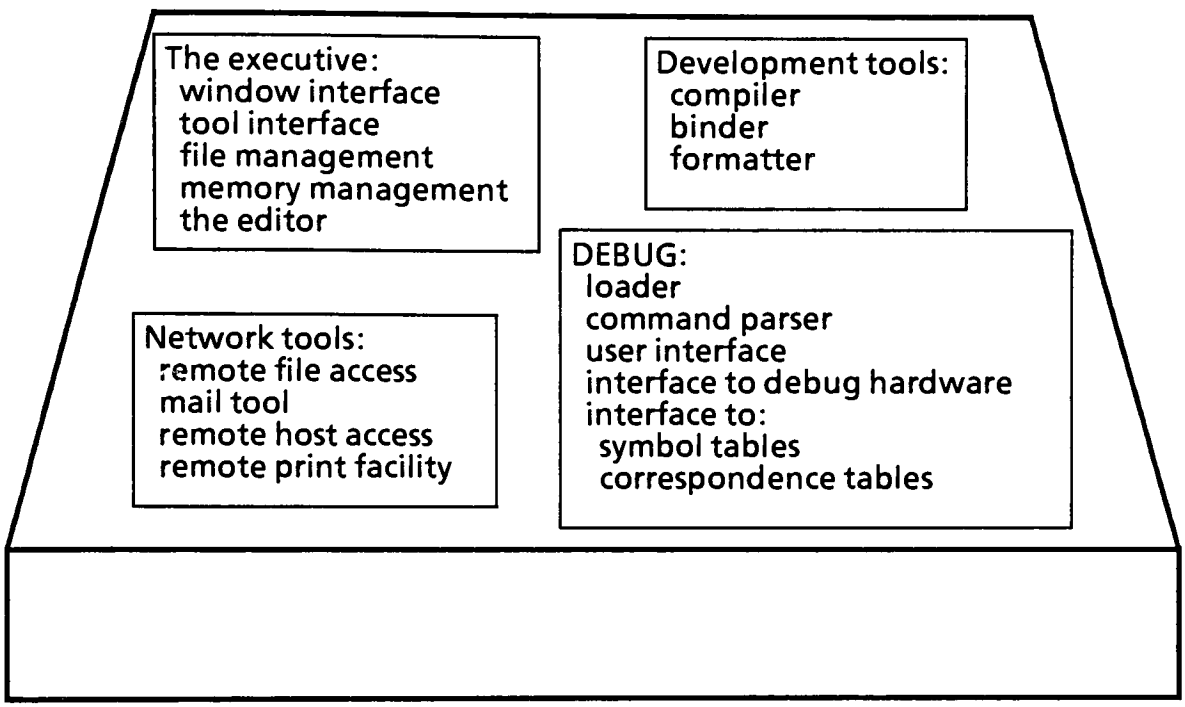

Figure 2. The software configuration of the workstation

Debugging multi-processor systems commonly involves monitoring parallel activities of multiple tasks, ports, processors and communication channels. The multiple windows and multi-tasking provided by the host preserve this parallelism. The availability, during debugging, of other tools like compilers, binders and remote file handlers, allow the user to fix bugs at source level while continuing to debug other parts of the system. Figure 2 is a list of the software available during debugging.

A "cut and paste" editor, an integral part of the operating system, is also available at all times. For example, the full capability of the editor can be used to enter a parameter item. The editor has the following features. The type-in point is set with the mouse. Text can be deleted by selecting it with the mouse and hitting the DELETE key. Selected text can be moved or copied anywhere on the screen, even to

other windows, with the MOVE and COPY keys. (MOVE, COPY and DELETE are special function keys on the extended keyboard.) Other special function keys find and replace text and paste deleted text.

The five DEBUG tools supported on the workstation are described in the remainder of this chapter. (See Appendix A for a detailed description of the tools.)

4.3 The Configuration Tool

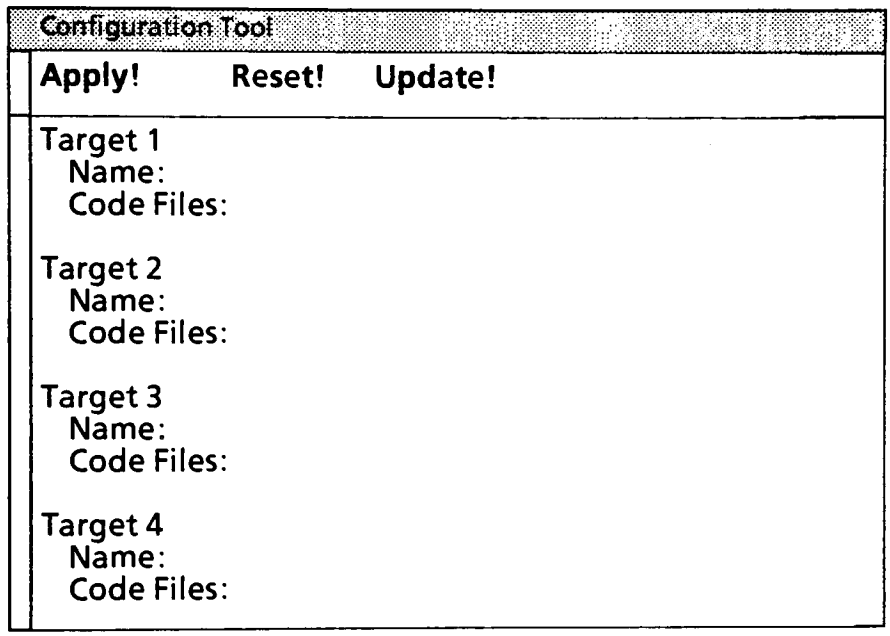

Figure 3. Configuration Tool window

Before debugging can begin, the run time executives and the application programs must be loaded into the target microprocessors. The debugger queries the MDM for the number and types of target processors. It then brings up a window for the Configuration Tool listing all the target processors. Figure 3 illustrates the Configuration Tool. This tool is used to name one or more processors in the target system and load them with the appropriate code files. Once the configuring is complete, the user may destroy the tool window and free up space on the display. Irrespective of the window status, the configuration is saved on the workstation. The user can re-activate the tool at anytime to load additional code files or newer versions of already loaded code files. Only the affected files are reloaded.

The configuration tool provides a compact way to load multiple files into multiple targets using a single command. The list of files loaded into a target is available for reference anytime during debugging. Incremental loading of code files makes it easier and faster to test source level changes.

4.4 Control of multiple processes on multiple processors

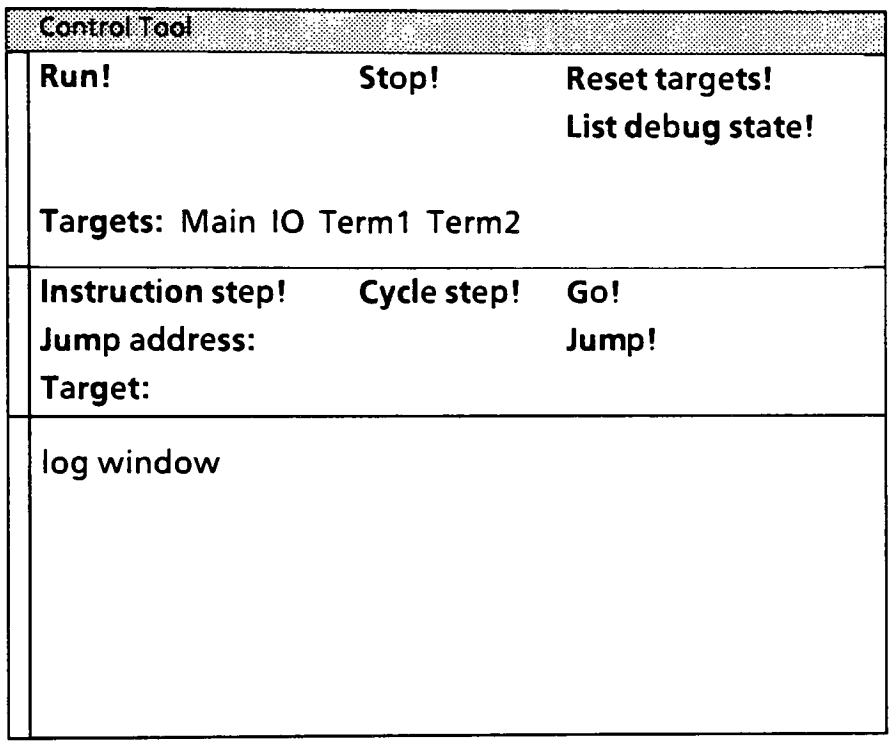

Figure 4. Control Tool window

Once the target microprocessors are loaded with the program files, the user brings up a second debugger tool, the Control Tool, to start program execution. This tool, illustrated in figure 4, contains commands to control the execution of programs in the targets, that is, to start and halt the processors, to reset them and to query their status.

These commands can be directed at multiple targets by selecting the appropriate ones from the list displayed in the window. When multiple targets are involved, the commands are sent to targets one by one. Since the targets are loosely coupled, there is no need to synchronize the starting or halting of the programs. The reset command is equivalent to a soft boot where task queues are flushed and run time

execs are re-initailized. The list command displays the run state, breakpoints and traces information.

The middle subwindow of this tool has commands to control program execution on a single target processor. These include commands to instruction step or cycle step through a program and to jump to a user specified location. A jump location can be specified by loading a source file in a file window and pointing to the source statement. The bottom window of the tool maintains a central log of all debug activities.

The Control Tool, in its current level of implementation, has basic commands for controlling program execution on multiple processors. The user can drive multiple targets with a single user command. Jump locations can be specified at the source level. These features improve the speed of debug.

4.5 Setting breakpoints

Breakpoints can be set on the targets, anytime during debug, using the Breakpoint Tool. When a target reaches a breakpoint only that processor is halted. Each processor in the debug configuration can have one Breakpoint Tool window. Figure 5 illustrates the Breakpoint Tool window for a target named Main. The debug hardware module, connected to a target, determines the number and types of breakpoints available for that target. In the current implementation there are one each of two types of breakpoints available for each target.

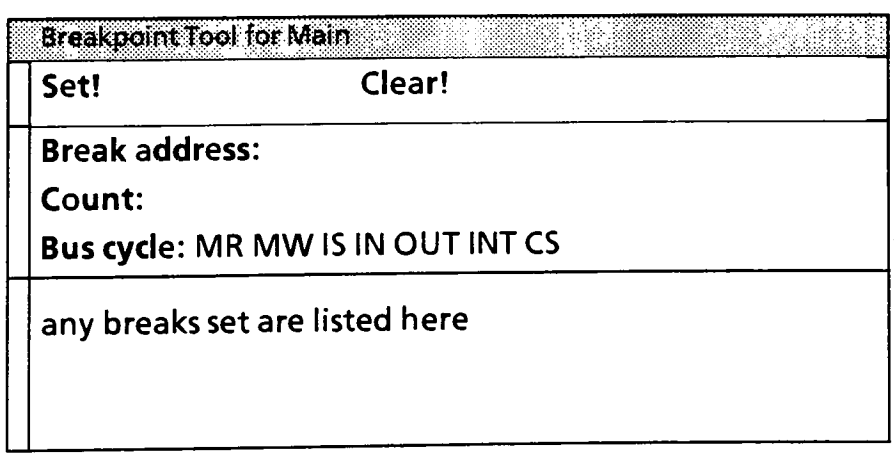

Figure 5. Breakpoint Tool window

One breakpoint is specified by a location, a count and a set of bus cycles as inputs. The break occurs when the location is accessed the number of times specified by the count, with a bus cycle from the specified set, for example: break when the data stored in location #AA00 is accessed 3 times with an instruction fetch bus cycle. The second breakpoint uses a range of locations and a set of bus cycles (as parameters) to effect a break when any memory location within the range is accessed with a bus cycle from the specified set, for example: break when any location between #AA00 and #AAFF is accessed with a memory write bus cycle.

The two types of breakpoints are useful in a variety of debug situations. They can be used to monitor IO ports, memory, interrupts and program execution.

4.6 Tracing

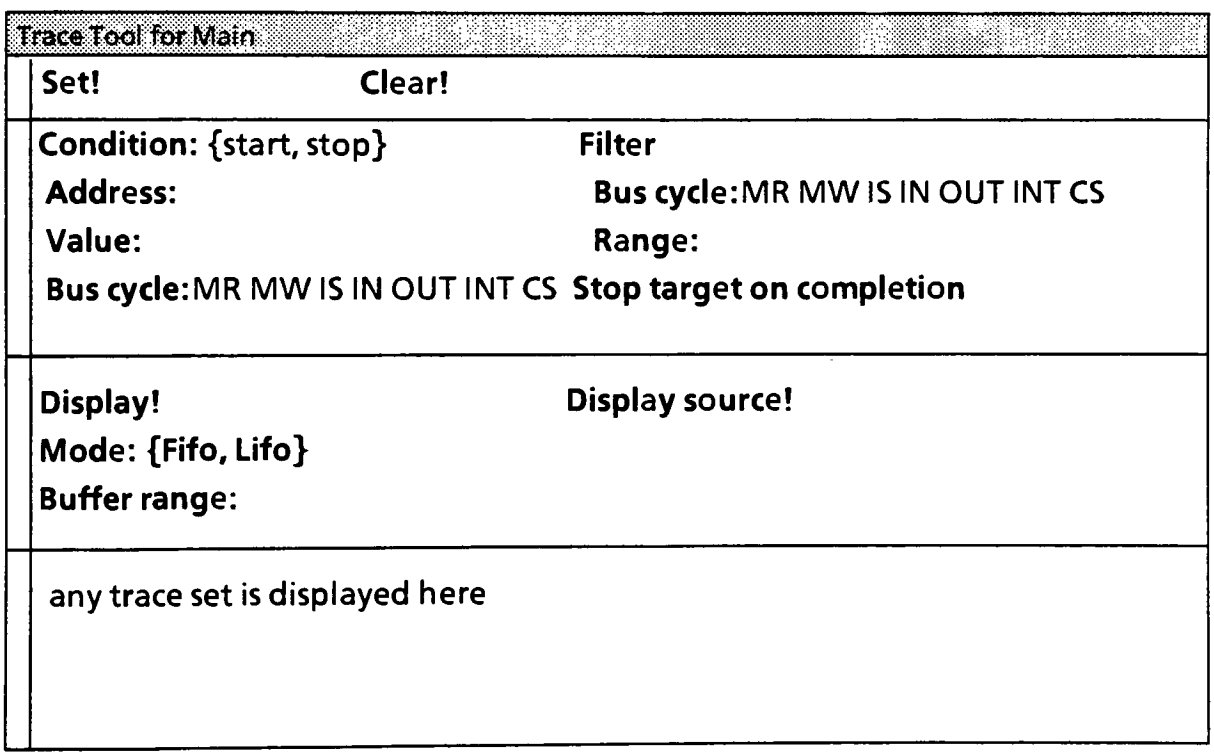

Figure 6. Trace Tool window

The debug hardware modules, connected to each target processor, have trace capability built into them. The user accesses a module's trace feature by bringing up the Trace Tool window for the connected target processor. The Trace Tool window

for a target named Main is shown in Figure 6. Each target in the debug configuration can have a Trace Tool window. Traces can be set on multiple targets using the target specific Trace Tool windows.

The user can specify a start or a stop trace condition. That is, if the user specifies a start condition the tracing starts when the condition becomes true and stops when the buffers are full or the processor stops. For a stop condition, tracing begins immediately and continues until the condition becomes true.

Trace conditions include three parameters: an address, a data value and a set of one or more types of bus cycles. The condition becomes true when the address is accessed with the optionally specified value and one of the specified bus cycle types. For example: if 4344, 128 and {MW, MR} are the parameters for the condition, the condition is satisfied when the value 128 is read from or written to the memory location 4344. It is also possible to specify a filter condition for the data collected in the debug module trace buffer. Trace entries can be restricted to those for a specified memory range and set of bus cycles. Upon completion of a trace, the target may stop or continue to run depending upon user specification.

Once tracing is completed, the debug module trace buffer can be displayed on the Control Tool log window as raw hex data, assembly language pnemonics or high level symbolic statements. Raw hex data can be displayed LIFO or FIFO. In all cases, segments of the trace buffer can be displayed by specifying address ranges. When displaying the the buffer in the high level symbolic format, the source files for the affected modules are loaded into file windows and the statements traced are highlighted. This provides a very concise picture of the execution path.

Tracing is specially useful in analyzing timing related bugs. Utilizing the flexibility of the start and stop conditions and the filter feature, trace specifications can be tailored for each problem. The display of the trace buffer as high level source statements enhances the utility of tracing.

4.7 Display and patching

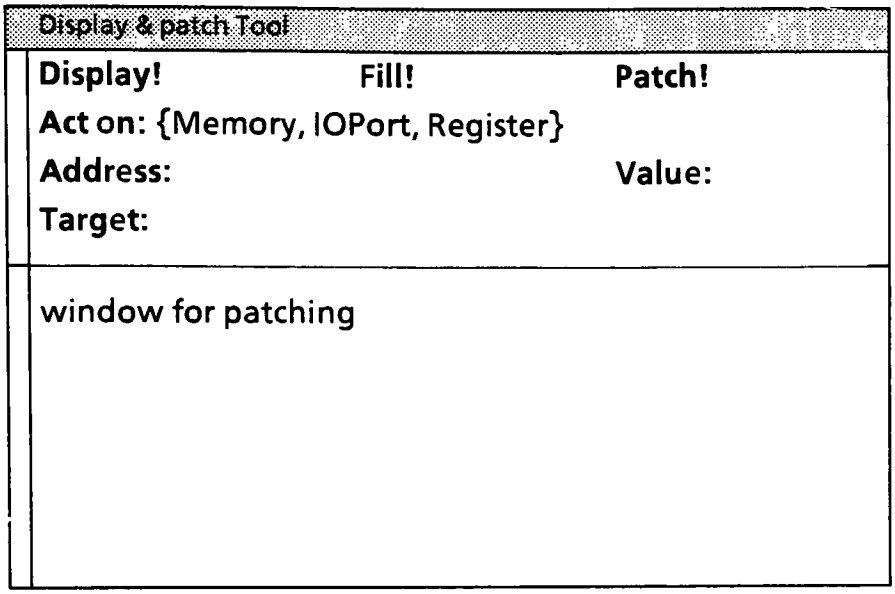

Figure 7. Display & patch Tool window

The Display & Patch tool, shown in figure 7, is used to access the target memory, IO ports and registers. This tool creates only one window. Entering a target name, the type of display item (memory, IOport or register) and a list of single addresses or ranges, and selecting the display command, causes the values to be displayed in the bottom subwindow. The addresses and the displayed values may be high level symbol names. Patches are applied to a target by editing the contents of the bottom subwindow and selecting the patch command. A command to fill a range of memory with a byte value is also provided. For every command executed from this window, a detailed entry is made in the Control Tool log window.

The Display & Patch tool provides the basic capability to access the target processor memory, IO ports and registers.

4.8 Conclusion

DEBUG offers many features useful to a programmer debugging multiple networked microprocessor systems. The host workstation provides the ability to control the entire system from a single console, the friendly user interface, the high level debug capability and the parallel availability of other tools. The debug

hardware implements useful control and monitor functions like breakpoints and the trace capability. Debugging is entirely from the outside and does not take up time or memory on the target processors. For all these reasons, the functionality and flexibility of DEBUG approaches the ideal debugger described in Chapter 3.

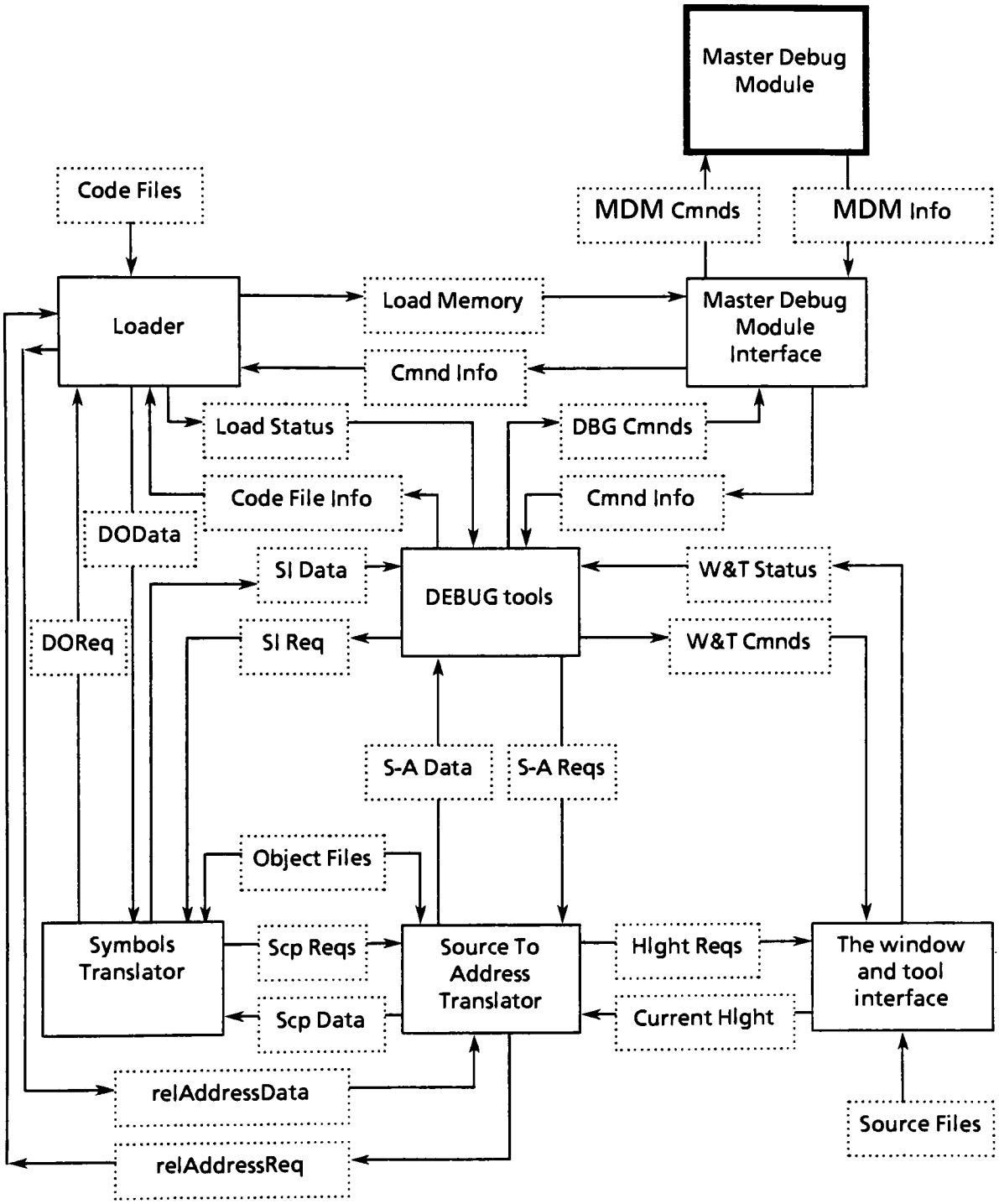

Figure 1. The top level data flow

the MDM commands.) The debug modules support a more extensive set of debugging commands issued by the workstation. These perform low level

the debug module for the selected target and (iii) request the debug module to set the break on the target. **breakRecord** specifies the type of breakpoint requested and their parameters. The debug module commands and parameters are different for the two types of breaks. Depending on the break type, the MDM Interface sends the appropriate break command and parameters to the MDM. When a breakpoint is hit on a target processor, the debug module sends a message containing the break address to the MDM. The MDM adds the debug module number to this message and forwards it to the workstation as a **MDMInfo**.

The window and tools interface: The window interface is a low level software package that manipulates a tree of workstation screen windows. The package is passive in that it creates no processes and allocates no storage. Its main function is to support a set of public procedures that create and destroy windows, arrange them on the screen and display data within them. Each window has an associated procedure that is used to display information within it. Clients of the window package supply these display procedures which, when called by the window package, repaint all or part of the window area. The window interface provides a few standard display procedures.

The tools interface is built on top of the window interface and supports the concept of a tool window. A tool window is a two level tree of windows. The top level is the single large rectangle that covers the entire tool window area. The second level is a set of multiple subwindows that horizontally divide the tool window. (All the figures in Appendix A are examples of tool windows.) The tools interface supports four types of subwindows: string, file, message and form.

String subwindows are used to enter textual information which are stored in memory as MESA strings. File subwindows are similar to string subwindows, except their contents are backed up by a disk file. Message subwindows are used to post messages to the user; text displayed as a message is stored as a fixed length string. There are three types of messages: information, warning and fatal. A form subwindow is a mechanism for specifying command parameters and invoking commands. It is made of the following types of form items: command, boolean, numeric, string and enumerated. (Appendix B.4 contains a block of pseudo code that illustrates the creation of a sample tool.) The data items, **W&T Cmnds** and **W&T Status**, in figure 1, represent procedure calls to the window interface to create, destroy and manipulate tool windows and the responses to these calls. A tool to

module obtains them from **the window and tools interface**. This is shown in figure 1 as **Current Hlght**. If the **S-A Reqs** is for converting from target to source address, the **source to address translator** calls **the window and tools interface** to highlight the translated source positions. This is shown in figure 1 as **Hlght Reqs**. The code addresses stored in the object and code files are not absolute target memory addresses. This module calls the loader to convert these addresses to absolute target locations, shown in figure 1 by the data items, **relAddressReq** and **relAddressData**. This module and correspondence tables are covered in detail in the later sections of this chapter.

The symbols translator: This module accesses the symbol and the correspondence tables, which are parts of object files, to determine the memory addresses, the type data and the lengths of variables declared in source modules. **SI Reqs** represent calls from the DEBUG tools for this service. **SI Data** is returned by this module in response to the requests. The variables available at a given time depend on the current scope of the target. This module calls **the source to address translator** to get the current scope information, as shown in figure 1 by the data items, **scpReq** and **scpData**. The data addresses stored in the object files are not absolute target memory addresses. This module calls the loader to convert these addresses to absolute target locations, shown in figure 1 by the data items, **DOReq** and **DOData**. This module and symbol and correspondence tables are covered in detail in the later sections of this chapter.

The five DEBUG tools constitute the user interface to DEBUG. Refer to chapter 4 for a functional description of each tool. The tools depend very heavily on the other modules previously described. Figure 1 illustrates these dependencies. There is very minimal dependence between the tools themselves, as can be seen in figure 3 which is the second level data flow diagram for the **DEBUG tools** block from figure 1. The dotted lines with directional arrows show data flow to the other modules in figure 1. **ProcData** is an array of target names and the identity of the debug module connected to each of them. **ProcData** is maintained by Control Tool and used by all the other tools for directing commands to the targets. The Configuration Tool updates **ProcData** whenever the user adds or deletes targets from the current debug configuartion. The only other type of data shared between the tools is log messages. The data items **Config Log, D&P Log,** Trace Log and **Break Log** are calls, by the other tools, to the Control Tool for posting log messages. A log message is generated for each DEBUG tool level command and contains all the information

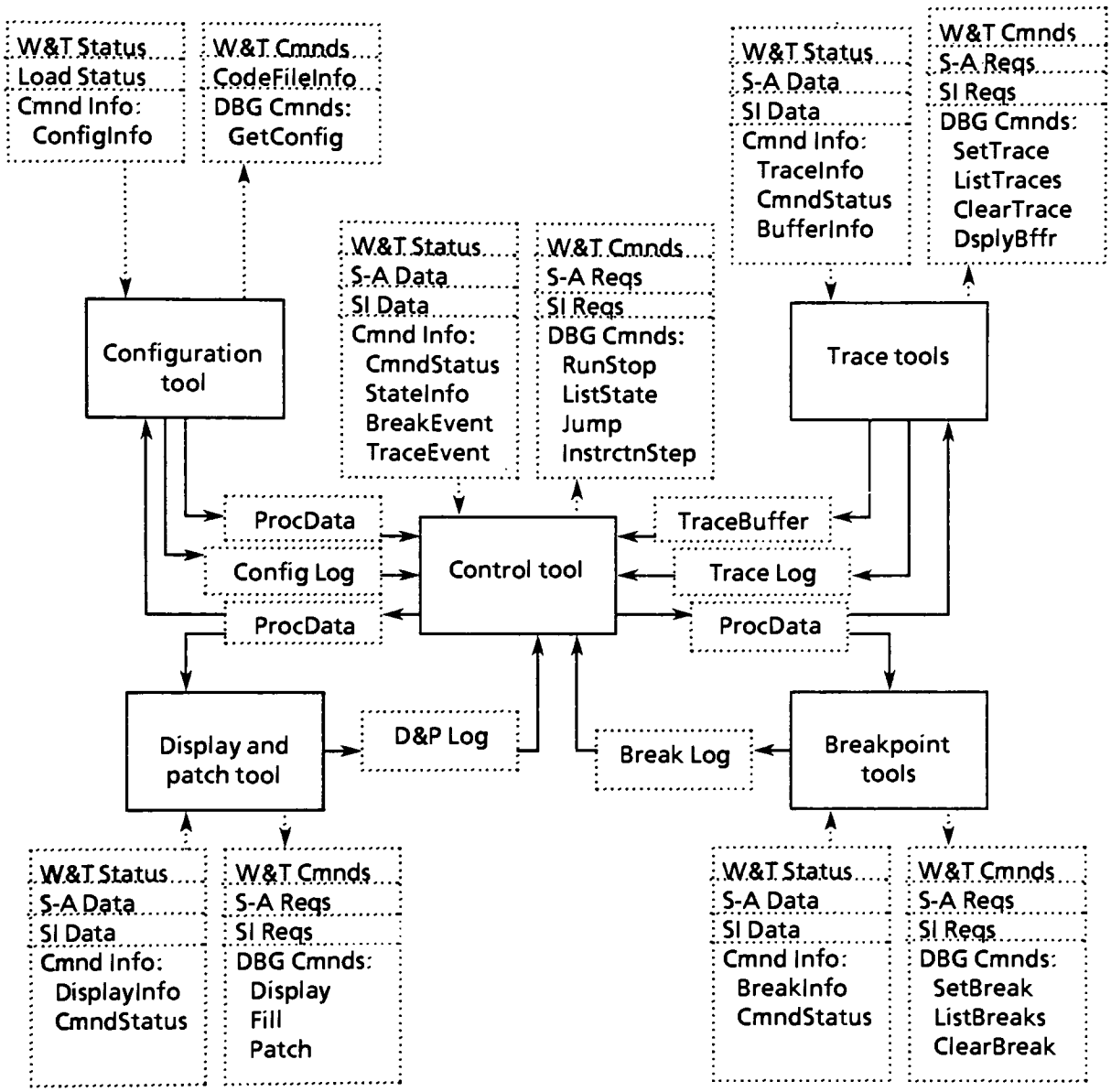

Figure 3. Second level data flow for DEBUG tools

necessary to repeat the command. **Trace Buffer** is obtained from the MDM Interface by the Trace Tool and displayed in the Control Tool log window. The five tools call procedures defined in the MDM Interface to execute debug commands. The **DBG Cmnds** and **Cmnd Info** data groups, in figure 3, list the data flow between each tool and the MDM Interface.

5.2 Compiler generated tables

Figure 4 is an overview of the object file generated by the compiler. The header

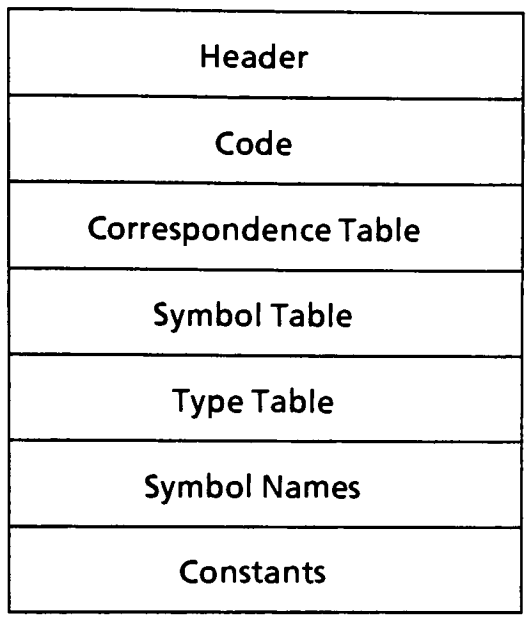

Figure 4. The object file format

contains the name and create time of the source file, the names and create times of all the interface files referenced in the object file and pointers to all the other sections of the file. Interface files are the templates for the data and procedures shared between multiple modules. They are object files that do not contain any code, only type definitions. Except in very special cases involving non-relocatable code, the compiler output contains relative memory addresses; therefore, the code is ready to be loaded into the target using the loader. Optionally, it can be input to the binder along with other object files and/or binder output files whose code parts are combined into a a single output code file, shown in figure 5. The code file has a table with the names and create dates of all the input files and their code and symbol offsets in the combined output file.

The loader, which loads both the compiler and the binder output files into the target memory, maintains a table of absolute load addresses for the targets. Relative addresses in an object file are converted to absolute memory locations by adding the target load address and one or more offsets to them. That is, if the

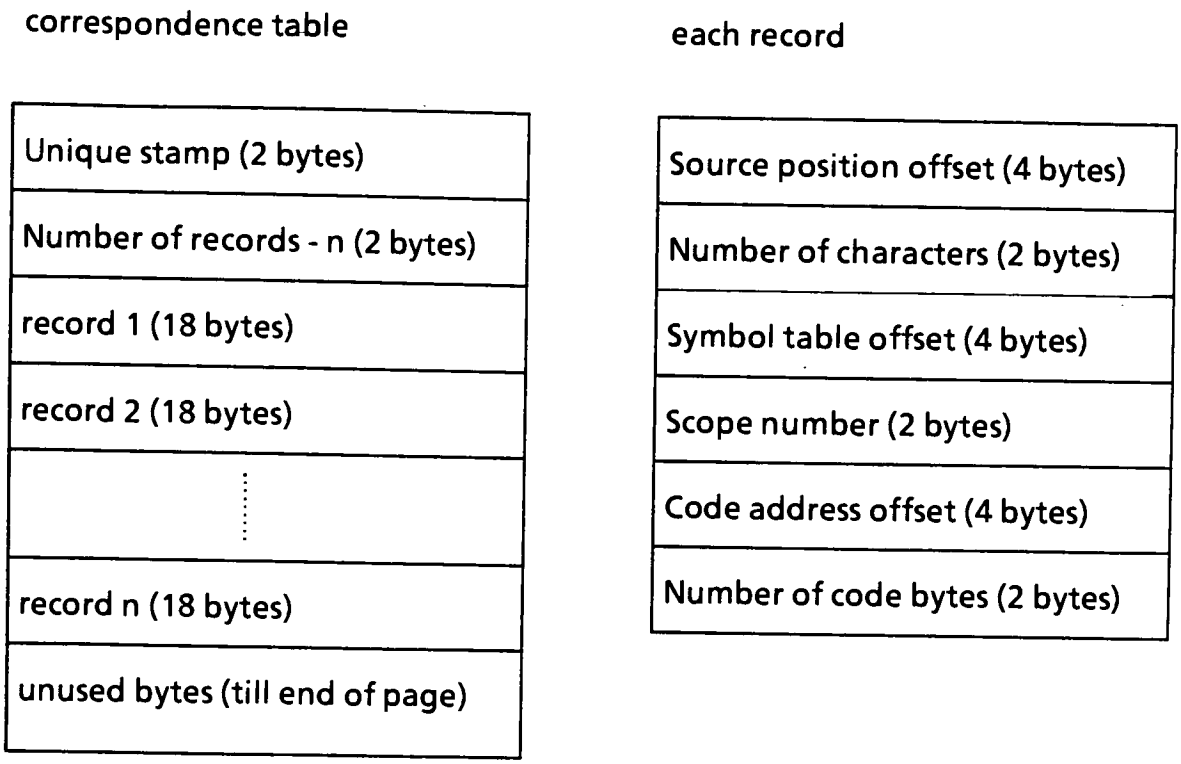

Figure 6. The correspondence table

section of the object file, is an unique number used to identify the section. There is one correspondence record for each high level source statement.

Each correspondence record is structured as follows:

1. The *source position offset* is the byte position of the first character in the statement in the source file.
2. The *number of characters* is the length of the source statement.
3. The *symbol table offset* is a pointer to the symbol table entry for the closest enclosing scope. That is, accessing the symbol table with this offset will yield the symbol record that is valid for the correspondence record's source statement. The language uses static scoping rules identical to Pascal.
4. Each scope in a source file is has an unique *scope number*. This is redundant data and is used for debugging DEBUG.
5. The *code address offset* is the relative address of the first byte of the compiled code.

6. The *number of code bytes* is the number of code bytes generated for the source statement.

5.2.2 The symbol table

The symbol table is a part of the object file generated by the compiler and contains information for all the variables and constants declared in a module. The symbol table addresses are relative and have to be adjusted with the loader and / or binder offsets to yield the machine address. The **Symbols Translator** does this conversion.

The current *scope* of a target is the logical code block that is being executed and the variables available at that time. The high level language used to implement the target programs is identical to PASCAL in defining blocks of code and static scoping of variables declared in these blocks. Therefore, there are global variables and levels of local variables. The same variable name can be declared several times in a source module, defining different data items, as long as each declaration is in a different block. A local variable is defined only when the code being executed is within its declaring block and accessible when not blocked by a variable with the same name declared at a lower level. The default scope for all modules is the global scope. That is, all the variables declared globally are always defined and accessible, if not blocked. Whenever a target processor halts, because of breakpoints, trace triggers or user command, DEBUG automatically sets the scope to the source module and the code block where execution has halted.

The symbol table is used, at debug time, for three types of operations: (i) given a variable name, a source module name and a scope, display the current value of the variable in a high level form (ii) given a variable name, a source module name, a scope and a new high level value for the variable, store the appropriate data in the appropriate memory location and (iii) given a source module name and a scope, list all the variables declared in that scope along with their current values.

The symbol table, figure 7, is made up of multiple scope records. There is one scope record for each source file code block. This is a variable length record and contains information about all the symbols defined in that scope. Each scope record maintains a pointer to its next higher enclosing scope record.

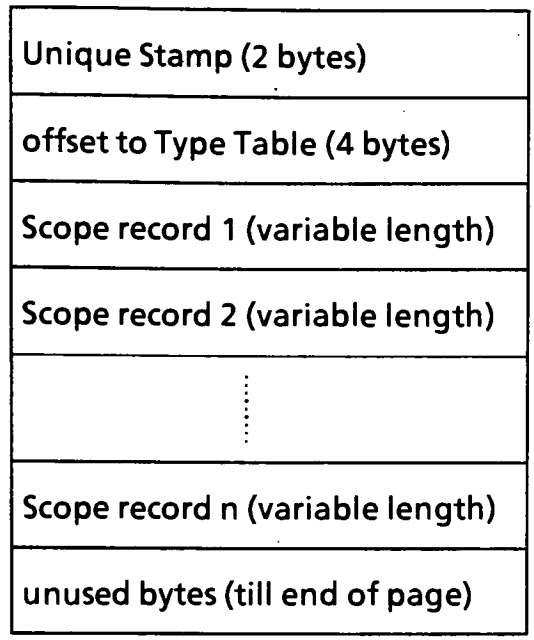

Figure 7. The symbol table

scope record:

Scope number (2 bytes)
Enclosing scope offset (4 bytes)
Number of symbols in scope - n (2 bytes)
Symbol record 1 (16 bytes)
⋮
Symbol record n (16 bytes)

symbol record:

Symbol name offset (4 bytes)
Type of symbol (2 bytes)
Type table offset (4 bytes)
Code address offset (4 bytes)
Number of code bytes (2 bytes)

Figure 8. The scope record

The fields of a scope record, shown in figure 8, are:

1. *Scope number* is used to identify the scope. It is stored in each correspondence table entry. This number is used for debugging DEBUG.

2. *Enclosing scope offset* is the pointer to the scope record for the next higher offset. This is used to search for variables in enclosing scopes.
3. *Number of symbols in scope* is a count of symbol records in the scope record.

Each symbol record of a scope record contains:
1. *Symbol name offset* is a pointer into the symbol names part of the object file where the actual variable names are stored, in alphabetic order.
2. *Type of symbol* is a number indicating the type of a variable. Each compiler defined data type is given an unique number. All user-defined data types are assigned the same unique number.
3. *Type table offset* is a pointer into the type table part of the object file where the actual declaration for the symbol type is stored. This field is only meaningful for user-defined types.
4. *Code address offset* is the relative location, in the object module, of the variable. This number, adjusted with the loader and/or binder offsets, determines the absolute memory address of the variable. For symbols declared constants, this field is the offset into the constants section of the object file.
5. *Number of code bytes* gives the size of the variable or constant.

5.3 Using the correspondence and symbol tables

The correspondence and the symbol tables are used frequently in DEBUG tools. For example, the Breakpoint Tool uses the correspondence table while setting a source level breakpoint. The Display & Patch Tool uses both tables to find the address and type of a variable. Access to these tables is through the **Source To Address Translator** and the **Symbols Translator**. The public procedures provided by these modules, to access the tables described in section 5.2, are covered here. Mesa language constructs are used to define and describe these procedures. Words with all upper case characters are Mesa keywords. Procedure names and type names begin with upper case letters while data variables begin with lower case letters.

Figure 9 defines the interface to access correspondence tables. **GetSourceInfo** accepts a memory address, **startAddress**, and a target number, **procNo**, as input and determines the source file name, **sourceName**, and the beginning and ending positions, **startPos** and **endPos**, of the corresponding source statement. **GetCodeAddress** accepts a source file name, **sourceName**, and a source position

range, **startPos** and **endPos**, and determines the corresponding absolute memory address range, **startAddress** and **endAddress**. **GetCodeAddressFromHighlight** is called when the user loads a source file into a window, moves the cursor to that window, clicks the mouse over one or more statements and issues a DEBUG command requiring a memory address. **GetCodeAddressFromHighlight** calls the window package to determine the source module name and the start and end position for the highlighted area. It then performs the source to target address translation for **procNo** and returns **startAddress** and **endAddress**.

```
GetSourceInfo: PROC[procNo: CARDINAL, startAddress:
  LONG CARDINAL, highLight: BOOLEAN] RETURNS [found:
  BOOLEAN, sourceName: STRING, startPos, endPos,
  endAddress, symbolTableOffset: LONG CARDINAL];

GetCodeAddress: PROC[procNo: CARDINAL, sourceName:
STRING,
  startPos, endPos: LONG CARDINAL] RETURNS
[startAddress,
  endAddress: LONG CARDINAL];
```

Figure 9. Procedures to access the correspondence table

Figure 10 defines the interface to access symbol tables. **GetAllSymbols** accepts a target number, **procNo**, and an absolute memory address, **addressToDefineScope**, that is used to determine the scope and returns a linked list, **symbolInfo**, of all the variables declared in that scope. **GetSymbolInfo** accepts a symbol name, **symbolName**, in addition to the target number, **procNo**, and memory address, **addressToDefineScope**, and returns the type and memory address and length of the symbol as **symbolInfo**. **FormatSymbol** uses the type information from the **symbolInfo** and translates an array of bytes, **byteStream**, into a string representing the value of the variable in high level format, **formatted**. It is used whenever the user requests a display of the current value of a variable. **DeFormatSymbol** does the opposite. It parses and converts a string, **formatted**, to an array of bytes, **byteStream**, and is used when the user modifies a variable using DEBUG.

The following are algorithms for the two basic procedures defined in figures 9 and 10. The first, figure 11, accesses the correspondence table. The second, figure 12, accesses both the tables to get symbol information.

```
GetAllSymbols: PROC[procNo: CARDINAL,
addressToDefineScope:
  LONG CARDINAL] RETURNS [symbolInfo: SymbolInfo];
GetSymbolInfo: PROC[procNo: CARDINAL,
addressToDefineScope:
  LONG CARDINAL, symbolName: STRING] RETURNS [found:
  BOOLEAN, symbolInfo: SymbolInfo];
FormatSymbol: PROC[symbolInfo: SymbolInfo, byteStream:
  ByteStream] RETURNS [formatted: STRING];
```

Figure 10. The procedures to access the symbol table

Given a target processor number and a hardware memory address on the target, **GetSourceInfo** determines the corresponding source file name and high level language statement. A procedure implemented by the **Loader** is called by the **Source to Address Translator** to determine the code file that is loaded in the specified memory address and convert that address to a relative address with respect to the code file. **RelAddressReq** and **RelAddressData** in figure 1 refer to this call to the loader. The code files are accessed to determine the source file name and the object file. This is done by converting the relative address, from an offset into the code file, to an offset into the object file. The correspondence table that is a part of the object file is then searched serially until the object file offset matches the code address range of a correspondence record. If the **highLight** boolean is true, a procedure call is made to the workstation file editor interface to highlight the appropriate high level statement. **Hlght Reqs** in figure 1 represents this action.

GetSymbolInfo uses **procNo** and **addressToDefineScope** to search the symbol table for **symbolName** and returns the symbol information as **symbolInfo**. **addressToDefineScope** is an absolute memory address that is used to define the current scope of the target. **GetSourceInfo** is called with **procNo** and **addressToDefineScope** to identify the source file name, **sourceName**, and the symbol table offset, **symbolTableOffset**. **ScpReq** and **ScpData** in figure 1 represent this. **GetSymbolInfo** uses this offset to access the symbol table scope record for the current scope given by **addressToDefineScope**. Each symbol record belonging to the scope record is searched serially until the symbol names match. If a match is not found successivlely higher scope records are searched. The symbol record provides an offset, the type and length information. A call is made to the Loader to convert

```
GetSourceInfo: PROC[procNo: CARDINAL,
  startAddress:LONG CARDINAL, highLight: BOOLEAN]
  RETURNS[found: BOOLEAN, sourceName: STRING,
  startPos, endPos, endAddress,
  symbolTableOffset: LONG CARDINAL] =
  BEGIN
  cRec: CorrRec;  endOfTable: BOOLEAN;
  codeFile, objectFile: File;
  relativeStartAddress: LONG CARDINAL;
  found ← FALSE;
  [codeFile, relativeStartAddress] ←
    Loader.GetName [procNo, startAddress];
  IF IsFileBound[codeFile] THEN
    [sourceName, relativeStartAddress] ←
     GetName[codeFile, relativeStartAddress]
  ELSE sourceName ← codeFile.name;
  objectFile ← StartCorrAccess[sourceName];
  [cRec, endOfTable] ← ReadCorrRecord[objectFile];
  UNTIL endOfTable OR
    cRec.startAddress <= relativeStartAddress <
    cRec.startAddress + cRec.length DO
    [cRec, done] ← ReadCorrRecord[objectFile];
    ENDLOOP;
  StopCorrAccess[objectFile];
  found ← NOT endOfTable;
  IF found THEN {
    startPos ← cRec.sourcePos;
    endPos ← cRec.sourcePos + cRec.numChars
    endAddress ← cRec.startAddress + cRec.length;
    symbolTableOffset ← cRec.symbolTableOffset;
    IF highLight THEN
      FileEditor.Highlight[sourceName, startPos,
endPos]};
```

Figure 11. Algorithm for accessing the correspondence table

the offset to an absolute memory location, in the data area of the target, where the symbol is stored.

The symbol information returned by **GetSymbolInfo** is used in two ways. When the current value of a symbol is needed, the address and length information returned is used to to access the contents of target memory; the raw memory contents are formatted into high level form by the procedure **FormatSymbol**. When the current value of a symbol in target memory is to be modified with a new value, specified in high level form, the symbol information returned by **GetSymbolInfo** is used to store the value after converting it to raw memory contents with **DeFormatSymbol**.

```
GetSymbolInfo: PROC[procNo: CARDINAL,
  addressToDefineScope: LONG CARDINAL, symbolName:
STRING]
  RETURNS[found: BOOLEAN, symbolInfo: SymbolInfo] =
  BEGIN
  endOfTable: BOOLEAN;  sourceName: STRING;
  symbolTableOffset, nextEnclosingOffset: LONG CARDINAL;
  absSymbolOffset: LONG CARDINAL;
  objectFile: File;
  [found, sourceName, , , , symbolTableOffset] ←
    GetSourceInfo[procNo. addressToDefineScope, FALSE];
  objectFile ← StartSymbolAccess[sourceName];
  nextEnclosingOffset ← ReadScopeRecord[objectFile,
    symbolTableOffset];
  UNTIL symbolTableOffset = 0 DO
    [symbolInfo, endOfTable] ←
      ReadSymbolRecord[objectFile];
    UNTIL endOfTable OR symbolInfo.symbolName =
symbolName
      DO
      [symbolInfo, endOfTable] ←
        ReadSymbolRecord[objectFile];
      ENDLOOP;
    IF endOfTable THEN {
      symbolTableOffset ← nextEnclosingOffset;
      nextEnclosingOffset ← ReadScopeRecord[objectFile,
        symbolTableOffset]};
    ELSE {found ← TRUE; EXIT};
    ENDLOOP;
  StopSymbolAccess[objectFile];
  IF found THEN{
    absSymbolOffset ← Loader.GetSymbolOffset[procNo,
      objectFile];
    symbolInfo.startAddress ←
      symbolInfo.startAddress + absSymbolOffset;
    symbolInfo.stopAddress ←
      symbolInfo.stopAddress + absSymbolOffset};
```

Figure 12. Algorithm for accessing the symbol table

5.4 Conclusion

This chapter presents the system design for DEBUG. Data flow diagrams describe the major components of the tool, their functions and the interactions between them. It stresses the high level features of the tool by describing the mechanisms for high level source access. The formats for the code and object files and the correspondence and symbol tables used to provide high level access are

documented. Mesa code is used to describe the algorithms for searching these tables.

6. Conclusion

Real-time software development in networked multi-microprocessor systems is made easier and more efficient by using sophisticated tools. One such tool is a suitable high level language that supports multi-tasking, IO processing, modularization and exception handling. Programming in a high level language provides a level of abstraction between the user and the microprocessor.

The debugger should preserve this abstraction by (i) allowing the use of symbols defined in the high level language program and (ii) by providing source line to machine address translation for branch instruction, breakpoints and such. In addition the user should be able to control and monitor the target processors from a single operator console. A window environment with a mouse and menu selection capability provides a friendly, easy to learn user-interface for the tools.

DEBUG, the debugger described in this report, satisfies all the above requirements. The debugger also has features to load the processors with the necessary code, to control the execution of code, to display and modify target memory, registers and IO ports and to set up breakpoints and trace events on the target processors. The user is allowed to work in the symbolic mode while using all these commands. Even data from the trace buffer is displayed in the form of source level statements.

The one major shortcoming of DEBUG is the absence of a replay or command file execution capability. It would be very useful to either automatically or manually record a series of user actions like mouse clicks, menu selections and type-ins and replay them later on user request. The operating system of the workstation on which DEBUG runs does not support this feature.

There are three approaches to implementing such a feature, each with its own set of complications. The command or replay file could record all low level inputs, that is, all type-ins and absolute display coordinates for mouse clicks. This requires both the recording and the replaying windows to be laid out with exactly the same size, position and layering. Another approach is for the replay mechanism to record at a higher level. It could record the activation of tools, changing of parameter values and the execution of commands. For this to work, the tools and the replay mechanism have to interact - the replay mechanism requires the names of the tools,

commands and parameters that were affected and, in turn, should provide these as input to the tools. A third alternative is to implement an independent TTY type serial command parser for each tool. That is, each tool will have two user interfaces: a window driven one for interactive use and a serial, TTY type for batch use. This has the disadvantage of forcing tool level changes and, in addition, the user has to learn the TTY syntax of each command.

Each DEBUG tool can be made more useful by the following tool specific enhancements:

The configuration tool should to be expanded to include patch management. Two new window commands are needed: (i) to save target patches as files on the workstation and (ii) to load these patch files from the workstation to the target. A command to compare the target processor memory with a code file would also be useful.

The control tool, in its current level of implementation has basic commands for controlling program execution on multiple processors. Some possible enhancements are: (i) an interpreter that allows a user to execute procedures on the targets and displays the return values, (ii) some high level control of the task queues including the ability to change task priorities and to activate suspended tasks and (iii) the source level display of the execution stack upon reaching a breakpoint.

The two breakpoints provided by the breakpoint tool are sometimes not enough to debug even moderately difficult problems. The number of breakpoints allowed should be increased, as well as the kinds of the break conditions. The ability to break at a location, only if other specified procedures are in the call stack, in a specified order, would be very useful. Another possibility is cross processor breaks, where a condition set on one target breaks or stops program execution on another target. Currently, DEBUG does not support the automatic execution of a set of pre-specified commands when a target reaches a breakpoint. This feature is only possible when DEBUG acquires some form of command file capability.

The tracing feature could be made more flexible and effective by increasing the size of the trace buffer and providing more and better trigger and filter conditions. The former requires adding more memory on the debug module. The later requires

In summation, DEBUG is a part of a software development environment for real-time, networked multi-microprocessor systems. It provides centralized, high level control and monitoring of software running on such systems. Many features of DEBUG, for example, the non-invasive debugging, the high level language access and tracing, approach the "ideal" debugger described in Chapter 3. It is a very useful and effective tool in its current level of implementation. Incorporating some of the suggested enhancements will increase its power and effectiveness.

Bibliography

Conte, G. and Gregoretti, F. "Software development and debug aids for the mu multi microprocessor system" *Sixth European symposium on microprocessing and microprogramming*, 1980

Geschke, C.M. *et. al.* "Early experience with Mesa" *Comm. ACM* 20, 8, Aug. 1977, p540-553

Gregoretti, F. "An experimental real time kernel for a multi-microprocessor prototype" *Proceedings of the 3rd international conference on distributed computing*, 1982

Haney, W. R., Jr. "A multi-microprocessor utility and monitoring system for high-level debugging and testing" *Internal conference on communications*, IEEE, 1980

Jones, D.M. "Debugger alleviates realtime programming complications" *Computer Design*, Mar. 1985, p183-191

Lampson, B.W. and Redell, D.D. "Experience with processes and monitors in Mesa" *Comm. ACM* 23, 2, Feb. 1980, p105-117

Lesh, F. "Software development aids in distributed microprocessor systems" *Proceedings of the international symposium on mini and micro computers*, IEEE, 1978

Smith, E.T. "Debugging techniques for communicating loosely-coupled processes" *Ph. D. Thesis, University of Rochester*, 1981

Wirth, N. "MODULA: a programming language for modular multiprogramming" *Software practice and experience*, Mar 1977

Young, S.J. *Realtime languages - design and development*, Ellis Horwood Limited, 1981

Appendix A. DEBUG Users' Guide

DEBUG is implemented on a Xerox 8000 series workstation, running Pilot as the operating system. The operating system as well as DEBUG are written in Mesa. DEBUG is a system made up of five tools that share data between them. The tools are: (i) the Configuration Tool, used to specify the debug configuraion and load files, (ii) the Control Tool, used to control execution of code on the target processors, (iii) the Breakpoint Tool, used to set and clear breakpoints, (iv) the Trace Tool, used to set and clear traces and display traced data and (v) the Display and Patch Tool, used to examine and modify the contents of memory, ports and registers.

A.1 Configuration Tool

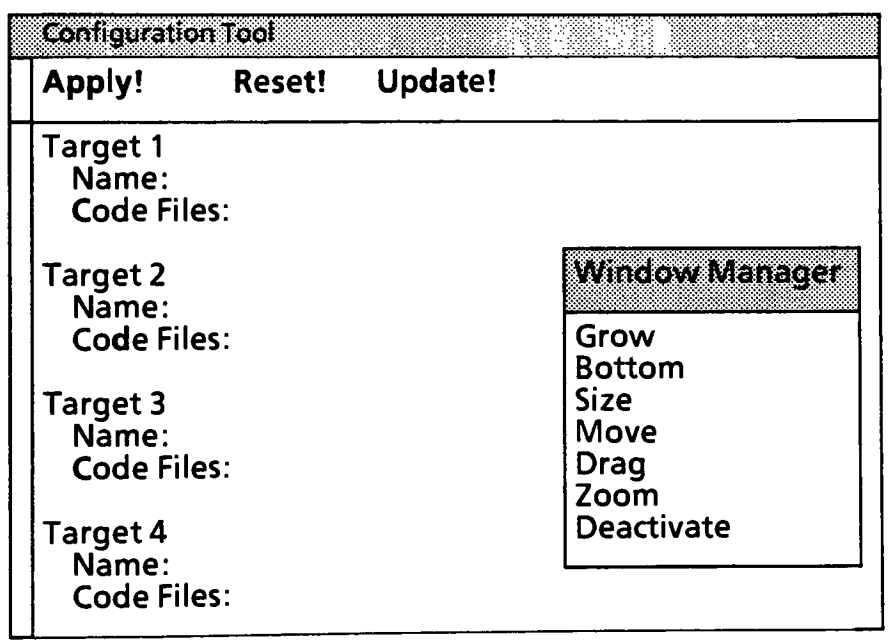

Figure 1. Configuration Tool window

This tool lists all the targets connected to the debug modules and is used to name some or all of them and load those named with code files. The user enters the **Name** and **Code Files** fields for the targets that are being used in the current debug session. The target names are used by the debugger to convey information to the user. The commands in this window are as follows.

Apply names the processors and loads them with the specified code files.

Update applies all changes, since the last Apply or Update, to the debug configuration. It loads a code file only if (a) the file was not loaded before or (b) the file was loaded but a new version of that file was found on the workstation.

Reset redisplays the parameter values in the window to reflect the actual current configuration, thus cleaning up any discrepancies on the screen.

The **Window Manager** menu is used to manipulate window size and position. This menu is made visible when the cursor is moved inside a window and both mouse buttons are held down.

A.2 Control Tool

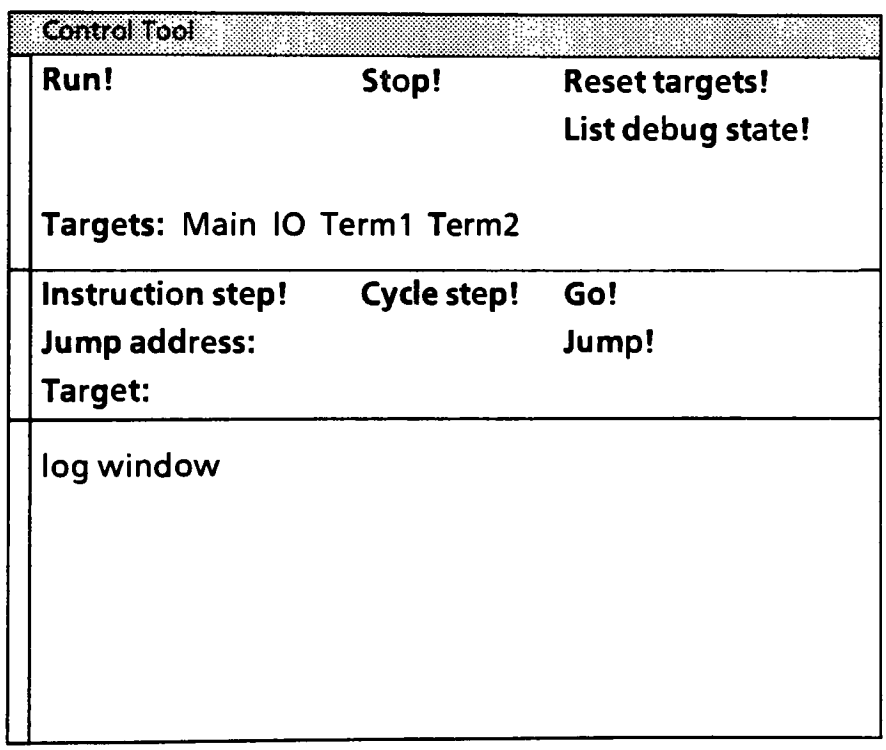

Figure 2. Control Tool window

This tool is used to control execution of code in the target processors. It has commands to start and stop one or more processors, to branch control to a given code location and to instruction step a target. The commands in the top subwindow work on multiple processors. All the named targets in the current configuration are

displayed in the **Targets** list from which the user selects one or more targets. The commands in the lower subwindow are applied to each selected target serially.

Run causes the targets to start or resume execution.

Stop halts the targets. The current state of the targets are saved, allowing the user to resume execution at a later time. The user can display memory and register values, apply patches and examine the trace buffers at this time.

Reset targets stops the targets and does a hardware reset on them. This is equivalent to a soft boot.

List debug state displays information in the bottom subwindow, about the selected processors. This includes information about the run/stop state of the processors and any breakpoints or traces set.

The commands in the middle window apply to a single target. The user enters the target name in the **Target** field of this subwindow.

Instruction step executes a single instruction on the specified target.

Cycle step executes a single machine cycle on the specified target.

Go is the same as **Run** for a single target.

Jump fills the program counter of the target with the value specified in the **Jump address** field. A subsequent **Go** command starts target execution at the new address.

The bottom subwindow is the central log for the entire debug process. All commands from the DEBUG tools and responses from the debug hardware are displayed here. This window is also backed up by a disk file. Therefore, the log can be scrolled at any time to the beginning of a debug session.

A.3 Breakpoint Tool

Each named target can have one Breakpoint Tool window which is used to set breakpoints. There can be at most two breakpoints set on a target. To set a breakpoint, the user fills in the parameters in the middle subwindow and clicks over **Set**. This sets the breakpoint on the debug hardware for the particular target processor and displays the break condition in the bottom subwindow.

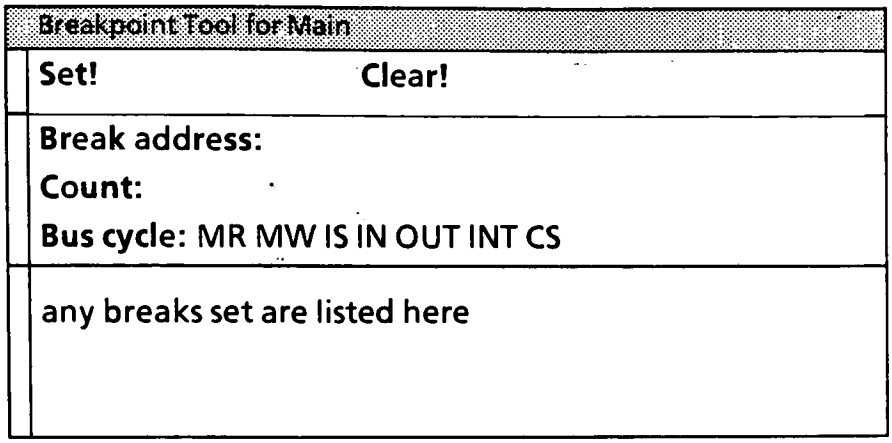

Figure 3. Breakpoint Tool window

To clear a breakpoint, the user selects the display for that breakpoint in the bottom subwindow and clicks **Clear**. The break is cleared from the debug hardware for the target processor, as well as from the display in the bottom subwindow.

The parameters for setting breakpoints are as follows.

Break address is a single address or a memory range. If it is a memory range, it starts and ends on a 256 byte block boundary. Two types of breaks are possible, one on a single address and the other on a range of memory. Two breakpoints are allowed, one of each type.

Count is valid only for the single address breakpoint. The default for this field is one.

Bus cycle is valid for both types of breaks. One or more of these can be selected. The default is all selected. The acronyms stand for:

MR	memory read
MW	memory write
IS	instruction step
IN	read from input port
OUT	write to output port
INT	interrupt
CS	cycle step

The two breakpoint types work as follows. The first type of break occurs when the single **Break address** is accessed **Count** number of times with a bus cycle from the

selected **Bus cycle** list. The second occurs when any location within the **Break address** range is accessed with a cycle from **Bus cycle**. **Count** is not valid for this type of breakpoint.

A.4 Trace Tool

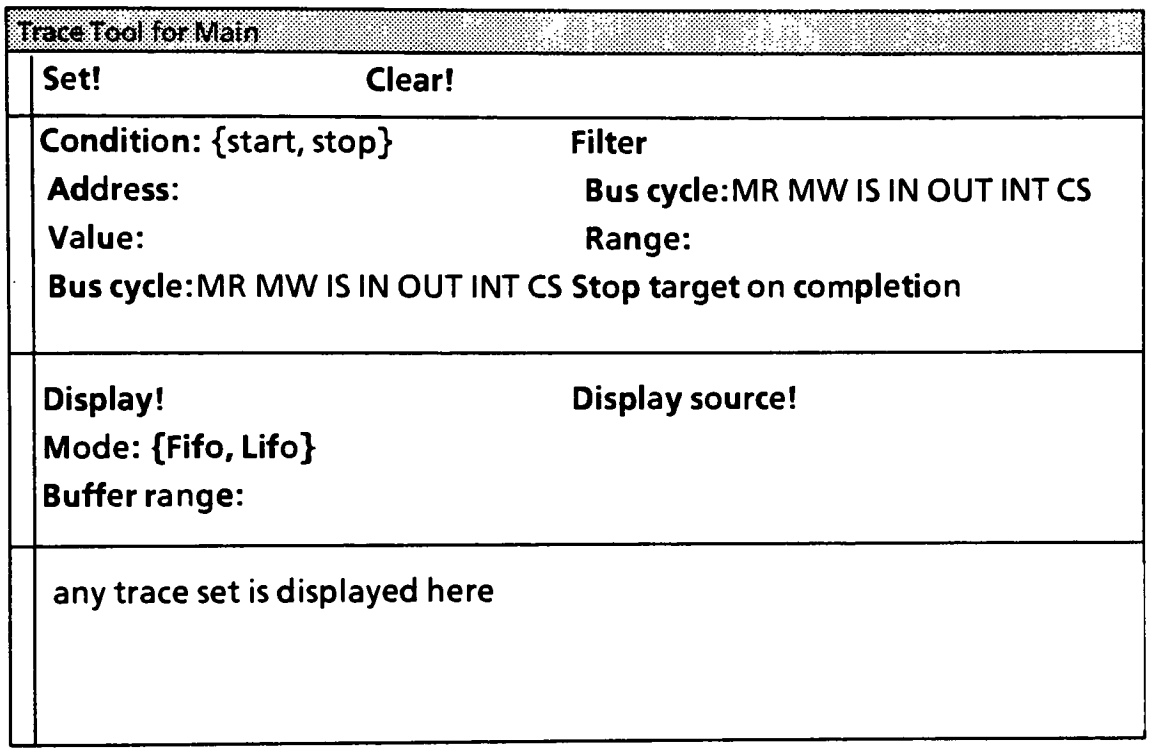

Figure 4. Trace Tool window

Each named target can have one Trace Tool window which is used to set traces. Each target may have at most one trace specification. To set a trace, the user fills in the parameters in the middle two subwindows and clicks **Set**. This sets the trace on the debug hardware for that target processor and displays the trace condition in the bottom subwindow. The trace command accepts two conditions. One determines when the trace should be started or stopped; and the other is a filter that controls the data collected in the trace buffer.

To clear a trace, the user clicks **Clear**. This clears the trace in the debug hardware for the target processor, as well as the display in the bottom subwindow.

The parameters for starting and stopping traces are entered in the next-to-top subwindow as follows.

If **Condition** is set to start, tracing begins when the condition is hit and continues until either the debug module trace buffer fills up or the processor stops. If it is set to stop, tracing begins as soon as the trace is set and continues until the processor stops or the condition is hit. If the buffer fills up before the condition is hit then trace data is lost on a first-in first-out basis.

Address is a single memory address.

Value is a number less than 256. This field may be left blank.

Bus cycle are bus cycle types selected by the user from the list displayed. The default is all selected.

The trace condition becomes true when the memory **Address** is accessed with the specified **Value** and a **Bus cycle** from the selected list. The status of the data bus is ignored if **Value** is left blank.

The parameters for filtering trace data, also in the next-to-top subwindow, are as follows.

Bus cycle are one or more cycles selected from the list displayed. The default is all selected.

Buffer **Range** is a range of memory locations that begins and ends on a 256 block boundary.

Only data satisfying the filter condition are entered in the debug module trace buffer. The filter condition is satisfied whenever any address within the **Range** specified is accessed with a **Bus cycle** from the selected list. For example, if the range is 2200 - 2300 in hexadecimal and the selected bus cycles are MR and MW then any instance of a memory read or write access between the addresses 2200 and 3300 is in the trace buffer.

The two bottom windows are used to display tracing information. **Display** displays the raw debug module trace buffer data, while **Display source** translates the trace buffer contents into source module statements, loads the affected source modules into windows and highlights the executed statements. The parameters in this subwindow are:

Mode, which indicates whether to display the trace data first-in first-out or last-in first-out.

Buffer range, which allows the user to selectively display a portion of the trace buffer by entering an address range before clicking the display commands. The default is the entire trace buffer.

A.5 Display & Patch Tool

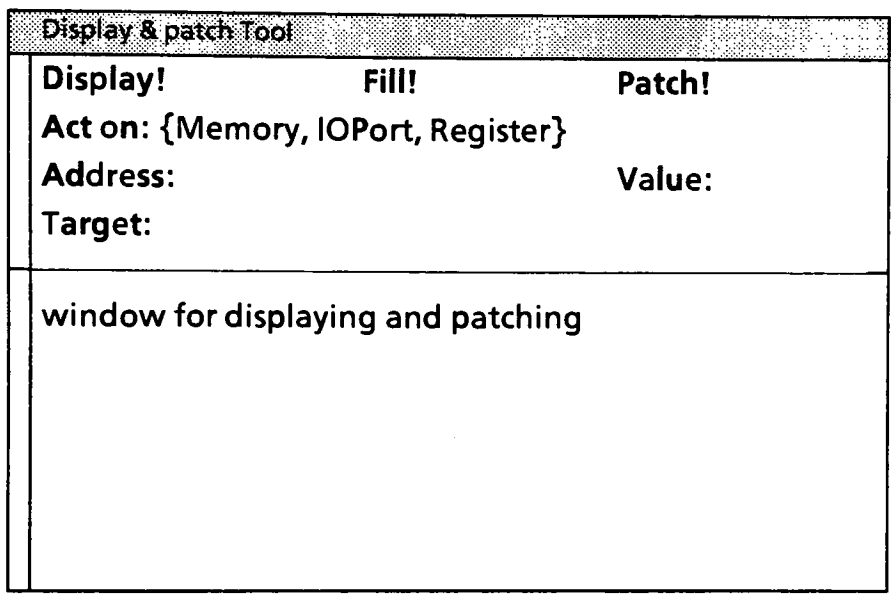

Figure 5. Display & patch Tool window

The Display & Patch Tool is used access and modify target memory, IO ports and registers. The parameters for this tool are:

Act on is a qualifier for **Address**, that determines if the address field refers to memory, input-output ports or processor registers.

Address, memory addresses, input or output port addresses or register names. The address can be a single or a range of locations. High level language symbol names are allowed.

Target is the target processor being debugged.

Value, a number less than 256 used by the **Fill!** command to initialize a range of memory.

The bottom window is used for displaying the current values of requested items and for patching. The **Display** command causes the current values of the requested items to be displayed in this subwindow. The **Fill** command is valid only for target memory and initializes a range of memory locations with a byte value. Patching is done by typing in one or more assignment statements in this window and then clicking the **Patch** command in the top subwindow. The **Patch** command interprets the contents of the bottom subwindow as a series of assignment statements and applies them to **Target**. It uses the **Act on** field to direct the patches. Sample assignment statements follow.

#3434 ← #FFFF
counter-2 ← counter-x
myrec.field2 ← dataPointer ↑ .field2

Appendix B. DEBUG Design Interfaces

B.1 The data dictionary

The names of data that are used in Chapter 5 are defined here. The items are listed in alphabetical order. Data items are sometimes defined using Mesa terminology. This is done only in cases where it improves the precision and clarity of the definition. For such items, the Mesa notation follows the data name and a semi-colon is used to indicate its termination. The Mesa notation is in a fixed pitch font. English language comments that follow explain the terminology. Character case of the dictionary entry names have no significance other than to improve readability.

Data

address: `LONG CARDINAL;` a target memory address
addressToDefineScope: `LONG CARDINAL;` an offset into target memory used to define the current scope
BreakEvent: a message sent by a debug module when the connected target satisfies a break condition.
BreakInfo: a record listing the number, type and parameters of currently active breakpoints set on a target.
BreakLog: messages about the setting and clearing of breakpoints, posted in the Control Tool log window.
BreakRecord:
```
RECORD[busStatesArray: BusStatesArray,
  SELECT type: BreakType FROM
    address => [address: LONG CARDINAL, count: LONG
CARDINAL],
    range => [startAddress, endAddress: LONG CARDINAL]];
```
a record used to describe a single address or a range breakpoint.
BufferInfo: the contents of the trace buffer for a target.
BusStates: `{memoryRead, memoryWrite, instructionStep, cycleStep, input, output, interrupt};` all the control bus values that are used for breakpoints and traces
BusStatesArray: `ARRAY BusStates OF BOOLEAN;` an array used to specify the selected bus states

Byte: `CARDINAL IN [0..255];` an eight bit quantity
ByteStream: `DESCRIPTOR FOR ARRAY of Byte;` a descriptor is a pointer to a variable length array.
ClearBreak: Remove a previously set breakpoint from a single target.
ClearTrace: Remove a previously set trace from a single target.
Cmnd Info: the data returned by the procedures defined in the MDM Interface. The type of data returned depends on the procedure. Most commands return a **Cmnd Status**. Others return command specific data.
Cmnd Status: `{success, abortedByUser, failure};` returned by all commands to the MDM interface.
Code file info: `RECORD[name: STRING, procNo: CARDINAL];` used to specify a code file name, **name**, and target number, **procNo**, to the Loader
ConfigInfo: a record with the number of debug modules and the type of each module.
ConfigLog: messages about changes to the debug configuration posted in the Control Tool log window.
count: `CARDINAL;` the number of times a break condition must be satisfied.
CorrRec: `RECORD[sourcePos: LONG CARDINAL, numChars: CARDINAL, symbolTableOffset: LONG CARDINAL, scopeNumber: CARDINAL, startAddress: LONG CARDINAL, length: CARDINAL];` a record that contains data about a source statement and its corresponding code
Current Hlght: `RECORD[sourceName: STRING, startPos, endPos: LONG CARDINAL];` a record that contains information about the start and end character positions of the highlighted area of a source file
DBG Cmnds: procedure calls to the MDM interface for the execution of commands along with the parametes required for the commands. An example, **SetBreak**, is given in Chapter 5 while describing the MDM interface.
D&P Log: display and patch messages. Each message will contain the target name, the data displayed and its current value.
Display: command to the fetch the current value of memory, registers or IO ports.
DisplayInfo: a record with the return values for a **Display** command.
DisplayBuffer: command requesting the contents of the trace buffer for a single target.
DOData: a response from the loader to a **DOreq**.
DOReq: a request to convert the relative memory address of a variable to an absolute memory address.

endAddress: `LONG CARDINAL;` offset into target memory
endPos: `LONG CARDINAL;` an offset into a source file
Fill: command to initialize a range of memory with a byte value.
GetConfig: command requesting ConfigInfo.
Hlght Reqs: `RECORD[sourceName: STRING, startPos, endPos: LONG CARDINAL, turnOn: BOOLEAN];` a request to the file editor to load the source file in a window and highlight the statement given by startPos and endPos
InstructionStep: a single target command that causes the target to execute a single instruction and go into a halted state.
Jump: a single target command that transfers program control to a specified address.
ListBreaks: command requesting data about breakpoints set on one or more targets.
ListState: command requesting status information from one or more targets. **StateInfo** is returned by this command.
length: `CARDINAL;` a field of the `CorrRec`. This specifies the length, in number of bytes, of code that corresponds to a source statement.
ListTrace: command requesting data about traces set on one or more targets.
Load Memory: `RECORD[procNo: CARDINAL, startAddress: LONG CARDINAL, numBytes: CARDINAL, byteStream: ByteStream];` this includes a command to write to memory and a record defining the area of memory to modify and the new values.
Load Status: `{success, abortedByUser, failure};` the outcome of a `LoadMemory` call
MDM commands: these are commands sent over the parallel port to the MDM hardware. They are listed in the next section.
MDM Info: messages sent by the MDM hardware to the workstation about the execution states of the targets or the results of the MDM commands.
numBytes: `CARDINAL;` a field of `LoadMemory`. This specifies the length of the byteStream.
numChars: `CARDINAL;` a field of the `CorrRec`. This specifies the number of characters in the source statement.
Patch: command for patching memory, registers or IO ports.
position: `LONG CARDINAL;` an offset into a source file
ProcData: `ARRAY OF Procs;` a data structure that contains the names and locations of the targets.

procNo: index into the ProcData array for a given target.
Procs: `RECORD [debugModule: CARDINAL, name: STRING];` for each named target processor in the configuration this record contains the name, **name**, and the number of the debug module, **debugModule**, connected to it. The number of the debug module is needed to route debug commands.
relAddressData: a response from the loader to a **relAddressReq**.
relAddressReq: a request to convert the absolute memory address to an offset within a code or object file or vice versa.
RunStop: depending on the parameters, this command gets a target out of a halted state or places it in a halted state.
S-A Data: the source position or target address data that is returned in response to a S-A Reqs.
S-A Reqs: a request for the corresponding source positions or target address data for given target address data or source positions.
Scp Data: the **SymbolTableOffset** needed to identify the current scope of a target.
Scp Reqs: a request for the current scope of a target using **addressToDefineScope**
scopeNumber: `CARDINAL;` a number used to identify symbol scopes
SetBreak: command to set a break point on a given target.
SetTrace: command to set a trace on a given target.
SI Data: the SymbolInfo for a given variable.
SI Req: a request for SymbolInfo for a given variable
sourceName: `STRING;` source file name
sourcePos: `LONG CARDINAL;` an offset into a source file
startAddress: `LONG CARDINAL;` offset into target memory
startPos: `LONG CARDINAL;` an offset into a source file
StateInfo: a record giving the current state of the targets. It contains their current run/stop state and data about any breaks or traces set on them.
stopAddress: `LONG CARDINAL;` offset into target memory
SymbolInfo: `POINTER TO SymbolInfoObject;`
SymbolInfoObject: `RECORD[next: SymbolInfo, symbolName: STRING, symbolType: SymbolType, typeInfo: TypeInfo, startAddress, stopAddress: LONG CARDINAL];` a record used to define the properties of a symbol declared in a source module. It contains the name, **symbolName**, the type, **symbolType**, and the start and end relative offsets, **startAddress** and **stopAddress**, for the symbol. **TypeInfo** contains type information and is valid only for user defined types. The **next** field is used to create a list using `SymbolInfoObjects`.

SymbolType: `{integer, longInteger, real, boolean, character, pointer, longPointer, userDefined};`
symbolTableOffset: an offset into the symbol table used to identify the current scope. The data is stored in the correspondence table
TraceBuffer: `ARRAY OF TraceSnapShot;`
TraceEvent: a message sent by a debug module when the connected target satisfies a trace condition.
TraceInfo: a record listing the number, type and trace parameters of currently active traces set on a target.
TraceLog: messages about the setting and clearing of traces, posted in the Control Tool log window.
TraceSnapShot: `RECORD[address: LONG CARDINAL, data: CARDINAL, control: CARDINAL];` a snapshot of the address, data and control buses
turnOn: `BOOLEAN;` a field in the Hlght Reqs that is used to switch highlighting on and off.
TypeInfo: a record specifying the properties of a user defined data type
W & T Cmnds: commands to manage the windows and tools controlled by DEBUG. Refer to Section B.4 for more details.
W & T Status: the status of the windows and tools controlled by DEBUG, active, tiny or inactive.

B.2 MDM Commands

The following commands are supported by the MDM. They are sent to the MDM from the DEBUG workstation.

Command	Parameter	Description
01	num	select the debug module specified by **num**.
02	num1, num2...	select multiple debug modules specified by **num1, num2** etc.
03	num	send the following **num** number of bytes to the previously selected debug module. This command is used to send commands to the debug modules.
04	num	same as the previous command, except this command is issued when multiple debug

	modules are selected and the messages are sent to all of them.
05	return the number of targets connected to debug modules.
07	select all the debug modules.
08	put all the targets in run mode.
09	halt all the targets.

The MDM responds to all commands with an acknowledge byte to the workstation. In response to Command 05, it also returns a stream of addresses of the debug modules connected to targets. The message is terminated by sending an acknowledge byte.

B.3 Debug Module Commands

The following are a few of the commands supported by the debug modules. They are sent to one or more debug modules from the workstation, through the MDM, by calling the MDM Commands 03 and 04.

Command	Parameter	Description
02		stop target
03		reset target
04		display the current values of all the registers.
05	num1, num2...	patch the current values of all the registers. An array of patch values, **num1**, **num2**, follows the commands. The length of the array is equal to the number of registers on the target connected to the debug module.
06	adr, num	display a block of memory. **Adr** and **num** specify the memory address and the number of bytes to be displayed.
07	adr, num, data	patch a block of memory. **Adr** and **num** specify the memory address and the number of bytes to be patched. **data** is a byte array containing the new values.

08	adr, adr, string	set a range breakpoint. The first two parameters specify the memory range. **string** is a list of the selected control bus values.

The debug module responds with an acknowledge byte to commands 02, 03, 05, 07 and 08. It sends an array of bytes, terminated by an acknowledge byte, as response to 04 and 06. The length of the array depends on the target type when responding to the display all registers command. It depends on the number of bytes that are requested by the user for the display memory command.

B.4 A Sample Tool

The features provided by the window and the tools interfaces are illustrated below with a sample program that implements a tool. Figure 1 contains the source listing for the MESA interface **SampleToolDefs**. The record, toolData, contains the data that is manipulated by a tool. **toolData** has a numeric item, a string item, a boolean item and an enumerated item. The tool is created by calling the procedure **MakeTool**. It creates a tool window, displays prompts and accepts inputs for each field of **toolData** and provides commands that operate on **toolData**.

```
DIRECTORY
  Window  USING [Handle];
SampleToolDefs: DEFINITIONS =
  BEGIN
  toolData: ToolData;
  Access TYPE = {read, writes, delete, changeAccess};
  ToolData: TYPE = RECORD [count: CARDINAL, name:
STRING,
    wizard: BOOLEAN, access: Access];
  MakeTool: PROCEDURE RETURNS[w: Window.Handle];
```

Figure 1. The interface SampleToolDefs

Figure 2 illustrates the screen layout for the tool. It contains a message subwindow, a form subwindow and a file subwindow. The message subwindow displays exceptions and warnings. The form subwindow contains the tool commands and parameters and the file subwindow is for logging tool activity.

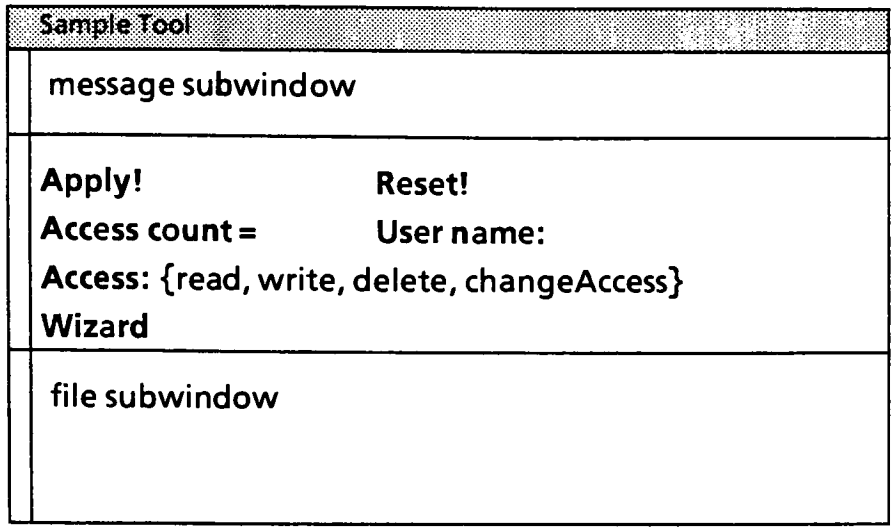

Figure 2. Sample Tool window

The program, **SampleTool**, implements **MakeTool** defined in **SampleToolDefs**. Figures 3 and 4 contain the source listing for the program. The DIRECTORY entry at the beginning of the program lists the public interfaces used in the program. The names of the procedures and data items used from each interface follow the USING keyword. The three subwindows shown in figure 2 are created by calling the procedures **MakeMsgSW**, **MakeFormSW** and **MakeFileSW** from the Tool interface. In MESA, it is possible to treat procedures as variables and pass them as parameters in procedure calls. This capability is used entensively by the tools interface. An example is given by the procedure **Tool.Create** when it accepts the procedure **MakeSWs** as an input parameter. The procedure **MakeForm** defines the items that appear in the form subwindow. The items include commands, strings, numbers, enumerated data and booleans. **ApplyProc** and **ResetProc** define the procedures that are called when the corresponding commands are clicked in the form subwindow.

```
DIRECTORY
  FormSW  USING  [AllocateItems, BooleanItem,
    CommandItem, Enumerated, EnumeratedItem,
    MakeFormItemsProc, NumberItem, ProcType,
    ReDisplay, StringItem],
  SampleToolDefs  USING  [Access, ToolData],
  Tool  USING  [Create, MakeFileSW, MakeFormSW.
MakeMsgSW,
    MakeSWsProc],
  Window  USING  [Handle];
SampleTool: PROGRAM EXPORTS SampleToolDefs =
  BEGIN
  toolData: PUBLIC SampleToolDefs.ToolData;
  tempToolData: SampleToolDefs.ToolData;
  msgSW, fileSW. formSW: Window.Handle;
  ApplyProc: FormSW.ProcType =
    BEGIN
    toolData ← tempToolData;
    END;
  ResetProc: FormSW.ProcType =
    BEGIN
    tempToolData ← toolData;
    FormSW.Redisplay[formSW];
    END;
  MakeTool: PUBLIC PROC RETURNS[w: Window.Handle] =
    BEGIN
    MakeSWs: Tool.MakeSWsProc =
      BEGIN
      msgSW ← Tool.MakeMsgSW[window: window];
      formSW ← Tool.MakeFormSW[window: window, formProc:
        MakeForm];
      fileSW ← Tool.MakeFileSW[window: window, fileName:
        "sample.log"];
      END;
    w ← Tool.Create[makeSWsProc: MakeSWs, toolName:
      "Sample Tool", initialState: active];
    RETURN[w];
    END;
  MakeForm: FormSW.MakeFormItemsProc =
    BEGIN
    enumerated: ARRAY SampleToolDefs.Access OF
      FormSW.Enumerated ← [[
      "read", read], ["write", write], [
      "delete", delete], [
      "change access", changeAccess]];
    items ← FormSW.AllocateItems[6];
    items[0] ← FormSW.CommandItem[name: "Apply", place:
      [0, line0], commandProc: ApplyProc];
```

Figure 3. SampleTool

```
--contd from figure 3
    items[1] ← FormSW.CommandItem[name: "Reset", place:
      [50, line0], commandProc: ResetProc];
    items[2] ← FormSW.NumberItem[name: "Access count",
      place:[0, line1], value: tempToolData.count];
    items[3] ← FormSW.StringItem[name: "User name",
place:
      [50, line1], value: tempToolData.name];
    items[4] ← FormSW.EnumeratedItem[name: "Access",
place:
      [0, line2], value: tempToolData.access, items:
      enumerated];
    items[5] ← FormSW.BooleanItem[name: "Wizard", place:
      [0, line3], value: tempToolData.wizard];
```

Figure 4. SampleTool