# ACTION-REPLAY : A REAL-TIME DEBUGGING TECHNIQUE

**Nikos G.K. Kanellopoulos**



A thesis submitted for the degree of
Doctor of Philosophy in Computing Science
at the University of Newcastle-upon-Tyne.
December, 1981.

**To my family**
**and friends.**

**ACKNOWLEDGEMENTS**

The author acknowledges the advice and encouragement of Mr. N. Ghani, Mr. J.G. Givens, Mr. K. Heron and Dr. C.R. Snow.

Thanks are also given to Ageliki, Athena, Gregory, Helen, Marcia, Maria, Toula and Vasiliki for their continued support expressed materially and spiritually throughout this Ph.D. study.

## A B S T R A C T

Currently available microcomputer development systems/tools become rather inefficient when employed to debug real-time malfunctions; that is, intermittent or even unrepeatable hardware/software malfunctions encountered in time-critical applications. A new debugging technique, namely the **Action-replay Debugging Technique,** is proposed which can efficiently deal with a large class of these malfunctions.

The aim of the Action-replay Debugging Technique is to provide an environment which is suitable for real-time debugging. In particular, an identical processor to the target, or a simulator of it, is forced to re-execute, or **Action-replay,** repeatedly and at any desirable speed the exact program path which the target processor traversed during the original interaction with its real-time environment. During successive "Action-replays" the user can investigate the system´s behaviour (including timing characteristics) without real-time constraints which normally exist in time-critical applications.

## ABBREVIATIONS

CPU : central processor unit.
DMA : direct memory access.
DSL : digital systems laboratory.
ECL : emitter coupled logic.
LED : light emitting diode.
LSI : large scale integration.
MMU : memory management unit.
MPU : micro-processor unit.
MSB : most significant bit.
MSI : medium scale integration.
MTS : Michigan terminal system.
PTR : paper tape reader.
RAM : random access memory.
ROM : read only memory.
SSI : small scale integration.
TTL : transistor-transistor logic.
VDU : video display unit.
[n] : reference number.
(# n): section number.
(A n): appendix number.

# S U M M A R Y

The development of microcomputer based real-time systems, being a complex process involving simultaneous design of both hardware and software, demands sophisticated debugging tools. However, a study undertaken at the start of this research work has revealed that currently available microcomputer development systems/tools cannot efficiently debug real-time malfunctions.

An analysis of the debugging process follows in order to reveal the differences between conventional and real-time debugging and with emphasis on the debugging tool capabilities, the availability of which should make efficient real-time debugging possible. In particular, real-time debugging presents two basic problems, namely **Execution Unrepeatability** and **Transparent Accessing of Program Status Information.** If efficient real-time debugging is to be achieved, the debugging system must be able to deal with these two problems.

Thereafter, a debugging system is proposed based on a new debugging technique, namely the **Action-replay Debugging Technique.** This technique allows real-time debugging to take place within a non-real-time environment, thus providing a "transparent" solution to the above mentioned problems. In particular, a snapshot of the target system state is saved together with a complete record of all external stimuli as seen by the target CPU during the real-time execution which follows. At the end of the program execution "Action-replay" may commence. That is, having reset the target system state to that given in the above mentioned snapshot, an exact reconstruction of the captured external stimuli is initiated which forces the target processor, or a simulator of it, to re-execute an identical program path with that which the target processor traversed during the original interaction with its real-time environment. During successive "Action-

replays" of the suspect program path the user can investigate the program's behaviour without real-time constraints which normally exist in time-critical applications.

Having defined the most important aspects of the Action-replay Debugging Technique, an Action-replay Prototype System is then developed, based on the "semi-resident approach"; that is, a host computer is connected to the target computer via an intelligent interface, namely the Host-MPU Interface. The target and host computers are the M6800 microcomputer and the LSI11/23 minicomputer respectively. The Host-MPU Interface complexity is kept at minimum levels, but even so it is necessary to employ more than 150 SSI/MSI TTL integrated circuits in the design.

Thereafter, three case studies are undertaken in an attempt to evaluate the Action-replay Debugging Technique. These studies show that the Action-replay Debugging Technique can indeed aid real-time debugging of software (as well as hardware) malfunctions by providing a debugging environment which encourages the user to embark into a systematic and efficient debugging process; debugging the same malfunctions via conventional debugging methods would require a vast amount of trace memory for providing less efficient diagnostic facilities. In particular, the erroneous program behaviour can be kept in a stable condition so that particular symptoms relating to particular faults can be focussed upon and analysed; that is, the user is not confused by rapidly changing symptoms.

Finally, the Action-replay Debugging Technique limitations are discussed and recommendations are given for overcoming these limitations and for upgrading the Action-replay Debugging Process in future implementations.

TABLE OF CONTENTS

# CHAPTER 1

## 1 INTRODUCTION

As we move into the LSI age of computer technology, hardware cost is decreasing rapidly as a result of employing mass-production processes and automating techniques within these processes. The selling cost of a **microprocessor unit** (MPU), which is the major product of the LSI technology, becomes almost negligible when compared to the development cost of the user system (hardware interfaces and driving software). If some kind of cost balance is to be achieved user system development time, being proportional to development cost, should be kept to a minimum level.

According to past statistical studies on hardware/software development [19], debugging represents more than 1/3 of the overall system development effort; the other 2/3 being the design and construction of the overall system. One way of reducing development cost, therefore, is to improve debugging efficiency; that is, to detect and correct in a short time the maximum number of design and construction errors.

It goes without saying that it is desirable to produce programs containing minimum errors in the first place. A lot of emphasis has been given in the past into the development of design methodologies and tools aiming towards the realisation of the above notion (section #1.1). In practice, however, not only the design process is extended further, due to the extra effort required for the necessary formalisation, but unfortunately not all errors are eliminated. Hence, the need for efficient debugging tools (# 1.2), the absence of which has

forced debugging to become a rather specialised and, therefore, an expensive process requiring experienced system designers.

As will be shown in section #2.1 of this thesis, the need for efficient debugging tools becomes even more apparent within **Real-time Systems;** that is, either feedback or open-loop control systems of asynchronous characteristics where the controlling system component (the computer) is directly influenced by the response of the system component which is either under control (e.g., an autopilot computer and the airplane), or simply monitored (e.g., an alarm system).

Currently, real-time debugging is increasingly required in microcomputers which, due to their low cost, are heavily involved in real-time applications. Hence, microcomputer real-time debugging is focussed upon in this research work although a major part of the ideas presented apply to any computer system which is employed in a real-time application (*).

Before presenting the basic organisation of a selection of currently available microcomputer debugging systems (# 1.2) a number of definitions will be stated concerning the program development process in general.

## 1.1 Program Development

Program Development is the process of "Designing" a program (# 1.1.1), "Validating" the logical correctness of its implementation (# 1.1.2), "Debugging" it (# 1.1.3) and evaluating its "Performance" (# 1.1.4).

Either a **Top-down** or a **Bottom-up** strategy may govern the overall program development process [47,51]. During the former strategy the program main procedure (i.e., the top-most

* Throughout this thesis a computer controlled real-time application is also referred to as either "user application" or "target application".

level) is developed first in the chosen programming language. Then, the program sub-procedures are developed which are also split up, and so on, until the entire program is developed. At each development level, "dummy" procedures substitute those program procedures which have not yet been developed. During the latter strategy the exact opposite happens.

The **Stepwise Refinement** program development strategy [49] is similar to the Top-down strategy; the basic difference being not having to work in the programming language. For example, one might use plain English statements describing the function to be performed at each level. When bottom level is reached, the algorithm is translated into programming language code.

If the program is deliberately split into well defined sections **(modules),** which are developed quite separately, only being brought together when they have all been found to work individually, then that is **Modular Programming.**

Finally, **Structured Programming** [8,22,32,48] is another method of program development; at each step of the development the program can be logically decomposed into distinct sub-structures whose correctness is manifest by the structure of the program itself. The **Principle of Data Abstraction** is often used in conjunction with structured programming, thus forming a powerful tool for designing programs (Explicit Data Abstraction). That is, data is defined by specifying the way it is represented on store and stating all possible operations on it. Then, implementation details are ignored by thinking solely in terms of meaningful operations performed upon the data than in terms of actual data. Data Abstraction may also be implemented within programming languages in the form of arrays, records, etc (Implicit Data Abstraction).

### 1.1.1 Program Design

Program Design is the process of constructing a program according to application requirements.

The program design process is divided into two stages, namely

the **Analysis** and **Synthesis** stages. During the former design stage a study of the available program requirements is undertaken yielding in the generation of **static specifications** (implementation independe specifications) for all program modules/sub-modules. During the latter design stage conversion of program static specifications into **procedural specifications** (implementation dependent specifications) and thereon into programming language code takes place.

### 1.1.2 Program Validation

Program Validation is the process of ensuring that the program performs the intended logical functions. The program validation process is divided into two stages, namely the **Verification** (or Proving) and **Testing** stages.

The former stage is a theoretical approach to program validation usually employed when the program domain is rather large and therefore testing cannot cover all possible cases. Hence, the program correctness is verified either by developing mathematical proofs (static approach) or by giving emphasis to the correct design of the program (constructive approach).

The latter stage is an empirical approach to program validation useful for checking programs with small finite number of program paths (in practice it is used to show the presence of errors rather than their absence). Hence, the program is tested for logical correctness by evaluating its response to a selected set of input data. According to whether a Top-down or a Bottom-up development strategy is employed, this input data stream consists of "real" and "symbolic" data correspondingly (except during the "system testing" stage of the Bottom-up strategy when the data has the form of "real" input).

A variety of program testing techniques are available [26] which are divided into two major categories, namely **Internal Testing,** during which the internal program structure is examined (e.g., path, branch and module testing), and **External Testing,** during which the set of input data is formed by

extreme values (acceptance and confidence testing). Information of the following nature should be generated during testing [38] :

* scope, purpose and objectives of testing.
* hardware/software resources required during testing.
* execution order of tests.
* set of input data forming each test.
* data to be collected during each test.
* success criteria of a test.
* procedure followed if a test fails.
* test verification method.

**Programming Errors** may be divided into four main categories according to where along the development process they are introduced.

1) **Requirement Errors** : failure to satisfy the application requirements when generating the program requirements (including performance requirements).
2) **Specification Errors** : failure to satisfy the program requirements when generating the static specifications.
3) **Design Errors** : failure to satisfy the static specifications when generating the procedural specifications.
4) **Implementation Errors** : failure to satisfy the procedural specifications in the implementation of the program algorithm resulting in incorrect data/control flow. Implementation errors are subdivided further into categories such as missing/extra/wrong program paths and actions within a path, which may be caused by various reasons (e.g., flowchart misinterpretation, undefined/multi-defined/misspelled identifiers, improper/omitted variable initialisation, incorrect branch-tests, etc.).

### 1.1.3 Program Debugging

Program Debugging [15,16] is the process of locating and correcting programming errors sensed either during the initial program execution, or during program testing, or later on during program maintenance.

In **Top-down Debugging** the main routine is debugged first and, as soon as a subroutine is coded, is debugged as part of the existing "debugged" program. Any data used has the same form as input for the final program. However, when the program is developed by a team of programmers, debugging of a particular program module may be delayed because subroutines, which communicate with this module, must be developed first.

In **Bottom-up Debugging** each program module/subroutine is debugged separately **(Module Debugging)** in artificial environment which generates input for the module in question. Only later on when all the modules are put together and debugged as a whole **(System Debugging)** data has the form of "real" input.

Debugging via **Inspections** [11,40] is a well organised, efficient and economical method based on team discussions taking place throughout program development. The team is formed by the Moderator (manages the team), Designer (produces the program design), Implementor (translates design into code) and Tester (tests the product). The process is described in terms of operations, namely the Overview (general discussion about program), Preparation (individual education), Inspection (find errors), Rework (fix errors), Follow-up (ensure all fixes are applied correctly).

**Code Walk-throughs** [40] are similar to Inspections but less formal (the structure of the team and the objectives for each operation differ in different places).

In debugging via **Desk Executions** [31] the program statements are simulated manually one by one while the current value of each variable is recorded on paper (one column for each variable).

When **On-line/Interactive Debugging** is employed program status information is monitored either in between, or during program execution sessions.

**Breakpoints** stop program execution for subsequent **Program Status Information** monitoring. In there simplest form are that of **Address Breakpoints;** the address of the current instruction is compared either by software or hardware with a set of preselected addresses (located in the "breakpoint table") and in the event of a "match" program execution is stopped and control is transferred to the monitoring system.

**Stop-conditions** activate the Breakpoint facility at the occurrence of one/many pre-selected conditions (e.g. a number of clock counts after a particular event has occurred, during a particular memory cycle, etc.).

**Dumps** record information about a "failing" state of computation (all or a selected part of store). Some examples of Dumps are the **Post-mortem Dump,** which is initiated after the termination of the user program, the **Snapshot Dump,** which is initiated at an intermediate execution point, and the **Comparison Dump,** which provides only the difference between the current computational state and a preselected one captured by a previous dump.

**Traces** provide a sequential record of program activity obtained during execution. Program-variable Trace, Instruction Trace, Subroutine Trace, are examples of Traces. Hence, Dumps are used to find wrong quantities, while Traces provide checks on specific code modules involving those quantities.

**Intermediate output statements** record intermediate data values of selected program variables during execution by being inserted at carefully selected places amongst the code in question. These statements can be conditional and therefore left within the final version of the program for maintenance purposes (e.g., to collect output used in confidence checks). The main advantage of this technique is that it is independent of program language and produces meaningful output which is in

a format easily readable by the programmer (unlike traces).

**Debug-messages** provide useful information about the program under suspicion in the form of State-diagrams, Tree-diagrams, functional descriptions, and records of relevant reference information (decisions taken, results obtained during the debugging of a particular program module, etc.).

### 1.1.4 Program Performance Evaluation

Program Performance Evaluation is the process of checking non-logical correctness of programs.

Expected utilisation of resources, such as overall execution time response to external stimuli, data throughput rates, required memory size, etc., is checked via a number of different software techniques. For example, execution statistics are collected by inserting software counters at key locations within the user code [21,43].

## 1.2 Microcomputer Development Tools

Microcomputer systems are increasingly more involved in the control of real-time applications and are interfaced to unaccountable real-time environments; hence, the appearance of time-dependent bugs which are usually accompanied with the problem of repeatability. Unlike conventional digital systems, microprocessor systems often involve simultaneous hardware and software implementation (and therefore debugging); software bugs may be taken as hardware malfunctions or vice-versa. In addition, problems such as accessing Program Status Information, controlling program execution and user interfacing (command language, Program Status Information display format, etc.), which were tackled in conventional systems via sophisticated operating systems, now emerge in the microcomputer world.

If only for the above reasons debugging such systems require a

high degree of hardware and software expertise. Therefore, it was realised early in the evolution of the microcomputer that there is a need for different debugging tools than those used in conventional systems.

Initially, application software was written in a **cross development mode;** that is, a host computer (maxi/mini) was used to assemble/compile the source program and even simulate its execution for debugging purposes. The obvious advantage of this method was that some of the software needed was already available within the chosen host system (e.g., file management and editing facilities). Any problems arising later on while the target MPU was executing the object code were tackled using either **storage oscilloscopes** or **logic analysers** [42]. These devices, because they operate down at the binary system level, some-times octal/hexadecimal representation is provided (e.g., the HP 1610B logic analyser), and can only capture a very small "trace" of the original execution (usually 64 execution steps), are very inefficient as far as software debugging is concerned. Even during hardware debugging, and especially when a large number of control signals is to be monitored, the interpretation of the captured data is very difficult because the information shown on the screen is limited to a large number of binary/hexadecimal digits with no reference whatsoever to corresponding signal-names or other target system information.

Therefore, it was realised that some means of controlling and monitoring the target MPU was required for debugging purposes. Initially, **microcomputer analysers** , which are logic analysers specially tuned to microcomputer needs, were developed. These devices, apart from monitoring facilities can also provide minimal control over the program execution. Later on a more substantial breakthrough was the development of **Microcomputer Debugging Consoles.** These consoles, according to their organisation, are divided into two main categories, namely the **Hardware Consoles** and the **Software Consoles.** Some examples of both kinds of consoles are given below in an attempt to show the variety of debugging techniques used and discuss their organisation.

### 1.2.1 Hardware Consoles

A hardware console [16] is based on a front-end panel (LEDs+switches) and attached to the target MPU buses via a dedicated hardware interface which not only takes care of the data traffic across itself but also controls the MPU in question. Both actions are initiated via certain signals generated by the panel control keys.

A hardware console, being part of the target microcomputer, has the advantage of debugging the suspect software within the real environment **(Resident Approach).** Its simplicity allows a cheap construction and, therefore, is quite popular especially amongst amateurs. Furthermore, its hardware interface is usually fast enough to be employed in a primitive type of critical-time debugging. On the other hand its functionality is limited due to the inflexible input/output device used and the lack of computing power; only basic debugging functions are available, such as MPU initialisation, single-step/continuous user program execution and direct memory access (useful for both hardware and software debugging since it is hardware driven). Any further debugging capability, such as accessing of MPU registers, would involve either additional hardware or a specially written program executed by the target MPU itself (e.g., NIKBUG [28]). In addition, the "hardness" of the console-MPU interface makes it impossible to attach a hardware console to different kinds of MPUs without major re-design of the interface. This inflexibility may be tackled successfully by employing a microprogrammable version of the console-MPU interface **(Firmware Console [28])** whose properties could be changed to suit different MPU-types.

### 1.2.2 Software Consoles

A software console is based on a flexible input/output device, such as a TTY, utilising the computing power of a processor to perform various tasks, including debugging. In particular, either a host computer is employed **(Cross Approach),** or the target MPU itself **(Resident Approach),** or both **(Semi-resident Approach).**

**Cross Approach** : In the cross approach the debugging process is controlled by the "debug" section of the host software. The host development system itself is a microcomputer based on an MPU functionally identical to the target one. This implies that, in addition to executing its operating system, the host MPU directly executes the target application software.

For example, Intel's **INTELLEC MDS** [27] and Motorola's **EXORciser** [37] microcomputer development systems employ the computing power of the 8080/8085 and M6800 MPUs respectively. These MPUs, not only provide the required software development environment (file handling, cross-assembly, etc.) but also execute the produced 8080/6800 code under the control of the "system monitor" and "EXbug" monitors respectively.

**Resident Approach** : In the resident approach the debugging process is controlled by a **Monitor** program which is loaded onto the target system together with the application software. Following the availability of increasingly more microcomputer support software (even high level language compilers are provided within microcomputer systems) the development of resident debugging consoles is not surprising.

Motorola's monitor, called **MIKBUG** [37], takes control either on "reset" or by executing a Software Interrupt Instruction (SWI) placed by the user at a key point within the program (primitive "breakpoint" facility). The software interrupt forces the MPU status into the system stack before passing control to MIKBUG which, responding to user commands entered via a serial asynchronous line, supplies some basic debugging functions such as accessing both memory and MPU-register contents (found in the stack). Control is passed back to user program by executing an "Return from Interrupt" instruction (RTI) which restores the MPU status.

The **BEDBUG** monitor [13] (implemented on the M6800 microcomputer) controls user-program execution by inserting automatically dummy SWI instructions in between user instructions.

Control is passed to BEDBUG, as explained in the previous paragraph, which updates the display memory and then receives user debug commands. The output device used is not a TTY but a memory-mapped character display. At the end of a debugging cycle, an RTI instruction is executed passing control to the next user instruction. Apart from the usual capabilities found amongst monitor programs, the Bedbug monitor provides continuous display of user-program status information (snapshots), object code disassembly and software breakpoint table using user-defined symbols for address identification.

The **ALADDIN** debugging system [12] is also based on a monitor program whose main difference from other monitors is that a location independent breakpoint facility is employed. In particular, debugging assertions, which describe logical relations among various components of the program state, are inserted in between assembly language statements. The user program is executed via interpretation because the ALADDIN execution handler must have control of the CPU between execution of successive object code instructions in order to evaluate the assertions and suspend execution of the target program if the outcome is false.

The **SOLDA** monitor [7] employs the source language debugging technique [41]. That is, all debug information is referred to the source listing and not to the machine language version of the program. In particular, the program is written in high-level-language (ESPRINT in this case) and during its compilation debugging code is inserted in between statements. The generated object code is linked with parts of the SOLDA system before being loaded into the memory. During debugging the user enters commands which, after being interpreted by SOLDA, are executed under its close supervision.

**Semi-resident Approach** : In the semi-resident approach the debugging process is controlled by the "debug" section of the host operating system as with the cross approach. However, the application software is executed by the target computer, which is connected to the host system, and not by the

host processor. Various ways of connecting the two systems exist some of which are given below.

Many microcomputer development systems are attached to the target system via either a ROM socket **(ROM Emulator)**, or an MPU socket **(In-circuit Emulator)**. The advantage of the ROM emulator is that it replaces the target system ROM with RAM which can be separately loaded and accessed. An in-circuit emulator [30] emulates under real-time conditions the functions of the missing MPU chip by employing either the same MPU model as the one used in the target system, or bit-slice architectures which mimic the target MPU functions. The software debug functions of the host development system may then be extended into the target system. However, debugging of real-time applications remained difficult. Hence, a real-time trace interface is usually employed which, in conjunction with the in-circuit emulator, captures the state of the MPU buses for a fixed number of clock cycles prior to an execution break.

For example, Intel´s INTELLEC MDS [27] microprocessor development system is equipped with a dedicated in-circuit emulator, namely the **ICE-80** which amongst other capabilities can capture 44 machine cycles in real time thus permitting limited real-time debugging.

The **HP-6400** [25] and **TEKTRONIX-8002/8001** [45] development systems use a generalised version of the in-circuit emulator technique. In particular, they can cope with different MPU types by employing the corresponding in-circuit emulator (based on an identical MPU chip) and changing some of the system software (e.g., cross-assembler).

A similar debugging approach is employed within the **AMPL TI-9900 Microprocessor Prototyping Laboratory** [46]; the difference being that interactive emulation and trace control is achieved via an in-circuit emulator which is supported by a high level interpretative debug language, the AMPL. The AMPL is an expression-oriented structured language and not a

command oriented monitor as in most microcomputer development systems. The user may enter an AMPL debug statement (or a block of debug statements) which is interpreted by the system software. A debugging activity may then be performed without close user supervision.

An interesting version of the semi-resident approach is that employed by **MicroAde** [6]; the host system (DEC PDP-11 controlled by RT-11 operating system) is connected to the target microcomputer via a standard serial asynchronous line. Although this type of interface ensures microcomputer independence at hardware level (a serial link is available within most MPU systems) both the host software and the target-MPU software must be adjusted appropriately. The former, apart from its general purpose run time system, interactive debugging system, user communication system, and other utility programs such as loader, editor, PROM programmer and floppy disk driver, requires a cross-assembler for the generation of the target MPU object code (program modules are written in target MPU assembly language). The latter, apart from its native monitor, requires a "debug interface" program which controls any interaction between the host debugging system and, via the native monitor, the user application programs. MicroAde also provides memory reference via user-set symbols, automatic execution of test sequence and result comparison against a pre-established record, test log, input/output simulator and finally MicroCOBOL, which enables the user to write microcomputer independent programs in COBOL syntax (these programs are compiled into MPU independent intermediate code, which is loaded into target memory together with the MPU dependent MicroCOBOL interpreter).

The **Millennium Micro System Emulator** [34] is also a generalised version of the in-circuit emulation technique but with the flexibility built-in in the in-circuit emulator end and not in the host development system end; that is, it is a stand-alone in-circuit emulator which is connected to any dedicated host computer via an asynchronous serial line (RS-232), thus transforming it into a universal development system. The above programmability is achieved via bit-slice

processor techniques. Finally, as with other in-circuit emulators, the Millennium Emulator has an optional real-time trace facility, which is one of the most advanced that currently exist. In particular, this trace facility employs a 128-location trace memory, event detection consisting of two comparators capable of performing either (=) or (</=) or (>/=) operations between address/data/control lines and user preset values, detection of fetch/memory/IO cycles, and a further detection of a pass count of n events and/or a delay count of n clocks. A combination of the above events may be chosen for trace qualification while a count of either real-time, or one of the above events may be made between two points in a program or between events.

A serial asynchronous line is also used by the **CONTEXT** microcomputer development system [44] for connecting the host computer (DEC PDP-11) to any target MPU. Program modules are written in CORAL-66, compiled into PDP-11 code, and run under the RSX 11-M operating system and a package of software aids (MASCOT). When all modules are completed and satisfactorily tested within the host, the corresponding CORAL-66 compiler is employed to translate them into target MPU code, which is transferred via the link to the target MPU. At this point the source language debugging technique is employed; that is, as far as the user is concerned the CORAL-66 version of the program is executed and not the target MPU code. Test are then run under CONTEXT control and via the TTY link. A monitor program, previously loaded into the micro, collects and passes information back to the host via the link for displaying and monitoring purposes.

**Approaches' Comparison** : The main advantage of the **cross approach** is that the application software is designed, written and debugged into a host environment which, provides many development software packages and sophisticated input/output devices. Mainly for reasons of economics these tools could not have been made available in a user system. However, if the produced software does not work first-time in the application environment (and this is very often the case) debugging

becomes rather difficult especially for real-time applications.

With the **resident approach** on the other hand, the application software is executed within the real environment; a desirable property since simulation cannot cover all real-world situations. However, a lot of system processing time is allocated for the upkeep of the resident monitor which controls the debugging process and, therefore, this approach is not appropriate for time-critical applications.

The **semi-resident debugging approach** combines the "best of both worlds"; that is, the powerful software development tools of a host computer and the capability of debugging the latest version of the program within the real environment (host and target computer are coupled together). However, as can be seen from the examples given above, a small monitor program is usually required within the target system for collecting program-status information and driving the Host-MPU link (MicroAde and CONTEXT systems). Therefore, debugging a program without slowing down its execution **(Real-time Debugging)** is not possible; only in the case of the in-circuit emulation technique, where a trace memory is incorporated into the system and assuming that it is possible to employ sophisticated monitoring conditions in order to counterbalance the limiting amount of this trace memory, real-time debugging is possible. However, even systems that satisfy the above assumption (e.g., the Millennium Emulator) cannot cope with certain aspects of real-time debugging such as MPU-registers monitoring (see Chapter 2).

## 1.3 Research Direction

An analysis of the debugging process follows in Chapter 2 of this thesis with emphasis on the debugging tool capabilities, the availability of which should make efficient real-time debugging possible. In Chapter 4, a debugging system is proposed based on a new debugging technique (Chapter 3) which is suitable for interactive debugging of real-time microcomputer

systems at the source language level. Finally, in Chapter 5 a prototype real-time debugging system is developed, which is then evaluated in Chapter 6.

# C H A P T E R 2

## 2 REAL-TIME DEBUGGING PROCESS

The degree of efficiency of the real-time debugging process depends not only on factors such as the quality of the program, the programmer´s open-mindness and knowledge in relation to both the application problem and the operational environment, the kind of the real-time application, the type of the error in question, etc., but also on the capabilities of the debugging tools.

### 2.1 Debugging Process Analysis

An analysis of the debugging process follows in order to reveal the differences between conventional and real-time debugging and, consequently, decide on the basic capabilities which a real-time debugging tool should have.

#### 2.1.1 Conventional Computer System Debugging

Figure 1 shows a computer system suitable for running programs of non-real-time nature and interfaced to some external hardware, partitioned into well defined independent modules (e.g., card reader, printer), which is controlled by the system´s utility software. The target program is defined by what the user wants to do, the processor employs known semantics for program interpretation, the initial state of the system is known, the input to the program is well defined and, therefore, this is a **Deterministic System.** Hence, a complete

description of what the system should do can be derived at any stage of the software development process.

Abnormal termination of program execution, generation of "strange" output, wrong sequencing of operations, etc., indicate the existence of at least one program error. Past psychological studies [19] have shown that programmers debug their programs by initially trying to understand the actual behaviour of the "wrong" program version. During this program behaviour analysis one subconsciously attaches a value to each program module indicating the probability of having (or not having) an error. After comparing all available Program Status Information with the expected program state, derived from the description of what the program is supposed to do, these values are either increased or decreased; an action which influences the progress of the debugging process. Then, after deciding on the type of Program Status Information to be monitored and on the execution interval during which this monitoring is to take place, a program re-execution is initiated which, because the system is deterministic, follows the original program path; that is, a malfunction occurring within a deterministic system is repeatable.

Therefore, a number of re-executions may be initiated, during each of which different Program Status Information in monitored, until a "mismatch" between the current program state and the expected program state is detected, which hopefully will lead to the error in question.

A variety of systems are available allowing Program Status Information monitoring of programs written in low level languages (# 1.2.2), or modern high level languages for that matter [15,43], running in conventional processor systems under the control of either a Monitor program or an operating system.

Microcomputer engineers usually employ the **Single-stepping** execution technique for Program Status Information monitoring. That is, the user decides that the probability of an error occurring at a particular point within a program module is

rather high, allows execution to continue until just before this point (up to which program behaviour is understood) planning to execute one instruction at the time thereon and collect Program Status Information for debugging purposes. **Address Breakpoints** control this activity.

Single-stepping, being a rather mechanical and slow process, is an inefficient technique as far as Program Status Information monitoring is concerned and in the world of large conventional systems has been replaced long ago by other more efficient techniques (# 1.1.3) such as tracing, dumping, monitoring of inter-module message flow, etc; although some times single-stepping is currently used when debugging CPU hardware. However, as will be shown in the next section, these techniques are not suitable for real-time debugging.

### 2.1.2 Real-time Computer System Debugging

Sophisticated operating systems have been traditionally provided, among other reasons, for handling real-time applications. However, real-time debugging remained a complex process.

Figure 2 shows a similar computer system to that of Figure 1 interfaced to a real-time application which normally consists of interacting hardware modules which the user program controls and which interact further with the real-time environment (it is assumed that the utility input/output devices are controlled by the operating system).

The input to the computer system is a function of the target hardware's timing/functional characteristics and also, if a feedback control is employed, a function of the control output of the computer system itself. Therefore, this input cannot be defined unless a complete description of the target hardware is available. In practice, however, the target hardware behaviour is unpredictable either because most of the real-time applications involve simultaneous software and hardware development (i.e., complete hardware description cannot yet be derived), or because hardware description

correctness cannot be guaranteed due to its high complexity and to uncertainty as to its response to a certain combination of real-time external events (highly asynchronous characteristics). Therefore, real-time systems are usually **non-deterministic;** a complete description of what the system should do at a specific instant in time cannot be guaranteed that it can be derived.

The non-deterministic nature of real-time computer systems implies that it is difficult to predict "expected" program states, against which any captured Program Status Information should be compared in order to decide that the program is (or is not) behaving properly. In other words, it is particularly difficult to decide on the type of the required Program Status Information; very often the wrong Program Status Information is monitored which does not help in the detection of the existing error. However, it is more than likely that the program path execution is unrepeatable and, therefore, additional program status information cannot be obtained.

**Execution unrepeatability** forces the programmer to increase the Program Status Information amount which is monitored and this presents various problems associated with trace memory overflowing (# 2.1.1); not to mention that most of the obtained information is bound to be irrelevant to the malfunction in question and, therefore, misleading (even in cases where **Trace Formaters** are available).

Even if the execution of the program path in question is repeatable, the currently available Program Status Information monitoring techniques introduce a considerable time-overhead to the target program execution time in cases where extensive monitoring is required (# 1.1.3). Consequently, the relationship between the program and its external environment is altered, making diagnosis of intermittent and other time-dependent faults under these conditions impossible. Hence, the unrepeatability problem discussed above persists.

The "stack" arrangement found in most modern MPU designs, forms a good environment for subroutine usage (including

interrupt routines). The result is that modular programming and, therefore, modular debugging is encouraged. However, microcomputer real-time debugging becomes critical only when program modules have been linked together **(System Debugging)** and the entier program is executed within the target environment. That is, only when both the target application timing and the program timing are coupled together forcing the occurrence of unprecedented timing errors. This implies that a cross-simulator can be used initially, for checking the logical correctness of each program module, until the complete program is available. Only then, it is necessary to transfer the program to the target environment for subsequent real-time debugging.

This thesis, is concerned with real-time debugging and, therefore, it is assumed that the program development has reached the System-Debugging level and that it has been transferred to the target environment for subsequent real-time debugging, which by its very nature is performed "on-line" **(Interactive Debugging).**

## 2.2 Real-time Debugging Tool Capabilities

Due to human memory and processing capacity limitations programmers are trying to detect only one error at a time and are focusing their attention on a local region of code. This also implies that specific information is usually requested from the debugging tool employed.

Obviously, selective monitoring is desirable during real-time debugging. This implies that monitoring must be controlled by evaluating logical conditions which relate program-status information elements to user-predetermined addresses/constants or other Program Status Information elements. The conditions involved should also specify the information type to be monitored (e.g., all the values which a particular variable acquires), the origin of the monitor activity (e.g., at the entrance of a subroutine, after an interrupt, etc.) and its duration in terms of either time or event occurrences (e.g.,

for a period of "n" clock-cycles, as soon as another condition is met, etc.).

Selective real-time monitoring of Program Status Information (excluding MPU registers) is currently implemented via hardware techniques (e.g., Millennium Emulator (# 1.2.2)). However, there is a limit to the number of operands which a logical condition can have, depending on the amount of hardware that is available for its evaluation. A software implementation would be more flexible but would slow down program execution which is not acceptable for real-time system debugging (# 2.1.2). Furthermore, because in practice real-time program behaviour is unpredictable, the majority of the real-time malfunctions, including intermittent faults, present serious problems; guessing what sort of information to monitor is difficult (# 2.1.2). Cases like these are very difficult to be debugged employing currently available debugging systems even though such systems may include selective monitoring of Program Status Information.

Monitoring in real-time Program Status Information (preferably selectively) is clearly desirable, but is it possible? Could either the MPU-register contents, or the program variables residing in memory, be monitored without halting or slowing down the target program execution? Could input/output device registers be monitored without affecting their contents (an inevitable fact in the case of "active" registers which are usually employed in input/output interface devices)? Could the vast amount of trace memory required for storing the captured Program Status Information be kept to a practical size? And, finally, could all these happen in a general kind of way in order to be applicable to a variety of real-time systems based on either identical or completely different microcomputers?

These are some of the "real" questions that emerge as soon as the design requirement of monitoring Program Status Information in real-time is considered.

## 2.3 Summary

Real-time debugging presents two basic problems, namely Execution Unrepeatability and Transparent Accessing of Program Status Information. If efficient real-time debugging is to be achieved, it is essential that the debugging system can deal with these two problems. Only then "Program Status Information Selective Monitoring" and other techniques may be employed to achieve proper user interfacing to the Debugging tool and, consequently, to increase the debugging efficiency even further.

# CHAPTER 3

## 3 ACTION-REPLAY DEBUGGING TECHNIQUE

Having presented the basic problems encountered in real-time debugging (Chapter 2), an investigation is now undertaken into the possibility of providing a "transparent" solution; that is, a solution that does not involve the user in extra, time consuming activities. In particular, the possibility is investigated of transforming an "unrepeatable" real-time execution into a "repeatable" one and at the same time of transforming the real-time Program Status Information monitoring into a non-real-time process and still being able to study all the real-time activities of the program.

Obviously, the nature of the system activities which influence a real-time program execution is a key factor in the above mentioned transformations and, therefore, needs some clarification. Starting at a particular instant in time, the execution flow is influenced by :

**1)** the internal MPU state at that instance (i.e., the current value of its registers),

**2)** any data entering the MPU during subsequent execution either via the system memory (including the DMA buffer), or via any of the input ports attached to the MPU buses (including any status information generated by these ports), or via input ports located within the MPU chip itself,

**3)** any hardware interrupts serviced by the MPU thereafter (e.g., the NMI and IRQ of the M6800).

Any other changes occurring at the user-application side of the input ports are redundant as far as software debugging is concerned since they do not influence the program execution flow.

As soon as program execution has been abnormally terminated and assuming that the internal state of a non-deterministic microcomputer system is known at an arbitrary moment in time, it is possible to reconstruct any subsequent system state if all the external stimuli that entered the microcomputer system from that moment onwards are known together with their correspondence to the program path in question.

Having kept both the initial microcomputer internal state and the external stimuli sequence, there is no reason what so ever for not being able to repeat the simulated re-execution several times and each time to traverse the same program path; that is, execution repeatability is achieved.

The above program path reconstruction need not take place in real time. Slowing-down the execution does not alter the program behaviour as long as all external stimuli, which have been recorded via some sort of real-time monitoring mechanism (# 3.1), are synchronised with this execution. Hence, execution-speed control is achieved.

Program status information, which is an exact image of the required real-time program status information, may be obtained selectively during these non-real-time re-executions; that is, the accessing of Program Status Information is no longer a critical design requirement.

Finally, the user himself does not need to know anything about the low-level information captured during the real-time execution of the target program, or indeed anything about the re-execution mechanism itself; instead, the user's attention can be focussed upon the type and amount of Program Status Information which is required in order to understand the behaviour of the program path in question. Hence, a "transparent" solution is achieved.

In general, therefore, the above technique allows real-time debugging to take place within a non-real-time environment. For obvious reasons it is named **Action-replay Debugging Technique** (*).

Problems associated with "external-stimuli monitoring", "program-path Action-replaying", "Program Status Information monitoring" and other processes relevant to the Action-replay Debugging Technique, are discussed in the following sections of this chapter.

## 3.1 External Stimuli Real-time Monitoring

The external stimuli, which are monitored during the real-time execution of the target program (# 3.0), consist of any data entering the target system via the input ports, any hardware interrupts triggered by an action initiated within the application hardware (software interrupts need not be monitored since their initiation is triggered from within the program itself) and any data entering the system memory via DMA techniques.

There are two basic problems associated with the real-time monitoring of the external stimuli. The first problem is their synchronisation with the Action-replay execution and the second problem is the risk of the trace memory overflowing. These problems are discussed in the following sections.

---

* A similar technique has been previously employed in the EXDAMS Debugging System [2] which, however, requires the insertion of a large amount of diagnostic/monitoring code into the target program and, therefore, is not suitable for real-time debugging.

### 3.1.1 Input Port Data Monitoring

**Input Port Data Synchronisation** : The required synchronisation is not difficult in the case of data entering the system via the input ports and under the control of the MPU itself. During the real-time execution, and in particular during input port accesses, the data bus contents are recorded and stored sequentially in the **Data Trace Memory.** During the Action-replay execution, this trace information is played back in a FIFO fashion; that is, the next data word to be provided during an input port cycle is the word which is located next in the trace memory.

The above procedure is independent of the number of input ports in the system.

**Data Trace Memory Overflow** : The basic problem when monitoring the input data sequence is that of data arriving at the input ports at a large rate and overflowing the trace memory in a very short period of time. This is especially true for the type of malfunctions for which the Action-replay debugging technique is most needed; that is, intermittent errors which by nature require long monitoring sessions. Obviously, if the trace memory overflows, an Action-replay of the program path in question can take place only until that point.

Assuming that the trace memory remains fixed to a practical size, the above problem can be dealt with in a variety of ways.

One way is to omit redundant data. For example, because only the data which actually enters the system influences program execution (# 3.0), monitoring at the MPU side of the input ports and not at the target application side ensures that only essential information is recorded. A similar technique may be applied when monitoring system interrupts (# 3.1.2), or data entering the system memory via DMA techniques (# 3.1.3).

Another way of delaying the overflowing of the trace memory is to compress repetitive data. For example, a repetitive data

word (say, n-bit wide) may be stored in a (n+1)-bit wide trace memory only once using the extra bit to indicate that in addition to this word a (n+1)-bit wide number, generated by a counter and specifying the number of times this particular word successively entered the system, is stored in the next trace memory location. This facility can take care of a situation where the MPU is executing a "test-for-device-not-busy" loop, reading the same status word. However, if more than one input port is employed the benefit of this facility is lost unless either only one such loop exists while the rest of the ports employ interrupts, or the input port identifiers are taken into account during the data compression process and a record of multiple counts is kept (one per port identifier).

Finally, conditional breakpoints can be employed making the above trace memory suppression technique application dependent. That is, the user may introduce non-critical time slots within the program real-time execution (e.g., after the application program enters a computational process during which any communication with the real-time world is suspended). Then, the debugging system may halt the target MPU not only when an error condition has been detected, but also as soon as one of these time slots is reached (and this is sensed by the conditional breakpoint mechanism). Then, a full system state snapshot (intermediate snapshot) is taken and program execution together with a new external-stimuli trace is restarted, in which case, the Action-replay execution of the program path will take place from that moment onwards.

Clearly, no matter how sophisticated data reduction techniques and monitoring controls are employed, the trace memory overflow problem still exists if the real-time monitoring process of the input ports is performed for long periods of time. Then, the only solution is to increase the trace memory size.

### 3.1.2 Interrupt Monitoring

**Interrupt Synchronisation** : The case of interrupt synchronisation is rather different from that of the input port data (# 3.1.1). During the real-

time execution, a hardware interrupt is not triggered directly by an MPU activity (as with the accessing of an input port register) but by an asynchronous external event. Therefore, monitoring information about the type of the interrupt in question is not sufficient for synchronising the interrupt with the Action-replay execution.

The obvious way of accomplishing the required interrupt synchronisation is via the **Time-synchronisation method.** That is, to employ a counter for measuring the time elapsed between two successive interrupts and, as soon as an interrupt is sensed, to store this information in the **Interrupt Trace Memory** together with some data indicating the interrupt type. This time interval must not be measured in terms of real-time but either in terms of clock-cycles (for clock-synchronised bus architectures), or in terms of bus activities (for asynchronous bus architectures); only then strict synchronisation is kept between a particular interrupt and the program path execution. For example, assuming an 1MHz clock, a 32-bit counter can measure a time interval of up to 71 minutes and a (32+n)-bit wide trace memory is required, where "n" is a binary number specifying the interrupt type (hence, an 1-bit number is sufficient for the two M6800 interrupt inputs while each of the eight "restart" addresses of the INTEL8080/5 may be encoded into a 3-bit number).

However, sensing the occurrence of an interrupt, while monitoring the MPU interrupt inputs, is not easy. That is, assuming that a race condition exists between two interrupts during the real-time execution, these interrupts may or may not be handled in the same order during the Action-replay. For example, in the case of the M6800 many special conditions of interrupt responses must be taken into account [37, A9], such as :

> "If IRQ occurs during an SWI instruction, the pulse will be lost because SWI clears the interrupt latches",
> "If IRQ and NMI are active concurrently, the MPU will recognise NMI. In so doing, the interrupt latches are reset",

"If an NMI occurs while a SWI is being executed, the interrupt vector will be retrieved from the IRQ location", etc.!

In addition to the above problem, an interrupt may occur very close to its recognition point (usually, it is latched in the MPU during the last cycle of an instruction). Then, the interrupt setup time (Ts) must be taken into account, making sure that this particular interrupt has or has not been latched-in (e.g., Ts = 200ns minimum for the M6800 and Ts = 120ns minimum for the INTEL 8080/5). This implies that the interrupt lines must be sampled at exactly "Ts" nanoseconds prior to the interrupt recognition point. However, sensing this point in time is not easy since the last instruction cycle must be detected first.

The above problems can be solved by monitoring the "interrupt acknowledge" MPU line instead of the corresponding interrupt line. This way, changes on interrupt lines which do not influence program execution (e.g., because the interrupt mask is disabled) are not monitored. However, some MPU makes do not provide an "interrupt acknowledge" signal.

In software terms, an interrupt is acknowledged the moment the first instruction of the corresponding routine is executed. Therefore, a more general way of sensing that an interrupt has occurred is to decode the "origin" address of the corresponding interrupt routine (assuming that interrupt routines are entered only via the MPU interrupt mechanism and never via a "jump" instruction from elsewhere in the program).

Another way of sensing that the MPU is responding to an interrupt is to decode the interrupt vector addresses as they appear onto the address bus during the MPU interrupt servicing sequence (# 5.4.1); however, in this case the user program may only write and not read from the interrupt vector location.

An alternative interrupt synchronisation may be achieved via the **Address-synchronisation method.** That is, instead of tracing the interrupt type and its arrival time, as with the time-synchronisation method, a history of the entire program

path is recorded in terms of addresses; the idea being, to synchronise the interrupt in question to the address bus activity.

An interesting side effect of the Address Synchronisation Method is that the above trace information may be used for verifying the Action-replay process itself (# 3.3).

**Interrupt Trace Memory Overflow** : If the time-synchronisation method is employed, either the timing counter or the interrupt trace memory may overflow resulting in the loss of the required synchronisation. In order to delay the timing counter from overflowing instruction cycles, instead of MPU clock cycles, can be used as time units. However, these cycles are not easily detected in some MPU types. Alternatively, code "fetch" cycles can be used as time units which are easily detected by finding out whether the current contents of the address bus correspond to the memory section which holds the program code and not data. Then, a 32-bit counter can measure a time interval of up to 2**32/(600*1024) = 1 hour 56 minutes (where 600kbytes/s is a typical amount of executable code for an 1MHz driven M6800 MPU (Appendix A)).

Obviously, the address synchronisation method involves the tracing of a vast amount of addresses (more than 1Mbyte/s in average for an M6800 driven at 1MHz). As it is shown in Appendix A, this amount of traced information may be reduced considerably (less than 30kbytes/s), but only at the expense of complex hardware.

Intermediate snapshots of the target system state, taken during the non-critical time slots within the real-time execution (# 3.1.1), may also be used for solving the Interrupt Trace Memory overflow problem. However, as with the Data Trace Memory, it may be necessary to increase the memory size in order to cope with certain applications.

### 3.1.3 DMA Input Data Monitoring

**DMA Input Data Synchronisation** : Special consideration must be applied to the synchronisation of data entering the target system memory via DMA techniques. That is, the incoming DMA data must be monitored and stored in the trace memory. However, the input port data synchronisation method (# 3.1.1) cannot be applied here, since the DMA activity is transparent to the target program after the initialisation of the DMA controller has taken place and since the operation of the DMA controller cannot be easily synchronised with the program operation.

Data entering the system memory via DMA influences indirectly the program execution; that is, only after it has been accessed by the MPU. Hence, by assuming that the address range corresponding to the block of memory which accepts the DMA data (namely the DMA buffer) is part of the "memory-mapped input/output" address space, the memory read-cycles which address this memory block long after the data entered the system can be treated as "port input cycles". Thus, data entering the MPU via the DMA section of the target system memory comply with the same rules applying to data entering through an input port as far as data-compression and Action-replay synchronisation are concerned.

In implementation terms, a programmable decoder can be employed which, having been loaded with the address limits of the DMA buffer, generates a signal indicating that the DMA buffer is currently accessed. Only then the data in question is monitored and stored in the Port Trace Memory (# 5.3.3). It should be noted, however, that any status returned to the MPU by the DMA controller during the original program execution should also be monitored so that synchronisation with the program execution can be achieved during the Action-replay.

**DMA Trace Memory Overflow** : Assuming that the above mentioned DMA data monitoring technique is employed, the port input data compression technique (# 3.1.1) may be applied in the DMA data monitoring process. However, due to the fact that DMA is usually associated

with high data rates, the DMA trace memory is bound to overflow quickly in which case trace memory expansion should be considered.

## 3.2 Program Path Action-replay

Having recorded in real-time the external-stimuli sequence which influences the target program behaviour, Action-replay debugging may now commence. As will be shown in the sections which follow program path Action-replay may be implemented in a variety of ways.

### 3.2.1 Hardware Simulator Approach

The ideal Action-replay "simulator" from a speed and reliability point of view is the MPU itself; the required re-execution may be performed at any speed up to the "real-time" one while there is no need to worry about the correctness of a software simulator.

In particular, during the Action-replay of the program path in question, the microcomputer system is isolated from the outside real-time world by disabling all input ports and interrupt lines; this is why the initial state of these ports need not be monitored prior to the real-time execution. Then, both the MPU and memory states are reset either to their initial state or to any other prerecorded intermediate state. Finally, the MPU is forced to re-execute the program path by placing onto the MPU data bus the data/status, which corresponds to the current input port cycle, and by activating the interrupt line, which corresponds to the interrupt-type (found in the interrupt trace memory), in synchronisation with the execution (# 3.1). During subsequent Action-replays any requested Program Status Information is monitored (#3.4).

During Action-replay the target computer output ports are also disabled unless it is required to debug the user application hardware; that is, in certain cases of open-loop control

systems it is possible during Action-replay to initiate the original sequences of operations in the target application hardware, by allowing the control signals to influence the external hardware as normal, and to study these sequences by employing storage oscilloscopes, logic analysers, etc.

### 3.2.2 Software Simulator Approach

A software simulator of the target MPU can be employed alternatively. Then, having initialised the simulated MPU in accordance to the prerecorded initial/intermediate target MPU state, Action-replay may commence. In particular, the simulator accepts as input the target MPU source code and the external stimuli sequence (# 3.1) and provides an "Action-replay" of the program path in question, during which Program Status Information is monitored (# 3.4).

However, unless the simulator correctness has been proven without any doubt, there is no guarantee whatsoever that the same program path as that followed during the original execution can be reconstructed during Action-replay.

An additional problem is that of the target MPU to simulator speed ratio. This is a critical consideration especially since it may be necessary to initiate multiple Action-replays for additional Program Status Information monitoring (# 3.4).

Finally, the MPU Action-replaying simulator may function either within the target MPU itself (resident approach), or within a host computer (cross-approach). The latter case is considered more advantageous (# 4.2), because a host computer may provide a more comprehensive range of facilities than the target microcomputer.

## 3.3 Action-replay Verification

The correctness of the Action-replay process must be checked since there is nothing worse than a debugging tool which cannot be trusted. Therefore, a mechanism is required which detects that the program path followed during the Action-replay execution is diverging from the original program path.

The **address-synchronisation method** (# 3.1.2), provides a simple and foolproof consistency check between the original execution and the Action-replay execution. That is, each program node generated during the Action-replay execution is compared with the corresponding program node found in the pre-recorded real-time program node record. Therefore, no other information is required for the consistency checking process than that which already has been recorded.

If however the **time-synchronisation method** has been chosen, then it is impossible to check for consistency unless some additional information is recorded. The most economical solution is to provide an **address label** (in terms of the Program Counter register contents) for each interrupt record and during Action-replay to check for consistency between this label and the current address shown on the address bus just before the interrupt service routine is entered. Correspondingly, a **time-label** (in terms of interrupt timer contents) must be provided for each input data record.

## 3.4 Program Status Information Non-real-time Monitoring

The program status information monitoring may be split into two distinct tasks, both of which are user programmable (but should also have a default state), namely the **Monitoring Conditions Evaluation** and the **Program Status Information Formating and Displaying** tasks.

### 3.4.1 Monitoring Conditions Evaluation

The Program Status Information monitoring process is controlled via a number of user defined logical conditions (that is, conditional breakpoints relating program addresses, variables, constants, as well as events), the basic purpose of which is to minimise the amount of Program Status Information down to the user current requirements.

In particular, prior to the Action-replay activation, the debugging system's **Monitoring Conditions Evaluator** finds the type of operands which are involved in the conditions themselves and activates the appropriate software hooks which are controlled by the **Condition Operands Monitor.** Then, the Action-replay process is activated and the required operands are obtained and passed to the Evaluator which, in the event of finding a condition "true", interrupts the Action-replay execution and enables the **Program Status Information Monitor.**

The above organisation may provide tracing of a particular memory location (program variable), the flow of instructions but not their results (or vice-versa), successful Branch/Jump instructions, program statistics, etc. In addition, tracing may be enabled only for an execution period specified by code address limits, real-time limits, etc.

A possible implementation of the above ideas is given in reference 9.

### 3.4.2 Program Status Information Formating and Displaying

The Program Status Information Monitor accesses the required information fairly easy when the software simulator approach (# 3.2.2) is employed. However, in the case of the hardware simulator approach (# 3.2.1) the MPU registers are usually located within the MPU chip itself and this presents an information access problem. By generating a dummy interrupt as soon as an address breakpoint is "hit", the MPU registers may be forced out of the MPU chip and into the microcomputer stack residing in the microcomputer memory either by the interrupt

acknowledge sequence or by the interrupt routine itself (special care must be taken so that such a mechanism is invisible to the target system [28]). This information is then forwarded, together with any other information that might have been requested, to the **Program Status Information Displaying Formater** which is also operating in accordance to user specifications.

Any output generated by the Display Formater must be related to the source program ("source language debugging technique" [41]) via program variable names and labels, instruction numbers, source-file line numbers, etc. Such a user interface implementation requires modification of the assembler/compliler. These are discussed in section #7.3.

## 3.5 Correction Verification

The user studies the program behaviour via a sequence of Action-replays during which different monitoring conditions are employed; their selection depends entirely on the debugging information required at the time. Having located an error, the source file is updated, re-assembled and the new program version is executed in real time. The correction may then be verified by initiating an Action-replay cycle during which the behaviour of the corrected section of the program is studied via further Program Status Information monitoring.

The proposed correction verification might be impossible in some cases because of the difficulty of forcing the target system to follow the required program path (unrepeatability problem (# 2.1)).

## 3.6 Summary

The Action-replay Debugging Technique provides a "transparent" solution to both the "execution unrepeatability" and the "program status information accessing" problems encountered in

non-deterministic, real-time systems by allowing real-time debugging to take place within a non-real-time environment.

# CHAPTER 4

## 4 ACTION-REPLAY DEBUGGING SYSTEM

Before discussing the basic organisation of the Action-replay Debugging System, a summary of the various system functions, derived from the ideas presented in Chapter 3, will be given.

### 4.1 System Functions

**a)** control target system (start/stop execution).
**b)** evaluate conditional breakpoints.
**c)** capture target system initial/intermediate state.
**d)** monitor external stimuli during real-time execution.
**e)** recover to system state captured during function "c".
**f)** activate Action-replay and reconstruct external stimuli.
**g)** verify Action-replay.
**h)** evaluate Program Status Information monitoring conditions.
**i)** monitor Program Status Information.
**j)** format and display Program Status Information.
**k)** either return to function "e" for further Action-replay iterations or correct discovered error.
**l)** verify error correction.

### 4.2 System Organisation

Because the Action-replay debugging system must provide ample computing power for the realisation of complex functions (# 4.1), it is better if its structure is based on the semi-resident approach (# 1.2.2) as shown in Figure 3.

The user completes the initial development of the target program, employing the resources of the **Host Computer** (e.g., file manager, file editor, code linker/loader, cross-assembler, high level language compiler). Before the prototype program is executed in the **Target Microcomputer** some basic debugging may be performed employing a cross-simulator (# 2.1). Thereafter, the program is transferred to the target computer (down-line loading) where debugging under real-time conditions may commence employing the Action-replay debugging technique. However, during the required external stimuli monitoring, the host computer cannot respond quickly enough to the real-time activities of the target MPU (# 3.1). By employing an intelligent interface between the host computer and the microcomputer, namely the **Host-MPU Interface** (Figure 3), it is possible to compensate for the slow host response; that is, all real-time tasks are controlled by the interface.

The interface-to-microcomputer connection is at the MPU-bus level (# 3.1). However, the type of the interface-to-host computer connection depends on whether the host-MPU interface does or does not include the required external-stimuli trace memory. That is, if the trace memory is located within the host-MPU interface, the interface-to-host data traffic is low and a serial link may be employed while, if the trace memory is located within the host computer system, the data traffic increases considerably (a rate of more than 20kbytes/s is expected after the data-compression has taken place) and it is necessary to employ a parallel connection between the host computer and the interface.

The external-stimuli trace memory does not require to be entirely in RAM. For example, assuming that enough RAM trace memory exists in the host computer for speed buffering purposes, the trace memory may be located thereon and, in addition, some sort of back-up memory may also be employed (e.g., floppy discs). Then DMA techniques, employed within the host computer system, may continuously dump the contents of the RAM trace memory into the back-up store.

An advantage of the above method is that some of the data-

compression functions may be implemented in host software (# 5.1). On the other hand, however, the host computer may run a multiuser operating system and may not have enough time to respond even to the trace-memory service requests. In such cases, DMA techniques must be used within the host computer system if it is chosen to reserve part of its store for the external-stimuli trace memory.

Similarly, disc back-up storage may also be employed if the trace memory is located within the host-MPU interface. However, in this later case the hardware complexity of the interface increases since all the data compression and trace memory control actions are to be implemented thereon. In addition, the propagation delay of the host-MPU interface logic becomes an important design consideration, especially since all these actions must be undertaken in less than 1000ns which is a typical MPU cycle period. By employing ECL technology and associative memory techniques in the implementation of the interface its overall propagation delay can be improved.

In either of the above cases, the host-MPU interface flexibility is also an important design consideration. Microprogram techniques can be employed within the interface control section in order to provide interface adjustability to a variety of MPU systems [28].

From a functionality point of view (# 4.1), the host-MPU interface may be divided into six distinct sections as shown in Figure 4 :

**1)** Interface + MPU Control (microprogrammable).
**2)** DMA Interface.
**3)** MPU-Registers Access Interface.
**4)** Monitoring Interface.
**5)** Action-replaying Interface.
**6)** Verification Interface.

Past implementations of hardware/software consoles have dealt with interfaces 1, 2 and 3 successfully (# 1.2) and, therefore, these interfaces are omitted from this discussion in

contrast to interfaces 4, 5 and 6 which are closely related to the Action-replay debugging technique. It should be noted that interfaces 5 and 6 are only required if the hardware simulator approach (# 3.2.1) is employed during Action-replay, otherwise their function is implemented in software.

## 4.3 Monitoring Interface

The **Monitoring Interface,** the basic characteristics of which are given in section #3.1, is divided into the **Input Port Section** and the **Interrupt Section.** A description of these two sections now follows with reference to Figure 5.

### Input Port Section

The **Port Input-cycle Sensor** forwards the "port type" of the currently activated port to the **Port-input Compressor** which consists of an "Input Counter" and a "Last-input Buffer" for each input port of the target system. The Input Counter specifies the number of times an identical input entered in succession through a port, while the Last-input Buffer contains the last data/status read-in through that port. In particular, the Port-input Compressor operates as it is described below.

During a "port input cycle" the "current input" of the port is captured and compared with the "last input" corresponding to this port. Then,

**a)** if they are found to be identical, the "input count" which corresponds to the input port is incremented. If this count overflows, the "port type" is transmitted to the trace memory indicating that the "input counter" of this particular port has overflowed.

**b)** if they are found to be different, the "current input" is stored in the "last-input buffer" while the "last input", together with the corresponding "input count" (if one exists), is transmitted to the **Data Trace Memory;** thus forming an "input-port entry".

**Interrupt Section**

The **Interrupt Timer** measures the interval between two successive interrupts in terms of the timing unit which is provided by the **Time Unit Sensor.** The **Overflow Counter** is incremented each time the Interrupt Timer overflows. The **Interrupt Acknowledge Sensor** detects the occurrence of an interrupt and clears the Interrupt Timer and its Overflow Counter after the "interrupt type" (generated by the **Interrupt Encoder** ), the "overflow count" (if one exists) and the "interrupt time" (generated by the Interrupt Timer) have been transmitted to the **Interrupt Trace Memory;** thus forming an "interrupt entry".

It should be noted that apart from encoding the interrupt identity into the "interrupt type", no other technique may be used for minimising the external-stimuli information corresponding to the system interrupts.

## 4.4 Action-replaying Interface

Assuming that the "software simulator approach" (# 3.2.2) is employed, no extra hardware facilities are required; Action-replay may commence within the host computer system by simulating the MPU operation and the external-stimuli activity. If however the "Hardware Simulator Approach" (# 3.2.1) is employed during the Action-replay phase, the **Action-replaying Interface** is required.

In order to minimise the hardware complexity of the Action-replay Interface, and since time is not a critical consideration during the Action-replay process, the expansion of the compressed external-stimuli may be performed via software; that is, within the host computer. Hence, the Action-replaying Interface is divided into three sections, namely the **Input Port Section,** the **Interrupt Section** and the **MPU Control Section.** A description of these three sections now follows with reference to Figure 6.

**Input Port Section**

The **Port Input Cycle Sensor** enables the **Output Latch** forcing its contents onto the microcomputer data bus (all ports are disabled and, therefore, no other data is currently present on the bus). Then, the "next port input", is requested from the host computer and the MPU is forced to enter a "wait" state by the **MPU Controller,** thus inhibiting the next port-input cycle. As soon as the requested information arrives, the "start" signal is generated and the Action-replay program execution continues.

**Interrupt Section**

The time interval between interrupts is measured in a similar way as it was measured during the monitoring session; the difference being that the Timer is disabled each time the MPU enters the "wait" state. The "interrupt time", stored in the **Interrupt Latch,** is compared with the Timer's current contents only after the "timer overflow count", decremented within the host via the "overflow" signal, has expired. As soon as equivalence is detected by the **Comparator,** the **Interrupt Decoder** is enabled so that the interrupt line corresponding to the "interrupt type" is activated. Then, the MPU is forced by the **MPU Control** to enter a "wait" state, the Timer is cleared and the "interrupt time/type" for the next interrupt is requested from the host computer. The MPU continues with the execution of the interrupt routine only after the interrupt section of the Action-replaying Interface has been provided with this information (start signal).

## 4.5 Verification Interface

If Action-replay verification is required, extra information must be captured during the monitoring phase, as described in section #3.3; that is, each "interrupt entry" and each "input-port entry" must also include an "address label" (i.e., the current address) and a "time label" (i.e., the current interrupt timer contents) respectively. In the first case an "address latch" is required between the MPU address bus and

the trace memory, while in the second case no extra circuitry is required since the connection between the Interrupt Timer and the trace memory already exists in the monitoring interface (Figure 5). Additional control circuitry is needed however in both cases.

As the above information is progressively obtained during the Action-replay phase it is compared with the corresponding information acquired during the monitoring phase. Action-replay is aborted if any inconsistency is sensed which implies that the Action-replay program path has deviated from the real-time execution program path; a situation that may arise either if an intermittent error exists in the target microcomputer, or if the Action-replay System itself malfunctions.

## 4.6 Summary

The structure of the Action-replay system is based on the semi-resident approach in order to make use of the facilities of existing development systems.

The External-stimuli Monitoring is performed under the control of an intelligent Host-MPU Interface while Action-replay is performed under the control of the host computer and either via an MPU software simulator or via the target MPU itself.

# CHAPTER 5

## 5 ACTION-REPLAY SYSTEM IMPLEMENTATION

An "Action-replay Debugging System" prototype, based on the "semi-resident" approach discussed in Chapter 4, is designed and constructed in order to establish that the realisation of the "Action-replay Debugging Technique" is possible and that such a system is in practical terms useful.

Due to time constraints, only ideas bearing originality are implemented, the design complexity is kept at minimum levels and existing resources within the Digital Systems Laboratory (DSL) are employed whenever possible in the construction of this prototype system. Furthermore, some knowledge is assumed on the various components employed in the design (reference to literature is given whenever possible).

### 5.1 Assumptions

The Action-replay prototype system has been developed according to the assumptions given below :

**a)** an **M6800 microprocessor** [36] is assumed to be the target computer.

**b)** an **LSI11/23 minicomputer** [52] is assumed to be the host computer.

**c)** DMA is not used in the target microcomputer.

**d)** the target program is either kept in ROM or is loaded in RAM via existing methods; hence, no down-line loading is required between the host and the target computers.

**e)** target program and data are kept separately so that the "fetch cycles" can be recognised during execution (# 3.1.2).

**f)** no intermediate system state is captured; instead, the "Reset" system state is always used as the "origin" for both the "External-stimuli Monitoring" and the "Action-replaying" phases.

**g)** data reduction during the Monitoring phase is independent of the "port type"; i.e., it is assumed that data streams entering via different ports constitute a single "input-port data" stream.

**h)** Action-replaying of the program path is performed in the target microcomputer.

**i)** only the "Monitoring" and "Action-replaying" interfaces will be implemented.

**j)** the HP-1610B Logic Analyser is employed for "Action-replay Verification" and "Program Status Information Monitoring".

**k)** the DSL's prefabricated integrated circuit boards are employed in the construction which are plugged into sockets at the back of a special purpose frame and interconnected at the front of this frame with pin-ended wires (flat-cables, twisted-pairs, etc.).

## 5.2 Target Computer System

The M6800 microcomputer system designed and constructed within the DSL [17] is employed as the target computer.

### 5.2.1 Target Computer System Hardware

The DSL M6800 microcomputer system is driven at 1Mhz clock rate and includes a Peripheral Interface Adaptor (PIA) with "memory mapped" registers, 256 bytes of ROM which accommodates a "Paper Tape Driver" program, 1kbyte of RAM and a "Paper Tape Reader" (PTR) which is interfaced via the CA1 signal (device busy) and CA2 signal (data request) to the A-side of the PIA and which can reach a speed of 150 characters/s. MPU control is provided via the "Control Console" (a hardware console), which also allows DMA via 16 "address/data" switches and 16 "address/data" LEDs.

The PTR mentioned above and the following two components simulate a non-deterministic, real-time application (# 5.2.2) which is assumed to be the application under development :

1) An 8-bit "Counter" which is connected to the B-side of the PIA (the CB2 PIA signal is connected to the clock input of the counter) and whose current value is read by the NMI/IRQ interrupt routine (# 5.2.2).

2) A "Pulse Generator" which provides the M6800 system with two asynchronous sequences of NMI and IRQ random interrupts; the NMI pulse is directly applied to the NMI input of the M6800 MPU while the IRQ is generated by the PIA responding to a negative-going edge on its CB1 input.

In addition, some circuitry which allows the M6800 microcomputer system to accept two additional control signals is employed, namely the "disable input ports" and the "disable memory" signals.

### 5.2.2 Target Computer System Software

It is assumed that the PTR driver program, which is in fact a fully tested program, is the software under development.

The PTR driver program loads object tapes into the M6800 memory employing "handshake" control. The "test for device

not busy" loop consists of the following instructions :

```
LOOP: TST  PIAST   test status
      BPL  LOOP    device busy? yes: repeat.
      LDAA PIADT                 no : read next data.
```

Thus, a "status" byte enters the system every 10 microseconds (TST = 6 clock cycles, BPL = 4 clock cycles) which implies a maximum input rate to the Action-replay Monitor Interface of about 100kbytes/s.

Since a target application employing interrupts is in fact required, the PTR driver program has been modified so that can handle interrupts. In particular, a small program is included which initialises the NMI/IRQ vectors, programs the B-side of the PIA for interrupt operation, enables the IRQ interrupt and transfers control to the PTR driver. In addition, two interrupt routines are included, namely the **IRQ Routine** and the **NMI Routine.**

The IRQ routine reads the current contents of the Counter (Figure 7) via the B-side of the PIA (an action which also increments the Counter via the CB2 signal), displays this value on the M6800 Control Console LEDs (8->15), loads the Index Register with a 16-bit "delay" value preset on the Control Console switches and decrements it to zero before executing an RTI instruction returning control to the PTR driver program.

The NMI routine performs nearly the same actions as the IRQ routine, the difference being that the counter value is displayed on the low LED byte (0->7) of the MPU Control Console; thus giving a visual indication of which of the two interrupts was just serviced.

The minimum "delay" value permitted is "1", which implies that on the occurrence of an interrupt each interrupt routine instruction is executed only once. In this case the propagation time through either the IRQ routine or the NMI routine is 50 microseconds. Hence, assuming that nested interrupts are

not allowed, the above application may handle interrupts at a maximum rate of 20 kHz.

As will be shown later on (# 5.5) the 100kbytes/s input rate is reduced by the Port Input Compressor of the Monitoring Interface by a factor of 130, while, unfortunately, the information generated while monitoring the system interrupts cannot be compressed (# 4.3).

It should be noted that the SWI of the M6800 is not included in the above arrangement since it is invoked from within the system and, therefore, does not require any special treatment as far as the Action-replay Technique is concerned.

## 5.3 Host Computer System

An LSI11/23 minicomputer is assumed to be the host computer but in fact the filing system, the editor and the M6800 cross-assebler residing at the University's mainframe computer were actually used by connecting the LSI11/23 to the local node of the University network.

### 5.3.1 Host Computer System Hardware

The LSI11/23 minicomputer consists of six boards :

1) A KDF11-AA board, including the LSI11/23 processor, the Memory Management Unit (MMU) and the ODT microprogram (a resident debugging monitor which accepts commands for DMA and execution-control and displays information onto a VDU screen).

2) An MXV11-AC board containing 32kbyte RAM, 2kbyte ROM (accommodating a program which enables the VDU to be used as a terminal to the University's mainframe) and two serial ports, one of which is used for the VDU while the other for connecting to the Newcastle University computer network.

**3)** A DRV11 board, consisting of a parallel input port and a parallel output port (16-lines each). A pair of signals is provided per port for "handshake" control, namely the "Request A"-"New Data Ready" pair and the "Request B"-"Data Transmitted" pair.

**4)** Three MSV11-DD boards (64kbyte RAM each) which increase the system RAM to 224kbyte.

### 5.3.2 Host Computer System Software

The basic purpose of the LSI11/23 program is to drive the Host-MPU Interface during the Monitoring and the Action-replay stages of a debugging cycle. In particular, it performs the following tasks :

**a)** initialises the MMU registers so that the virtual 64-kbytes system memory is mapped to the 224-kbytes physical memory in such a way so that the latter is split into the "program section", the "input/output section" (memory-mapped input/output) and the "trace memory section".

**b)** initialises the Monitoring Interface.

**c)** drives the input DRV11 port, which receives data sent by the Monitoring Interface (i.e., data which has been "compressed" within this interface) and stores this data into the LSI11 trace memory. The time response of this section of the program is critical since if a data word arrives at the DRV11 input before the previous data word has been read in, this later data is lost and, consequently, no Action-replay can progress beyond this point of program execution (# 3.1). Therefore, this program section uses direct addressing ("register mode") as much as possible thus producing efficient code; a delay of 17 microseconds per DRV11 input is achieved, with 15 microseconds additional delay every 8kbyte of input per trace memory, when virtual to physical page relocation is performed.

**d)** initialises the Action-replay Interface.

**e)** drives the output DRV11 port, and consequently the Action-replay Interface, with information generated by "expanding" the trace memory contents.

Two interrupt routines are also included; they employ software counters for keeping track of overflows in hardware counters during the Monitoring and the Action-replay phases respective (# 5.4.1).

### 5.3.3 Trace Memory Organisation

The LSI11 Trace Memory consists of 216 kbytes RAM organised in two 16-bit wide trace-memories/stacks of variable length (minimum 8kbyte, maximum 208kbyte), namely the **Port Trace Memory** and the **Interrupt Trace Memory.** "Paging" is performed in such a way so that these two trace memories start at opposite ends in the physical memory and, as they fill up, they extend towards each other. As soon as their stack pointers acquire the same physical address value, the system trace memory overflows and monitoring is aborted (# 3.1).

**Port Trace Memory**

Up to three different entries may be accepted by the Port Trace Memory :

**1)** an 8-bit **port-input entry** which may be either a port data or a port status byte. If this entry is entered twice in succession, entry number 2 below follows.

**2)** a 16-bit **input-count entry** specifying how many times the last data/status entered through the port. If the MSB of this entry is "set", entry number 3 below follows.

**3)** a 16-bit **overflow-count entry** specifying how many times the "input count" has overflowed.

Entries 1 and 2 above enter the system via the DRV11 LSI11

port (reqb), while entry 3 is formed by the LSI11 computer software (# 5.4.1).

The following five/six entries are required for the PTR application :

**n:** - - - - - - - -
**n+1:** "PTR BUSY STATUS"
**n+2:** <as n+1 entry> ("input count" follows)
**n+3:** "I/O, INPUT-COUNT"
**n+4:** "OV-COUNT" (if n+3 entry's MSB=1)
**n+5:** "PTR NOT-BUSY STATUS"
**n+6:** "PTR DATA"
**n+7:** - - - - - - - -

An example of the Port Trace Memory state at the end of a test monitoring-session can be seen on Table 1, section #5.5.1.

**Interrupt Trace Memory**

Up to two different entries may be received by the Interrupt Trace Memory for each interrupt, namely the 16-bit **interrupt entry** and the 14-bit **overflow-count entry.**

The "interrupt entry" enters the system via the DRV11 LSI11/23 port (requa) and consists of the following two fields :

**1)** The 2-bit **interrupt-type field** (entry bits 15 and 14) which indicates that the interrupt in question is of the IRQ type (bit-15 is set) or the NMI type (bit-14 is set).

**2)** The 14-bit **code-fetch count field** which specifies the time elapsed between two successive interrupts in terms of "code-fetch cycles" (# 3.1.2).

The "overflow-count entry" is formed within the LSI11/23 (# 5.4.1) and specifies the number of times that the "code-fetch count field" has overflowed (bits 15 and 14 are cleared identifying this as an "overflow count entry" and not as an "interrupt entry").

The following two entries are required per interrupt for the PTR application (n+2 is required only if CF-COUNT has overflowed at least once) :

**n:** - - - - - - - -
**n+1:** "10/01, CF-COUNT"
**n+2:** "00, OV-COUNT"
**n+3:** - - - - - - - -

An example of the Interrupt Trace Memory state at the end of a test monitoring-session can be seen on Table 2, section #5.5.2.

## 5.4 LSI11/23-M6800 Interface Implementation

As can be seen in Figure 7, the LSI11/23-M6800 Interface is divided into three basic sections, namely the **Monitor Interface** section, the **Action-replay Interface** section and the **Interface Control** section.

### 5.4.1 Monitoring Interface Implementation

About 150 integrated circuits have been employed in the implementation of the Port Input Section and the Interrupt Section of the Monitoring Interface (# 4.3). The structure of both these sections is now described with reference to Figure 8. A description of the Host-MPU Interface Control Section which is relevant to the Monitoring Interface is also given.

**Port Input Section**

Latches A and B hold the "last" and "current" port data/status inputs thus forming a 2-level, 8-bit wide pipeline. An 8-bit Comparator decides if these two inputs are equal or not. In the former case the 15-bit Input Counter is incremented. In the latter case the "port-input entry" (i.e., the "last" input) is directed via a 3-to-1 Selector to a FIFO buffer

(path 2) and, if an "input count" exists, an "input count entry" follows immediately afterwards (path 3).

**Interrupt Section**

In addition to the "port-input entry" and "input-count entry", the FIFO Buffer (18-bits * 40-words) accepts the "interrupt entry" (path 1), which consists of the 2-bit "interrupt-type" field (generated by the Interrupt Decoder when either the "FFF8" or the "FFFC" interrupt vector addresses appear on the MPU address bus) and the 14-bit "code-fetch count" field (generated by the Code-fetch Counter).

The Code-fetch Decoder provides the timing-unit which drives the Code-fetch Counter. Ideally, this timing unit is a processor signal which identifies all opcode-fetch cycles during execution. For example, the INTEL 8080 MPU provides an 8-bit status on the Data bus during a SYNC pulse, which identifies the type of the current machine cycle; hence, the opcode-fetch cycle can be identified and the timing-unit is that of an "instruction cycle". However, the M6800 MPU architecture does not provide such a signal. Instead, the timing-unit is that of a "code-fetch cycle" which is the next best alternative after the "instruction cycle"; there are about 2.3 code-fetch cycles in average per instruction (Appendix A).

The Code-fetch Decoder is in fact formed by two registers, holding the upper and lower address limits (Au,Al) of the memory section which accommodates the program code, and two comparators which during a memory-access-cycle check for $Au<A$ and $A<Al$ respectively ("A" is the current value of the Address bus). The timing pulse is then generated and the Code-fetch Counter triggered only if both the above conditions are found to be "true".

**Interface Control Section**

Each time either a "port-input entry" (ldinput) or an "interrupt entry" (ldint) arrives at the FIFO inputs, the FIFO "input register empty" signal is tested and if it is found to be "false" the Monitoring Interface is disabled (fail) and

Monitoring is aborted because a FIFO entry is lost (# 5.3.2).

The signals "ldinput" and "ldint" are also latched into the FIFO in order to evoke the proper LSI11 response via the REQA and REQB signals (# 5.3.3) each time an entry reaches the FIFO outputs and, therefore, the 16 DRV11 inputs.

Finally, as soon as either of the two counters overflows the BEVENT signal is generated and an LSI11 interrupt routine (# 5.3.2) increments the corresponding software counter.

It is not necessary to synchronise a counter overflow with the corresponding "port-input entry"/"interrupt entry" because long before one of the counters overflows for the first time, the LSI program has read all the entries pending within the FIFO and, consequently, the previous "input count"/"code-fetch count" together with the corresponding "overflow count" (if one exists within the LSI11). Therefore, there is no possibility of incrementing one of the software "overflow counters" before its last contents, corresponding to an entry previously transmitted to the FIFO, are read and stored in the trace memory. If such a possibility existed, overflows should be entered into the system via the FIFO immediately after the corresponding count and not via the LSI11 interrupt mechanism.

### 5.4.2 Action-replay Interface Implementation

About 60 integrated circuits are employed in the implementation of the Port Input Section and the Interrupt Section of the Action-replay Interface (# 4.4); parts of the Monitoring Interface, such as the Code-fetch Counter and the Interrupt Decoder, are also used. Due to the lack of space at the DSL board which accommodates the Monitoring interface an additional board is employed for the Action-replay Interface; a number of flat cables connect the two boards together.

A description of the Port Input and Interrupt sections and their part of the Host-MPU Interface Control Section is now given with reference to Figure 9.

**Port Input Section**

While a "port input-cycle" is taking place, the "current" port input, located within the **Output Latch** (Figure 9), is forced onto the MPU data bus via the **Output Buffer.** Immediately afterwards, target program Action-replay execution is suspended so that the Output Buffer is reloaded with the "next" port input; an action initiated by activating the DRV11 Request-B flip-flop which controls the "reqb" signal. In particular, the "next" port input is sent to the Host-MPU Interface by the LSI11 minicomputer, which controls the "expansion" of the "compressed" information residing within the Port Trace Memory (# 5.3.3) via two routines, namely the "Transmit Same Data ´input-count´ Times" (TRDT) routine and the "Call TRDT Routine ´overflow-count´ Times" (TROV). Both these routines are described in detail in section #6.1.

**Interrupt Section**

As soon as the **Code-fetch Comparator** detects that the "code-fetch count", located within the **Interrupt Latch,** equals the current contents of the Code-fetch Counter (Figure 8), the **Interrupt Decoder,** which decodes the 2-bit "interrupt type", is enabled and the NMI/IRQ interrupt is generated. Immediately afterwards, program execution is suspended so that the next "code-fetch" + "interrupt-type" information is transferred from the Interrupt Trace-memory to the Interrupt Latch; an action initiated by activating the DRV11 Request-A flip-flop which controls the "reqa" signal (see "Load Interrupt" command).

**Interface Control Section**

A somewhat autonomous part of the Host-MPU Interface Control Section is that of the **MPU Control** which provides the means for suspending target program Action-replay execution under the following four cases :

**1)** Immediately after the last cycle of 4-cycle instructions, such as the "LDA", "CMP", and "BIT" instructions, when this last cycle is a "port input cycle" and "extended"

addressing mode is used (memory-mapped input/output is employed in the DSL M6800 system).

2) Two cycles after the fourth cycle of the "TST" instruction in the "extended" addressing mode only if this fourth cycle is a "port input cycle".

3) Two cycles after the "interrupt acknowledge" signal which is generated by the Interrupt Decoder (Figure 8) as soon as the FFF8 and FFFC addresses, corresponding to the IRQ and NMI interrupt vectors respectively, appear on the MPU address bus.

4) Finally, Action-replay execution is stopped for program status information monitoring purposes (# 3.4).

The M6800 architecture responds rather strangely to an interrupt when the MPU is in the HALT state; that is, a single instruction is executed as soon as execution is resumed and only then the MPU enters the interrupt sequence. Hence, in order to cover the case where an interrupt is generated during an instruction which activates a "port input cycle" (see example in section #5.4.3), the HALT MPU Sequence may not be used for suspending Action-replay execution. Instead, a "BRA" instruction is executed in the place of the target program instruction which follows the "port input cycle". This "BRA" instruction transfers control to itself (thus forming a small loop, namely the "Branch Loop"). That is, a 4-bit shift register is employed in the generation of the appropriate signals which are required in order to force onto the MPU data bus the opcode (20) and operand (FE which is the 2's complement of -2) of this 2-byte, 4-cycle instruction. Concurrently, the microcomputer memory is disabled so that no other data is present on the data bus (care must be taken not to activate the DISMEM signal while the MPU is in the interrupt sequence since the memory stack is used for saving the MPU status).

The opcode of the BRA instruction is forced onto the MPU data bus during the "fetch" cycle of the next program instruction. However, as mentioned above, in the case of the "TST"

instruction, the "port input cycle" is not the last machine cycle as with all the other instructions and, therefore, this instruction requires special attention. The TST Decoder has been designed which decodes the TST instruction opcode (7D) and provides the required 2 cycles delay if the 4th machine cycle of the TST instruction is a "port input cycle".

The Interface Control Section is initialised by the LSI11 software via the DRV11 CSR0/1 signals. These signals are decoded into the following four **Interface Commands** when the NDR signal is activated (i.e., when the appropriate bits in the DRV11 control register are accessed) :

| **COMMAND** | **TASK** |
|---|---|
| **1) Load Input** | : load "next" port-input byte into the Output Latch, de-activate the Request-B flip-flop and abort the "Branch Loop" if MPU is running (BA=false) and if the Request-A flip-flop is not active (i.e., no interrupt information has been requested). |
| **2) Start/Halt** | : start/stop target program execution. |
| **3) Enable Comparator:** | enable the Comparator on the next Code-fetch Counter overflow. |
| **4) Load Interrupt** | : load "code-fetch count" + "interrupt-type" into the Interrupt Latch (ldint), de-activate Request-A flip-flop, enable Comparator if command "Enable Comparator" has not been received and abort "Branch Loop" if Request-B flip-flop is not active (i.e., no port-input information has been requested). |

During the Monitoring phase any Code-fetch Counter overflow is recorded and stored in the Interrupt Trace Memory via an LSI11 interrupt routine (# 5.4.1). An interrupt routine is also provided during the Action-replay phase so that the host computer can keep track of the Code-fetch Counter overflows. The

"Load Interrupt" command is issued when no other overflow is expected; that is, only then a comparison between the "code-fetch count" and the current value of the Code-fetch Counter is performed in order to enable the Interrupt Decoder in the event of a "match". However, it is possible that the remaining time after the last overflow and before the required interrupt generation is less than 4 code-fetch cycles. This implies that the interrupt may need to be generated during the execution of the "current" instruction; an M6800 instruction may have a length between 1 and 3 bytes.

The above action is not possible under the arrangement described in the MPU control section above because the M6800 organisation does not permit stopping execution in the middle of an instruction; an action required since enough time must be provided for the LSI11 software to issue the "Load Interrupt" command responding to the overflow interrupt. The "Enable Comparator" command deals with this situation; that is, it allows to issue the "Load Interrupt" command before the last Code-Fetch Counter overflow enabling the Comparator on the occurrence of this overflow (and not with the "Load Interrupt" command as it is done normally).

### 5.4.3 Action-replay Interface Operation

To provide a better understanding of the Action-replay Interface operation, an example is given of how this interface responds to the following situation :

"an interrupt is to be generated during the execution of an instruction which activates an input-port-cycle".

The following actions take place :

**a)** the interrupt is generated as soon as the predefined "code-fetch count" is reached; that is, after the current instruction code has been fetched from the memory (it should be noted that in fact the interrupt in question is generated ["code-fetch count" minus 2] accesses after the last "interrupt acknowledge", because the M6800 interrupt

sequence performs two dummy code-fetches at the beginning of the interrupt sequence).

**b)** the current instruction is executed initiating a "port input cycle" during which the Output Buffer forces the required port data/status onto the Data bus and

**c)** the DRV11 Request-B flip-flop is activated, requesting the next "port input".

**d)** the M6800 interrupt sequence is then entered.

**e)** thirteen cycles later the "interrupt acknowledge" signal activates both the BRA Shift Register (thus forcing the MPU to stop program execution by entering the Branch Loop) and the Request-A flip-flop (thus requesting the next "interrupt time/type").

**f)** the LSI11 software, responding to the DRV11 Request-A/B signals, transmits the requested information to the Interrupt Latch (ldint) and the Output Buffer (ldinput) correspondingly.

**g)** finally, the Branch Loop is aborted and program Action-replay execution continues with the first instruction of the corresponding interrupt routine.

## 5.5 Performance Evaluation

The performance of the Action-replay Prototype System was investigated via two experiments during which it is assumed that the PTR driver program is the real-time software under development and that there are no malfunctions in either this software or in the target system hardware; a malfunction is going to be introduced later on when the Action-replay Debugging Technique itself is evaluated (case study III).

The "count" facility of the HP-1610B Logic Analyser was used for taking "time" measurements and "event occurrence" measurements during both experiments. In addition, the Logic Analyser's "compare" facility was employed for Action-replay

verification (# 3.3) purposes; that is, a sequence of certain types of events (e.g., interrupts) was recorded during real-time execution and then was successfully compared for equivalence with the sequence of the same events during the corresponding Action-replay execution.

Action-replay verification was also performed by loading a program into the M6800 system RAM (while monitoring the PTR program activity), then clearing this RAM area and, finally, activating Action-replay of the original PTR program execution; at the end of the Action-replay the program had been reloaded into the RAM correctly.

### 5.5.1 Experiment A

In the first experiment the PTR program execution is monitored in real-time until the LSI-11/23 Port Trace Memory overflows. The performance of the Port Input Section of the Host-MPU Interface is under test and, therefore, the M6800 interrupts are disabled.

Table 1 shows the state of the first 24 locations of the Port Trace Memory at the end of a test monitoring-session; the Port Trace Memory begins at location 20000 (low end of the overall trace memory) and extends upwards and towards the Interrupt Trace Memory. All numbers are represented in octal.

The PTR "busy status" (054) has been read [13*"overflow count"+23,104+1] times, where each "overflow count" equals to 2**15. The next PTR input is the "not-busy status" (254) followed by the PTR ASCII data (200).

Five/six entries are required per PTR data input. However, better memory utilisation can be achieved since only entries n+3 and n+4 need to be 16-bit wide for the M6800 implementation (an 8-bit machine). The above organisation is chosen not only because it is independent of the word size of the target MPU but also to assist in the debugging of the Host-MPU Interface.

| ADDRESS | | CONTENTS |
|---|---|---|
| 20000 | : | 000054 |
| 20002 | : | 000054 |
| 20004 | : | 123104 |
| 20006 | : | 000013 |
| 20010 | : | 000254 |
| 20012 | : | 000200 |
| 20014 | : | 000054 |
| 20016 | : | 000054 |
| 20020 | : | 001162 |
| 20022 | : | 000254 |
| 20024 | : | 000200 |
| 20026 | : | 000054 |
| 20030 | : | 000054 |
| 20032 | : | 001226 |
| 20034 | : | 000254 |
| 20036 | : | 000012 |
| 20040 | : | 000054 |
| 20042 | : | 000054 |
| 20044 | : | 001221 |
| 20046 | : | 000254 |
| 20050 | : | 000076 |
| 20052 | : | 000054 |
| 20054 | : | 000054 |
| 20056 | : | 001224 |

**Table 1** : Data Trace Memory Example.

**Measurements**

**a) M6800 Clock** : 1.1 microsecond.
**b) PTR speed** : 143 characters/s.
**c) Monitoring time** : 140 seconds.
**d) Action-replay time**: 318 seconds.

Action-replay execution is running 2.27 times slower than real-time execution (d/c).

**e) Code-fetch cycles** : 70,069,312 accesses of code.

**f) PIA input** : 13,871,709 data/status entries (99,083 bytes/s).

20020 bytes of PTR data entered through the PIA (b*c).
5.03 code-fetch cycles per input in average (e/f).

**g) DRV11 input** : 106,496 16-bit entries.

760 entries/s (g/c) or 5.3 entries per PTR data character in average (760/b).

The Monitoring Interface reduced the PIA entries by a factor of 130 (f/g).

### 5.5.2 Experiment B

In this experiment interrupts are taken into account. That is, the PTR program is expanded to include two interrupt routines (# 5.2.2) and the Pulse Generator (# 5.2.1) generates asynchronously a random sequence of IRQ interrupts and a random sequence of NMI interrupts. The PTR is kept switched off throughout this process. The PTR program execution is monitored in real-time until the LSI11/23 trace memory overflows.

Table 2 shows the state of the first 22 locations of the Interrupt Trace Memory at the end of a test Monitoring session; this trace memory begins at the highest location (677776) and extends downwards and towards the Port Trace Memory.

The first interrupt of Type "10" was serviced [475*"overflow count"+3,434] code-fetches after the initiation of the M6800 program execution (each "overflow count" is equal to 2**14 code-fetches). The next interrupt of Type "01" was serviced after [203*"overflow count"+74,704] code-fetches.

| ADDRESS | | CONTENTS |
|---|---|---|
| 677776 | : | 103434 |
| 677774 | : | 000457 |
| 677772 | : | 074704 |
| 677770 | : | 000203 |
| 677766 | : | 103246 |
| 677764 | : | 000255 |
| 677762 | : | 137471 |
| 677760 | : | 000010 |
| 677756 | : | 061505 |
| 677754 | : | 000001 |
| 677752 | : | 116372 |
| 677750 | : | 000007 |
| 677746 | : | 137130 |
| 677744 | : | 000010 |
| 677742 | : | 054243 |
| 677740 | : | 000007 |
| 677736 | : | 123453 |
| 677734 | : | 000001 |
| 677732 | : | 137453 |
| 677730 | : | 000010 |
| 677726 | : | 137417 |
| 677724 | : | 000032 |

**Table 2 :** Interrupt Trace Memory Example.

**Measurements**

**a) Interrupt rate** : 200Hz maximum (5ms/interrupt).
**b) Monitoring time** : 92 seconds.
**c) DRV11 input** : (200 interrupt entries + 200 overflow count entries)/s in addition to the usual input-port entries.
**d) Action-replay time:** 209 seconds.

**a) Interrupt rate** : 5kHz (0.2ms/interrupt).
**b) Monitoring time** : 4 seconds.
**c) DRV11 input** : (5K interrupt entries + 5K overflow

count entries)/s in addition to the input-port entries.

**d) Action-replay time:** 9 seconds.

2 entries are required per interrupt in addition to the input port entries.

Action-replay execution is still running 2.27 times slower than real-time execution (c/b).

## 5.6 M6800 Action-replay System Limitations

One of the Action-replay Debugging System design assumptions is that Action-replay verification and Program Status Information monitoring are performed via a Logic Analyser.

**Monitoring Process Limitations** : Experiments A and B described in the previous sections show that the M6800 implementation can cope with applications employing input data rates as high as 100 Kbyte/s and interrupt rates approaching 6KHz. These limits are basically imposed by the speed capabilities of the host computer and the size/speed of the FIFO buffer employed at the input of the host computer's input port. Therefore, these limitations are implementation dependent and can be cured by employing DMA techniques within the LSI11/23 computer so that the data entering the system is directly stored into the trace memory. This implies however that all monitoring functions must be performed within the LSI11/23-M6800 Interface (currently, some of these functions are implemented in LSI11/23 software).

**Action-replay Process Limitations:** Time measurements during both experiments A and B revealed that the Action-replay execution is 2.27 times slower than the original real-time execution. This delay is in fact introduced by the LSI11/23 responding to the interface data/interrupt requests. However, there is no reason whatsoever for not being able to achieve an Action-replay execution speed very close to that of the

original execution; any Action-replay functions implemented in LSI11/23 software (e.g., data decompression of the trace memory data) should be implemented in hardware.

## 5.7 Summary

An Action-replay Prototype System has been successfully developed, based on the "semi-resident approach" (# 4.2) where the target and host computers are the M6800 microcomputer and the LSI11/23 minicomputer respectively. The Host-MPU Interface complexity has been kept at minimum levels, but even so it was necessary to employ more than 150 SSI/MSI TTL integrated circuits in the design. Finally, the performance of the overall system was evaluated via two experiments, the first of which was focusing upon the performance of the "port input section" of the Interface while the second upon the performance of the "interrupt section" of the Interface.

# CHAPTER 6

## 6 ACTION-REPLAY TECHNIQUE EVALUATION

Many evaluations of program debugging methodologies have been undertaken in the past under controlled experiments [19]. However, due to the large number of factors involved, such as, the type of the errors introduced in the program, the experience of the programmers involved in the evaluation and the small sample of experiments carried out, these studies produced rather uncertain results.

Unfortunately, an equally large number of factors apply also in the evaluation of the Action-replay prototype system; especially since only parts of this system are actually available (# 5.0) and proper experimental evaluation is not possible. Even so, in an attempt to show that the debugging process can greatly benefit from the use of the Action-replay Debugging Technique, two hypothetical case-studies and one actual case study are employed here, all of which concern real-time malfunctions, that appeared during the hardware/software development which took place in this research work.

In particular, the malfunctions described in case studies I and II were chosen from real situations encountered early in the development of the prototype M6800 Action-replay System (a highly asynchronous system) and debugged with the help of a Logic Analyser; a lot of time and effort was required for each of them. During both these case studies it is assumed that an LSI11/23 Action-replay System is available and that this system has in fact all the properties listed in section #4.1.

The malfunction described in case study III was encountered later on within the M6800 system while performing Experiment B (# 5.5.2) and was debugged employing the prototype M6800 Action-replay Debugging System itself.

## 6.1 Case Study I

### The Malfunction

During the Action-replay phase the "expansion" of the "compressed" input port data/status information, residing in the Port Trace Memory, is performed via software techniques (# 5.4.2). In particular, the main routine is normally executing a "test reqa/b" loop and as soon as the Host-MPU Interface requests the next "port input" from the LSI11 by activating the DRV11 "reqb" signal, the "test reqa/b" loop is abandoned and the next "port-input entry" is fetched from the Port Trace Memory.

Let us assume that an "input-count entry" is found next, followed by an "overflow-count entry" (# 5.3.3). Then, the TR_OV subroutine is called and, as shown in Figure 10, the TR_DT subroutine is entered "overflow count" times with an "input count" equal to 2**15. After control has been returned to the main routine, the TR_DT subroutine is re-entered with an "input count" equal to that found in the "input-count entry"; this later action takes place normally when no "overflow-count entry" exists.

While in the TR_DT subroutine, and after a "port input" has been transmitted to the Host-MPU Interface (an action which also de-activates the "reqb" flip-flop), a secondary "test-reqb" loop is entered before the next transmission takes place as can be seen in Figure 10. However, after transmitting the "port input" for 2**15 times, and assuming that "overflow count" > 1, control is returned to the TR_OV subroutine without testing for "reqb" (point "2" at Figure 10); during the TR_DT subroutine development (and before the TR_OV development) it had been decided that it was not necessary to

employ a secondary "test reqb" loop at this point because control was always returned to the primary "test-reqb" loop in the main program routine. The result is that, every time the TR_DT subroutine is re-entered, two "port inputs" are sent to the Host-MPU Interface in the place of one and synchronisation between "port inputs" and Action-replay execution is eventually lost.

**Debugging Process Employed**

"Obviously what is needed is a secondary ´test-reqb´ loop either at point ´1´ shown on the TR_OV flowchart, or at point ´2´ shown on the TR_DT flowchart". However, when the above malfunction was encountered, the only available indication was that synchronisation is lost at the end of the Action-replay execution. Therefore, it is easy to realise that there were a number of alternative explanations. That is, not being able to establish a connection between the number of lost "port inputs" and the value of the "overflow count" it was not possible to even decide if this is a malfunction due to a software error or a malfunction due to a hardware error; after all, hardware and software development was undertaken concurrently. For example, because real-time is involved during the monitoring phase of the process, no method had been found for evaluating the performance of either the monitoring interface or the corresponding parts of the LSI11/23 software and, therefore, the information stored in the Port Trace Memory could not be trusted.

After a number of Monitoring and corresponding Action-replay trials it was decided that because a different number of "port inputs" was lost at the end of each trial the malfunction was intermittent. Therefore, having checked many times both the software and the hardware design of the system and not being able to pinpoint anything wrong, the so convenient explanation of "random noise at the hardware level" was given. Suspect wires were changed, "twisted pair" cables installed and the performance of integrated circuits was checked with the Logic Analyser.

The result of the above actions was negative and a time-consuming hardware/software debugging process was initiated by arranging so that "extreme values" were placed in the Port Trace Memory; not an easy process since real-time execution is involved while in the monitoring phase.

During one of the many tests it was arranged that only a single "overflow count" was monitored throughout the whole monitoring phase. At this point it was realised that, despite previous indications, the error was in fact repeatable if Action-replay is performed starting with the same information at the Port Trace Memory; the same final program state was always established. A few tests later on, the connection between the lost "port inputs" and the number of overflows was finally made; that is, the number of lost "port inputs" was always equal to [2*"overflow count"].

The malfunction could be within the TR_OV subroutine (the alternative being that the wrong information was placed in the Port Trace Memory). Therefore, the next step was to introduce a software breakpoint just before the "overflow count" test (point 4 at the flowchart) planning to single-step from there on for Program Status Information monitoring purposes. The result was that, after halting execution at the breakpoint, any indication of the malfunction in question disappeared for any "overflow count" value (even in test runs where single-stepping was not employed but uninterrupted execution was immediately commenced). Obviously, this was a real-time dependent error related to the Action-replay and not to the Monitoring part of the LSI11/23 program execution.

Not being able to collect any Program Status Information, it was necessary to employ "desk executions" (# 1.1.3) via which the design error was eventually discovered and finally corrected by introducing a "test reqb" loop at point "1" shown in the TR_OV subroutine flowchart (Figure 10).

**Discussion**

Let us consider why it was taken so long to discover the above malfunction (more than a week). One of the basic difficulties

encountered was the uncertainty over the validity of the Port Trace Memory contents; in fact, the correctness of the Monitoring Interface was proven later on after the development of the Action-replay Interface and during the evaluation experiments (# 5.3.3).

While taking the error as intermittent a very inefficient "trial and error" debugging process was employed, resulting in wrong assumptions (e.g., that of "random noise at the hardware level").

Having employed "extreme values" debugging, it was realised that the error was in fact repeatable and hence it was possible to organise a more systematic debugging process. However, because Program Status Information monitoring was prevented by the real-time nature of the error in question, time-consuming "desk executions" were employed in order to obtain the required Program Status Information.

Let us now assume that an "Action-replay Debugging System" is available for the LSI11/23 minicomputer.

As soon as it is realised that the synchronisation between the M6800 program Action-replay execution and the Port Trace Memory contents is lost the program is re-executed employing the Action-replay Debugging Technique the use of which implies the following basic advantages over the debugging process described above :

a) the Action-replay Debugging System can capture the failing program path and, consequently, repeat the error in question at will (this action is independent of the error type). Therefore, instead of wasting time over the "random noise" assumption and the "extreme values" debugging technique, it is now possible to organise a systematic debugging process immediately.

b) debugging is in fact undertaken within the non-real-time environment provided by the LSI11/23 Action-replay System. Therefore, not only the real-time execution of the

Monitoring part of the LSI11/23 program can be studied in non-real-time (thus establishing the validity of the Port Trace Memory information), but also it is not necessary to employ "desk executions" for Program Status Information monitoring purposes.

## 6.2 Case Study II

### The Malfunction

The 14-bit Code-fetch Counter output (path 1 at Figure 8) is directed via a 15-line flat cable to the Action-replay Interface (path 1 at Figure 9). This cable also holds the counter's overflow signal, which is activated as soon as the 2**14 count is reached, and an interrupt is immediately generated transferring control to the LSI11/23 interrupt routine that decrements the 14-bit "overflow-count entry" (# 5.3.3). Unfortunately line 14 on the cable and the "overflow" line (line 15) were accidentally short-circuited. The result was that the "overflow" interrupt was generated 2**13 code-fetch counts too soon and synchronisation was lost.

### Debugging Process Employed

The above hardware malfunction proved to be very difficult to detect because the indication that the synchronisation was lost was given long after the actual occurrence of the malfunction. In addition, the malfunction seemed to be "random" (different number of overflows in the trace-memory produced different erroneous results). Finally, the system has been designed in such a way that during the Action-replay the counter is automatically cleared after the generation of an M6800 interrupt making it impossible to check the exact number of code-fetch counts, which exist between two successive interrupts, against the corresponding code-fetch count located within the trace-memory.

During one of the many debugging strategies employed the counter was modified to be 13-bit wide (i.e., cable line 14

was not used) and it was then found that the malfunction did not occur any more. This discovery led eventually to the detection of the short-circuit.

**Discussion**

As with the previous case study one of the basic difficulties encountered during the debugging process was the uncertainty over the validity of the trace memory contents. An Action-replay Debugging System for the LSI11/23 minicomputer can greatly help with this problem as mentioned in section #6.1-b.

Having acquired a suspicion that the malfunction takes place along the code-fetch counter overflow process, it was decided to monitor this process employing Address-breakpoints; a very inefficient debugging process was then taken place since large variables and, correspondingly, many passes through the same code were involved. The need for the Action-replay Debugging system was strongly felt then; the "selective monitoring of Program Status Information" facility of the Action-replay Debugging System greatly improves debugging efficiency in such cases (# 3.4.1).

Finally, it was felt that the system under development had reached a high level of complexity and this created the additional problem of relating the large amount of monitored Program Status Information to the overall hardware/software system; the fact that the overall system status information was represented in three different number systems (binary for the hardware, octal for the LSI11/23 minicomputer and hexadecimal for the M6800 microcomputer) did not help either. This conclusion verifies the need for proper user interaction in debugging systems.

## 6.3 Case Study III

### The Malfunction

As mentioned in section #5.2.2, the PTR program was modified to accommodate two interrupt routines. These routines were placed in RAM just after the PTR program which was located in PROM. Having completed experiments A and B (# 5.5) it was decided to introduce an error into the target system and then to use the prototype M6800 Action-replay System to analyse the program response.

Hence, experiment B was performed as before but with the PTR speed increased beyond its permitted maximum value; that is, more than 150 characters per second. It was expected that the PTR program would cope with this situation. However, the result of the above action was disastrous; execution control was lost while at the same time the program section located in RAM was overwritten.

### Debugging Process Employed

The PTR program execution was monitored and then Action-replayed a number of times while the Logic Analyser was employed for Program Status Information monitoring during the Action-replay phase in conjunction with the M6800 Control Console which enables the user to access the target system memory (MPU register access is not provided). Thus, it was easily realised that the PTR system may respond to the above malfunction in one of the following four different ways.

1) because the paper tape is moving while the read operation takes place it is possible that a non-ASCII character is read in and, as expected, the program responds by sending an error message to the console's LEDs and by executing a Wait-For-Interrupt instruction (there is no HALT instruction available in the M6800 Assembly language).

2) a record of valid ASCII characters is read in but, because some of the characters are read by the PTR wrongly, the checksum test fails. As expected, an error message is

displayed at the console's LEDs and the M6800 MPU enters the WAIT state.

3) the hardware interface of the PTR fails to function properly. That is, the PIA status register shows that the PTR is busy while in fact the paper tape has been advanced to the next character; hence, a deadlock.

4) having entered the wait state (1 or 2 above) the MPU is forced by the next interrupt to execute the corresponding interrupt routine. This meant that as soon as the RTI instruction is executed at the end of the interrupt routine, the MPU continues execution with the code stored in the memory location following the WAIT instruction; that is, execution is not halted at the WAIT instruction as expected and control is lost.

Finally, one of the most experienced engineers in computer troubleshooting within the DSL, after having been told that the irrational program behaviour appeared as soon as the PTR speed had been increased, was invited to analyse the target system behaviour by employing conventional debugging tools (his written report may be found in Appendix B).

**Discussion**

The introduced malfunction (i.e., speeding up the PTR) had some surprising consequences which transformed this last experiment into a genuine debugging session. The prototype M6800 Action-replay System was found to be very efficient not only in easily confirming the two expected situations (1 and 2 above) (even the invalid ASCII characters responsible for the error message were monitored during Action-replaying) but also in detecting two more bugs, a hardware bug (3 above) and a software bug (4 above). Neither of these later bugs were expected since it was thought that the program would respond to the invalid PTR speed with 1 and 2 above; after all, the PTR loader, the NMI interrupt routine and the IRQ interrupt routine had been tested during the Experiment B (# 5.5.2) and were found to function properly. Obviously, the PTR program had to be modified further in order to cope with interrupts

properly.

All four situations described above were studied in less than one hour. However, the real potential of the Action-replay Debugging Technique was shown while studying the fourth situation; not only invalid ASCII characters were entering the system randomly (hence, the WAIT instruction was executed at different points along the program execution), but also the contents of the RAM locations following the WAIT instruction were not fixed to certain values. Hence, the user was presented with different symptoms during and at the end of the test executions since different program paths were traversed each time the program was executed. Therefore, it was very difficult to focus on a particular assumption of what might be wrong and to decide on a particular debugging strategy.

As can be seen in the written report at Appendix B, one of the first actions that the engineer wanted to take was to single-step through the PTR loader execution. To save time, it was then revealed to him that even if he had single-stepped through the complete program execution (and that would be many instructions indeed) the program would have been loaded properly; the M6800 does not respond to interrupts when in single-stepping mode. He then decided that a timing problem existed and, as he claims, if he did not know that the PTR speed had been increased to a non permitted range, he would have found about it via oscilloscope observations of the reader interface logic.

Again in order to save time an additional clue was given; that is, it was revealed that if an incorrect tape was used and the PTR speed was re-adjusted to its normal value (100 characters/s), the loader program would still fail and RAM would be overwritten; in effect, the purpose of this clue was to direct his attention to the error handling routines of the program. As can be seen from his report, he did continue with a static debugging session of the PTR loader program, but he failed to realise the connection between the WAIT instruction located at the end of a error handling routine and the random interrupts existing at the background.

Finally, he decided to continue with a dynamic debugging session employing the logic analyser for Program Status Information monitoring and execution tracing purposes. At this point in time it was mutually decided to stop the experiment, since this process could take a long time before obtaining proper results, and the interaction between the WAIT instruction and the interrupts was revealed to him.

It should be noted that, although there is no doubt that the malfunction would be discovered eventually, triggering the logic analyser at the proper instance in time could prove to be a problem especially since an error routine was randomly entered.

When the Action-replay System was used to analyse the malfunction in question, no data was entering the External-stimuli Trace Memory while the MPU is "WAITing" (even the Code-fetch Counter is idle).

This case study clearly shows the type of problems that a real-time malfunction can trigger and the difficulty involved in debugging such a malfunction by employing conventional tools; symptoms keep changing and it is difficult to focus on a particular fault.

Furthermore, this case study shows that the Action-replay Debugging Technique can indeed combat such real-time malfunctions successfully by allowing the user to concentrate not on how to reconstruct the malfunction (if it is at all possible), not on how to collect information about the program execution flow and its computational state, but on what type of information to collect during successive Action-replays.

## 6.4 Action-replay Debugging Technique Limitations

All three case studies given in the previous sections show the usefulness of the Action-replay Debugging Technique when real-time accessing of system-status information is required

during the debugging of intermittent/unrepeatable malfunctions. However, as shown below, there are some real-time situations which the Action-replay Debugging Technique cannot easily handle.

High input data rates (e.g., use of DMA within the target system), high interrupt rates and long monitoring periods (say, due to intermittent malfunctions of low frequency of occurrence) lead to a limitation related to the trace-memory size; that is, although it is now evident (# 5.5.1) that port input data can be compressed considerably in cases where the same data word is repeatedly entering the system, the trace memory may still overflow in a very short period of time (Experiment B, #5.5.2) in which case the monitoring process is aborted.

The trace memory overflow problem can be overcome by organising the host trace memory in such a way that its contents are continuously damped into floppy/hard disc storage via DMA techniques. However, it should be noted that in some cases the problem may persist because the required trace memory may exceed the amount of back up storage which can be available in practice. In cases like these the real-time execution and the corresponding external-stimuli monitoring must be re-initiated repeatedly until the malfunction occurs before the trace memory overflows; this technique was actually used during the experimental study III.

An alternative solution to the above problem, is to suspend real-time execution as soon as the trace memory overflows and to re-initiate the external-stimuli monitoring process only after an intermediate system snapshot is taken; this technique does assume that the target application specifications permit such interruptions of the program execution.

Another type of malfunction that can upset Action-replaying is that of an intermittent hardware malfunction within the target computer itself. Then, assuming that the Hardware Simulation approach is employed, such a malfunction either may occur during the original execution and not occur during the Action-

replay execution (or vice-versa), or may occur at different points along the Action-replay execution each time Action-replaying is re-initiated. In either of the above cases Action-replaying will fail. However, the Action-replay Verification mechanism will show the exact point on the program path where diversion occurs; according to the type of the malfunction such a hint may or may not help (conventional debugging tools must be employed in the latter case).

Finally, Action-replaying fails if the malfunction occurs within either the Host-MPU Interface, or within the host computer itself; conventional debugging techniques must be employed in discovering such a malfunction. However, since there is nothing worst in debugging than a tool which cannot be trusted, a test program should be available which must be used at the beginning of an Action-replay session to exercise the Action-replay Debugging System hardware (confidence testing).

## 6.5 Summary

Three case studies were undertaken in an attempt to evaluate the Action-replay Debugging Technique. These studies show that the Action-replay Debugging Technique can indeed aid real-time debugging of software (as well as hardware) malfunctions by providing a debugging environment which encourages the user to embark into a systematic and efficient debugging process; debugging the same malfunctions via conventional debugging methods would require a vast amount of trace memory for providing less efficient diagnostic facilities.

Finally, the basic limitations of the Action-replay Debugging Technique were outlined.

# CHAPTER 7

## 7 CONCLUSIONS

It is now evident that the Action-replay Debugging System can be successfully implemented in practice (Chapter 5) and that the Action-replay Debugging Technique eliminates the basic problem encountered in real-time debugging (Chapter 6); that is, it allows accessing of program status information without influencing in any way the timing of the target program/system. However, as was mentioned in Chapter 6, the Action-replay Debugging Technique does have certain limitations; recommendations for overcoming these limitations in future Action-replay Debugging System designs were also given.

Conclusions may now be drawn as to the generality of the Action-replay Debugging Technique and recommendations can be given related to possible Action-replay Debugging System implementation strategies.

### 7.1 Implementation Dependent Features

It is now possible to outline the Action-replay Debugging System features which are target MPU dependent; that is, the sections of the Host-MPU Interface which are dedicated to the target microcomputer system architecture (it is assumed that interfacing at the bus level of a target MPU is possible).

### 7.1.1 Monitoring Phase

During the monitoring phase (and the Action-replaying phase in the case where the hardware simulator approach has been chosen) the following three units of the Host-MPU Interface are target-MPU dependent :

1) Timing-unit Sensor; that is, decoding either a code-fetch or an instruction-fetch cycle.

2) Port Input-cycle Sensor; that is, decoding that the current machine cycle addresses an input port.

3) Interrupt Acknowledge Sensor; that is, decoding that an interrupt has been generated and that the MPU has already responded to it.

Having considered a number of microcomputers it is clear that differences exist in all three of the above areas. However, as can be seen below, these differences can easily be taken into account in a Action-replay System which incorporates microprogrammable control logic :

1) most microcomputers provide either status information which indicates the type of the current machine cycle (e.g., M6809, M68000, INTEL8080/85, Z8000), or a dedicated signal which indicates an opcode-fetch cycle (e.g., TMS9900, F100-L). Hence, it is almost always possible to sense an instruction-fetch cycle; in cases where this is not possible (e.g., M6800) the code-fetch technique can be used instead (# 5.4.1).

2) port input-cycles can be easily decoded either via the state information mentioned above or, in the case where "memory-mapped input/output" is employed, by decoding the memory address space dedicated to input/output activity while a "read" operation takes place (as with the M6800 implementation). This later case can also be employed for overcoming the DMA problem as was mentioned in section #3.1.3; that is, it can be safely assumed that the DMA buffer is part of the "memory-mapped input/output" address

space and that the memory read-cycles, which address this space, can be taken as port input-cycles.
It should be noted that special consideration must be given to MPUs which have input ports on-chip; that is, ports connected to the environment bypassing the MPU buses (e.g., the SID input of the INTEL8085).

3) some microcomputers provide "interrupt acknowledge" signals (e.g., F100-L). If however such a signal is not provided, it can be generated easily either by decoding the status information provided during "interrupt acknowledge" machine-cycles, or by decoding the appearance of an interrupt-vector address on the MPU address bus during a memory-read cycle (as with the M6800 implementation) and assuming that the user program may only write-into and not read-from such memory location.

### 7.1.2 Action-replay Phase

**Hardware Simulator Approach**

In the case of the hardware simulator approach to Action-replaying (# 3.2.1) it must be possible to :

1) disable all input ports of the system and all interrupt lines entering the system; these in general are easily implemented target-system modifications.

2) to interrupt program execution for Program Status Information monitoring purposes. Most microcomputers can be halted via an external signal (e.g., WAIT for INTEL MPUs and HALT for Motorola MPUs). However, care must be taken to avoid strange interactions between the "halt sequence" and the "interrupt sequence" of the MPU, in which case external hardware techniques must be employed to achieve the required suspension of the Action-replay execution (e.g., the "branch-to-itself loop" technique incorporated in the M6800 design (# 5.4.2)).

3) to generate a dummy interrupt in order to permit MPU

register accessing; the MPU registers are forced out of the MPU chip and into the microcomputer memory either by the interrupt-acknowledge sequence itself, or by the interrupt routine which follows. Although this process seems to be easily implemented it requires some complex hardware since a transparent implementation is needed. For example, in the case where a small interrupt routine is incorporated within the microcomputer during the Action-replaying phase for Program Status Information monitoring purposes, the execution of this routine must be transparent to the Action-replay execution of the target program (and in general to the Action-replay part of the Host-MPU Interface) otherwise synchronisation between the trace memory contents and the target program execution is lost and Action-replaying will fail.

**Software Simulator Approach**

The software simulator approach to Action-replaying * is much more flexible than the hardware simulator approach as far as execution control and Program Status Information monitoring are concerned (e.g., the generation of the dummy interrupt is not required in order to access the MPU registers). However, the software simulator's speed becomes a serious limitation (over 1000 times slower than the real-time execution).

* The software simulator approach is actually chosen in an INTEL8085 based Action-replay System implementation [18,29] which currently takes place at the "Microelectronics Applications Research Institute" following an investigation of existing real-time debugging techniques/tools [10].

## 7.2 Alternative Implementation

Flexibility and speed can be achieved during the Action-replay phase by combining together the Software and the Hardware Simulator approaches. Such an alternative system functions as follows :

> the Action-replay process takes place initially within the target MPU (when Program Status Information monitoring is not required) and up to an execution point where a simple breakpoint mechanism interrupts this process and transfers execution control to the software simulator where Action-replay progresses for Program Status Information monitoring purposes.
>
> As soon as Action-replay within the software simulator progresses beyond the specified bounds (i.e., within code areas where Program Status Information monitoring is not required) control is transferred back to the hardware simulator (i.e., the MPU) until further "zooming-in" is required for Program Status Information monitoring purposes.

Address breakpoints seem to be appropriate for the above mentioned breakpoint mechanism. In particular, they must be implemented via hardware techniques since the target program timing must not be altered during Action-replaying (e.g., the breakpoint table can be implemented via associative memory techniques). Then, the debugging system enters into the breakpoint table a number of address pairs, each specifying the lower and upper limits of the section of code during the execution of which Program Status Information monitoring is to take place. These limits may either be explicitly specified by the user in terms of source file line numbers, which can then be translated by the debugging system into addresses and loaded into the "breakpoint table", or can be automatically generated by the debugging system taking into account the operands employed in the user specified monitoring conditions (# 3.4.1).

Each time the Action-replay control is transferred from the

target MPU to its simulator, or vice-versa, the current state of the execution **(computation snapshot)** must be transferred also so that one may continue executing where the other has left off. Depending on the application, the size of the computation snapshot can be rather large making the simulation technique in question inefficient. Obviously, the ideal solution is to keep all this information at one place which is then accessed by either the MPU or the host computer where the software simulator resides. The microcomputer memory itself seems to be such a place since it is there that most of this information initially resides; the exception being the MPU registers which may end up in the system stack by generating a dummy interrupt; this interrupt may be generated as soon as an address breakpoint is "hit".

The above mentioned ideas imply that the host computer must be supplied with a mechanism of accessing the target system memory (DMA Interface at Figure 4). This can only be done via some additional but minimal hardware for generating the required control signals and, subsequently, disabling the MPU chip and acquiring full control of its buses [28]. Then, only two actions have to be undertaken each time an interrupt is generated by the breakpoint mechanism and Action-replay control is passed between the target MPU and its simulator :

**1)** the MPU is disabled and the simulator in the host computer is enabled (or vice-versa) and

**2)** the MPU registers are forced into either the MPU chip or its simulator (depending on which of the two was just enabled).

In MPU architectures where the "interrupt-acknowledge" sequence (and consequently the "return-from-interrupt" sequence) do not force the MPU registers into the memory stack (or back into the MPU chip) explicit means must be employed in order to acquire access to those registers as was mentioned earlier on.

## 7.3 Action-replay Debugging Process Upgrading

Instead of developing a new debugging system which is based on the Action-replay Debugging Technique the Action-replay facility may be provided as an autonomous part of an already existing development system. Then, according to user requirements, the Action-replay facility may either be invoked or bypassed. Whatever implementation strategy is chosen in future Action-replay Debugging System designs, software techniques must be employed in order to upgrade the debugging process. For example, since interactive software debugging is underway, and since debugging in general is usually associated with a large number of small changes, rapid turnaround time on program changes would increase debugging efficiency.

"Program-structure" information (functional and performance requirements of individual modules, definitions of interfaces between them, etc.), produced during early development phases, becomes "obvious" to the programmer as the development process progresses and therefore is all-to-often preserved only by memory. Later on, when the debugging stage is reached, such information has either been forgotten or remembered only vaguely and therefore program-status information is wrongly evaluated as the result of which the wrong actions are taken leading to inefficient debugging. Forming a "software description" file during the program design stage of the development could be rather advantageous later on in the debugging stage. Similarly a "hardware description" file should be employed.

Future developments of an Action-replay Debugging System should employ software techniques in order to upgrade the Action-replay Debugging process to the source language level [41]; that is, during debugging the programmer may refer to variables in the program symbolically.

However, symbolic referencing can be used only with those language processors which generate a symbol table with the object programs they produce. Hence, compilers which do not make available their symbol tables will have to be modified to do so. Special consideration should be given however to stack

orientated languages, such as ALGOL, since the absolute addresses of variables are known only after the corresponding declaration statement has been executed; the same declaration statement may even result in different absolute memory address for the same variable name according to where along the program it is executed (e.g., when the statement in question is within a subroutine and recursion is employed).

For Program Status Information monitoring purposes the Action-replay system must be able to set address breakpoints. Since source language debugging is assumed, the locations within the executable code (e.g., the entry point to a subroutine) must be specified symbolically (i.e., in terms of source file line numbers, statement numbers/labels, etc.). However most compilers generate relocatable code the absolute addresses of which is generated by a linker/loader according to certain relocation factors and Only at this stage the absolute addresses of breakpoints are known and can be found in "load maps" in order to be entered into the breakpoint table. Special consideration should also be drawn to compilers which perform code optimisation (e.g., the order of execution could be altered, invariant code could be moved from loops, etc).

It should be noted that most operating systems and high-level languages have their own debugging systems (e.g., "Symbolic Debugging System" in MTS [53]). Such systems can probably be used in conjunction with the Action-replay Debugging Technique assuming that there is a way of distinguishing between target program execution and execution of statements inserted between the source code by the compiler/debugger for target program execution control purposes. The above distinction is required so that the Action-replay "Time" is stopped when these statements are executed otherwise target program timing is altered and Action-replay synchronisation is lost.

Having established the Action-replay debugging technique at the source language level, there is no reason whatsoever for not being able to raise the debugging level even more and, consequently, improve the debugging efficiency even further.

For example, more sophisticated formating routines can relate the Program Status Information directly to the source file itself without any extra confusing information such as statement numbers. For example, as an instruction trace record is displayed on the left of the VDU screen, a trace of one of the program variables is displayed on the right of the screen, while the corresponding source-file assignment statement, responsible for each of the variable changes, is automatically underlined on the left of the screen. In addition, program execution statistical data can be generated and displayed at the VDU screen.

This research study has shown that the Action-replay Debugging System can indeed be implemented in practice and that the Action-replay Debugging Technique aids the detection of a large class of previously difficult intermittent/unrepeatable malfunctions by providing a debugging environment which keeps the erroneous program behaviour in a stable condition so that particular symptoms relating to particular faults can be focussed upon and analysed. This latter property of the Action-replay Debugging Technique minimises the chances of taking the wrong action along the debugging process (since the user is not confused by rapidly changing symptoms) and, thus, a systematic and efficient debugging process is achieved.

# FIGURES

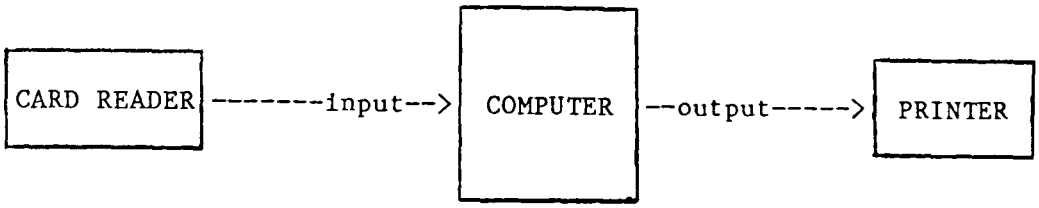


**FIGURE 1** : Conventional Computer System.

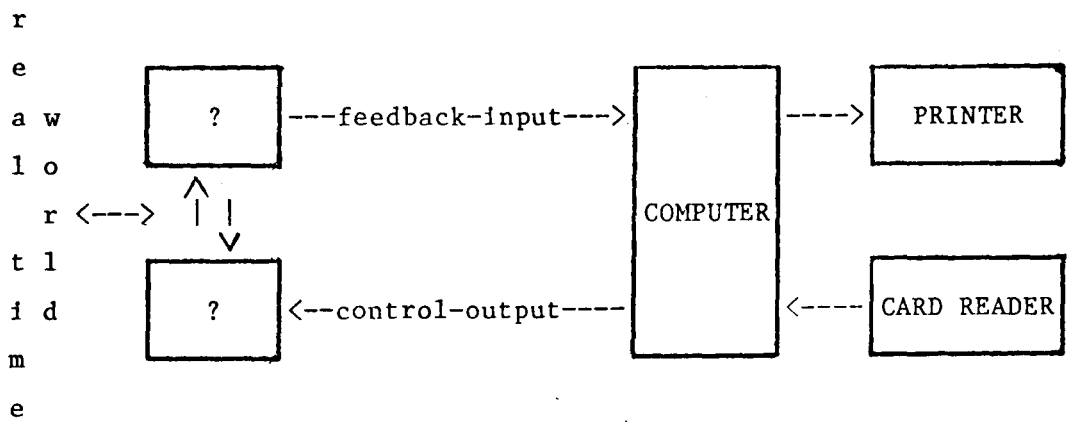


**FIGURE 2** : Real-time Computer System.

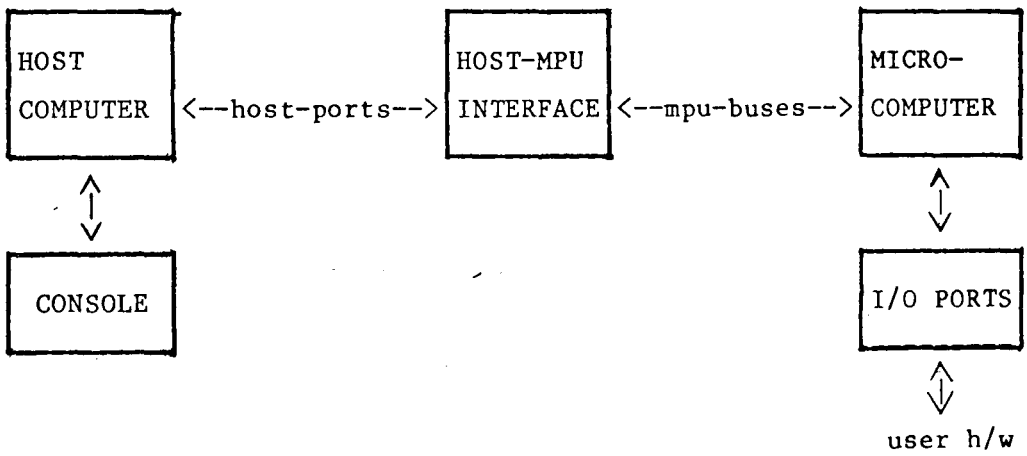


**FIGURE 3 :** "Action-replay System" Block Diagram.

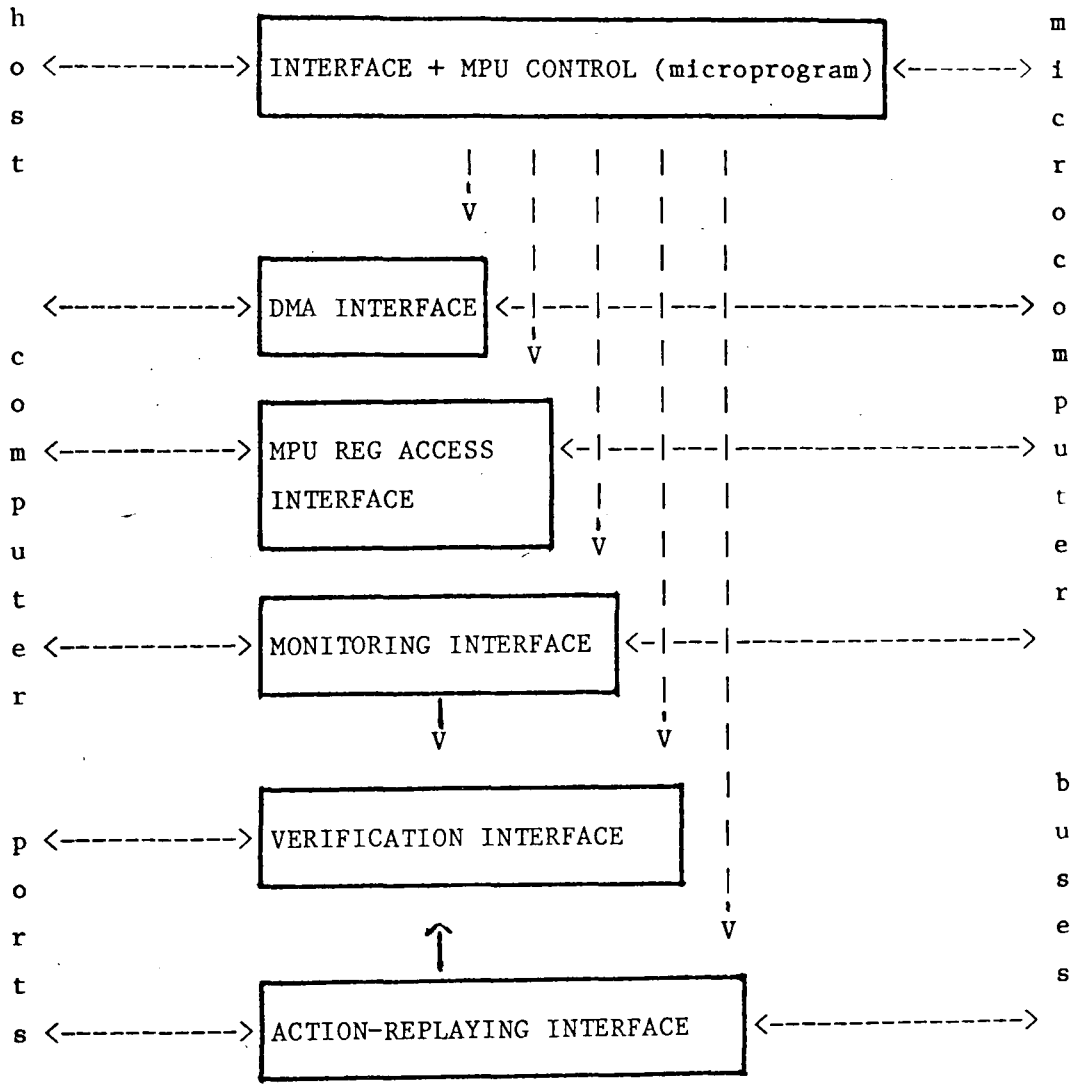


**FIGURE 4 :** "Host-MPU Interface" Block Diagram.

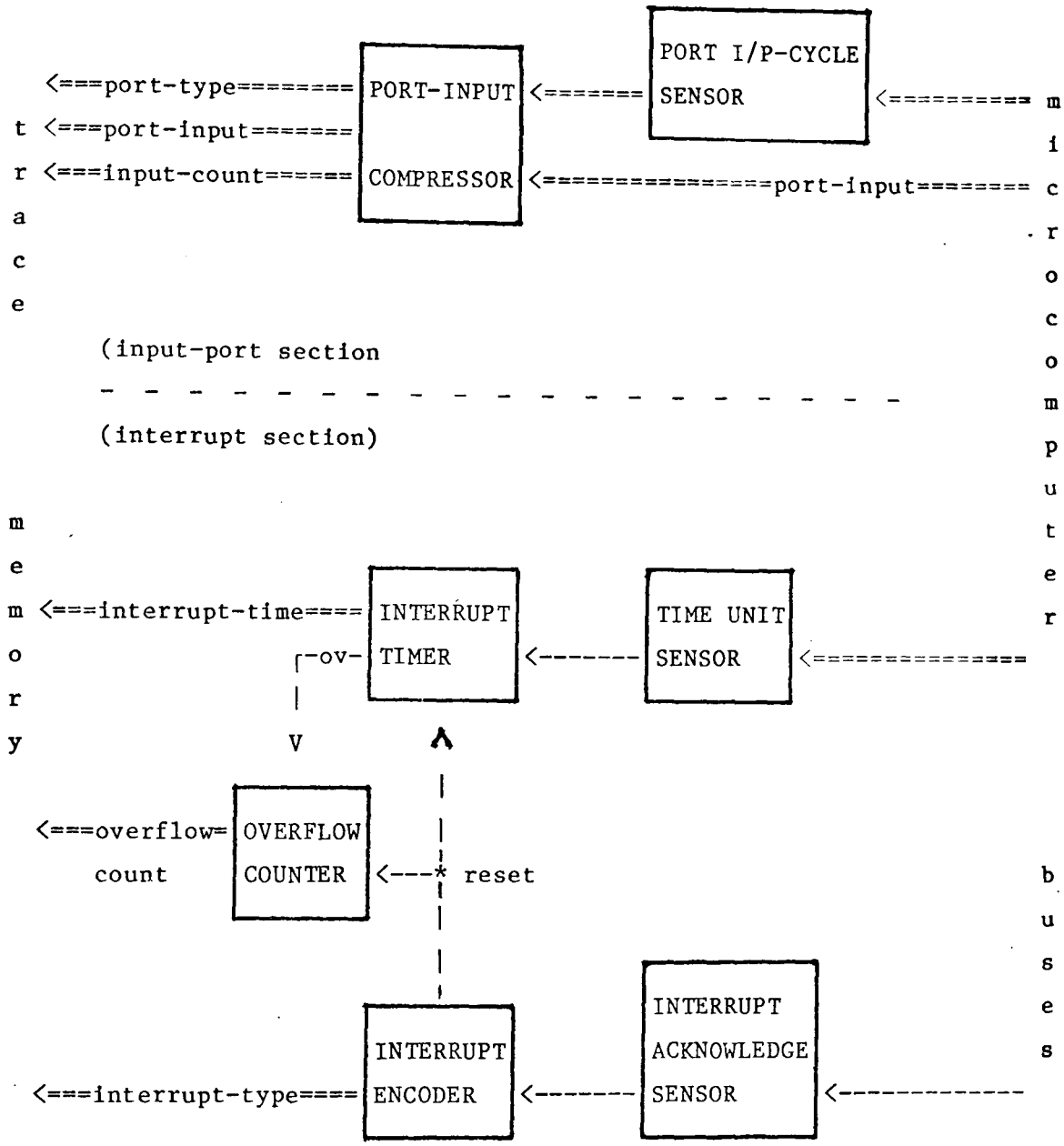


**FIGURE 5** : "Monitoring Interface" Organisation.

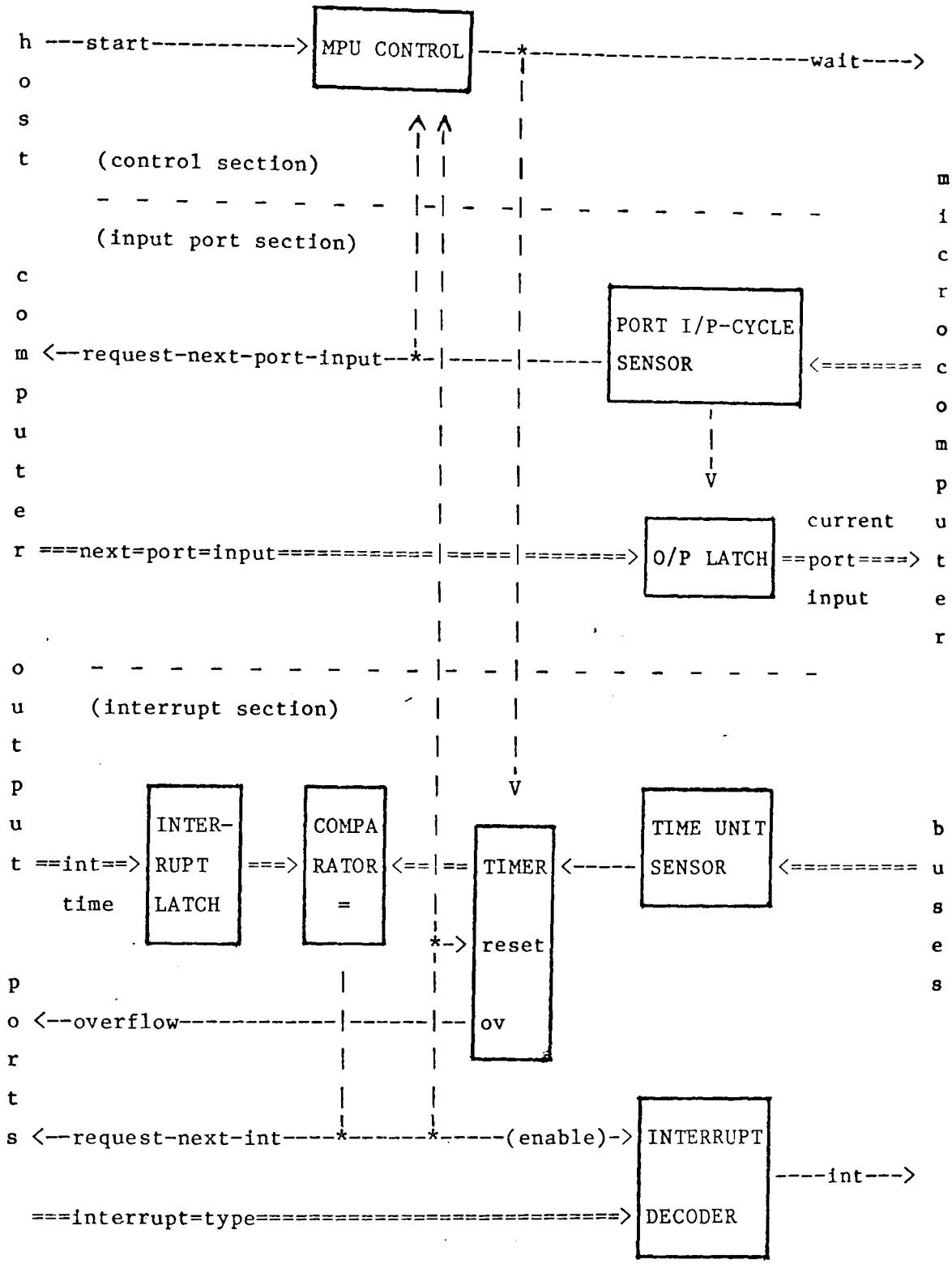


**FIGURE 6 :** "Action-replaying Interface" Organisation.

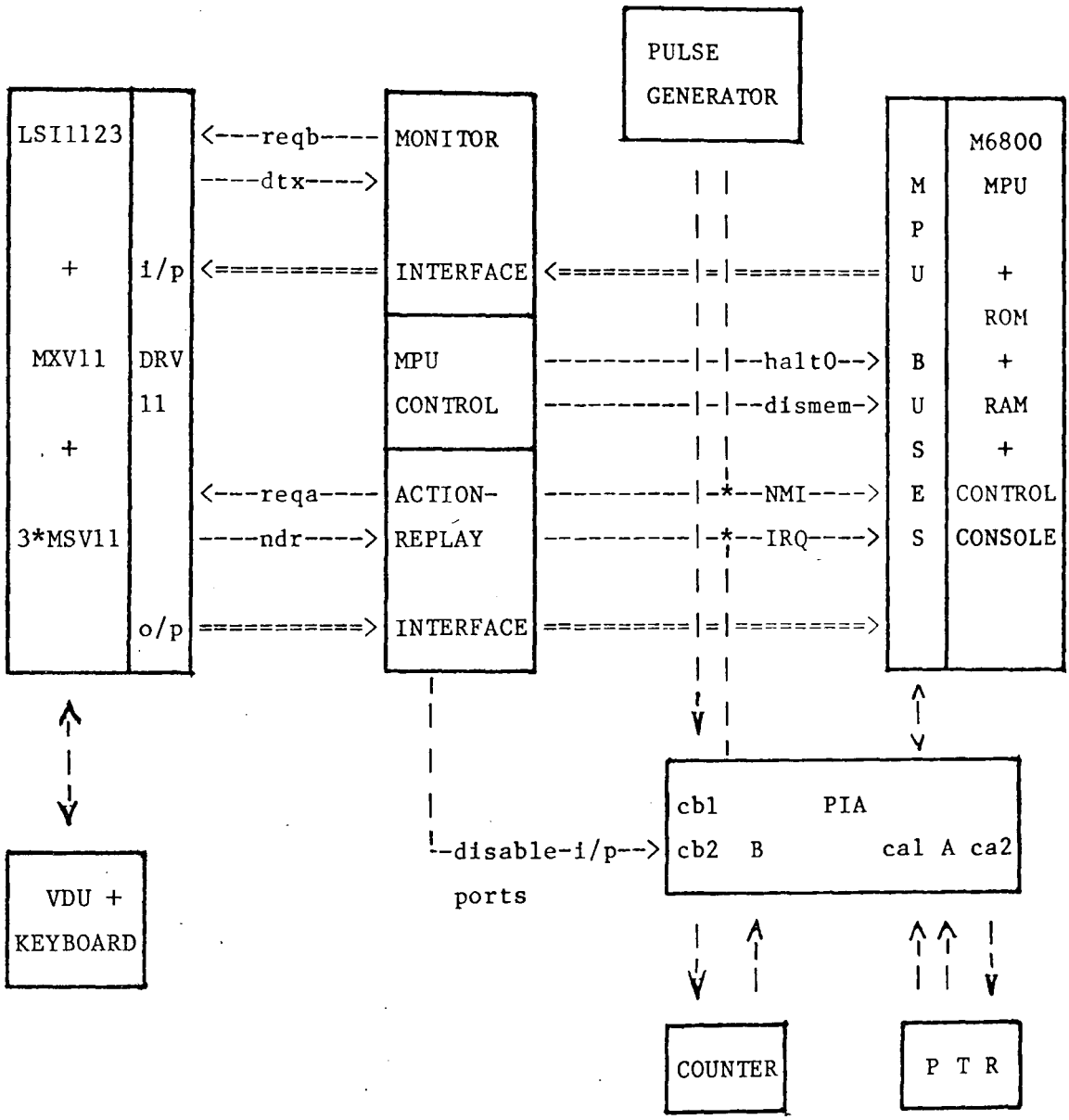


**FIGURE 7** : Action-replay System Implementation.

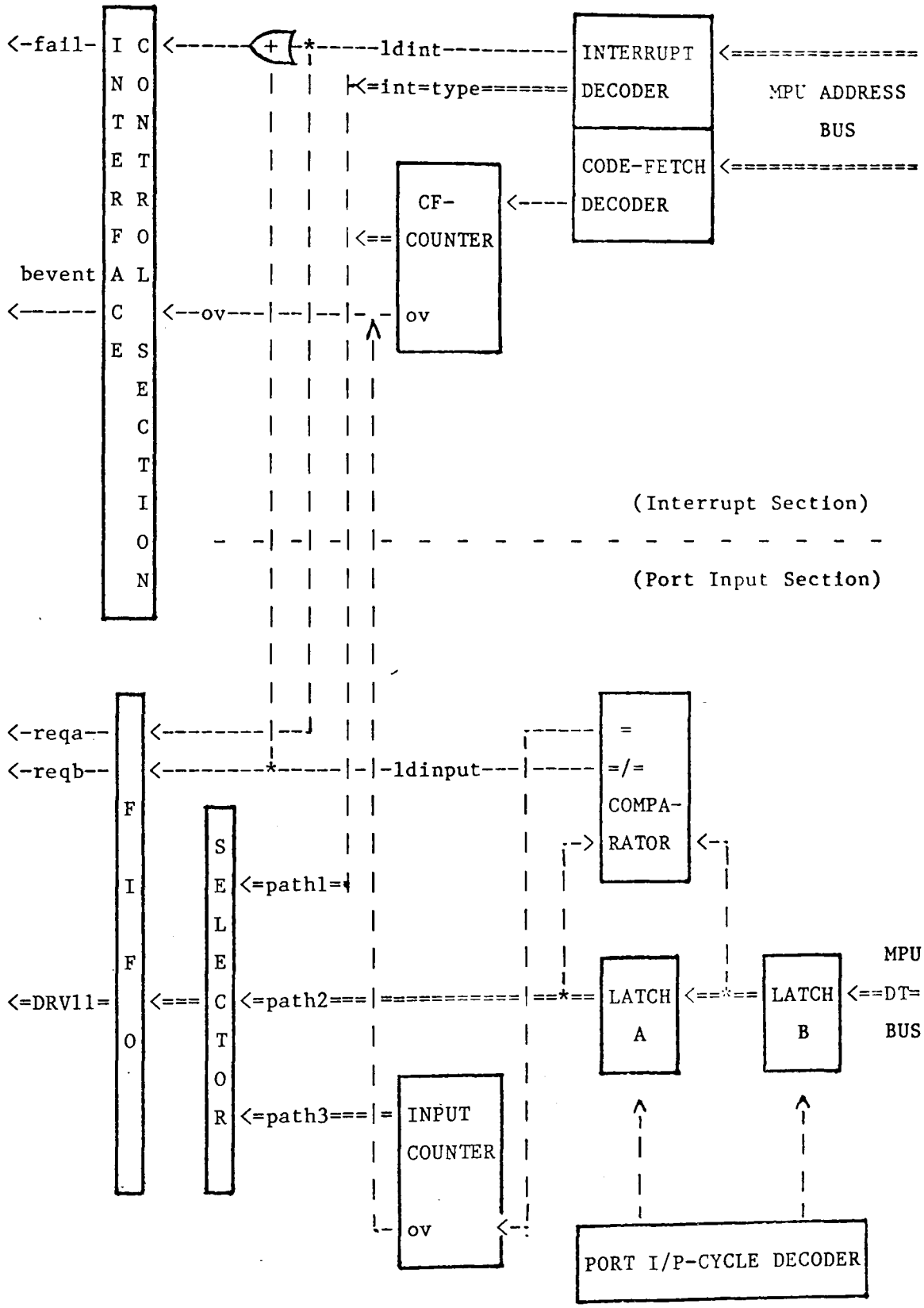


**FIGURE 8** : Monitoring Interface (located at DSL board A).

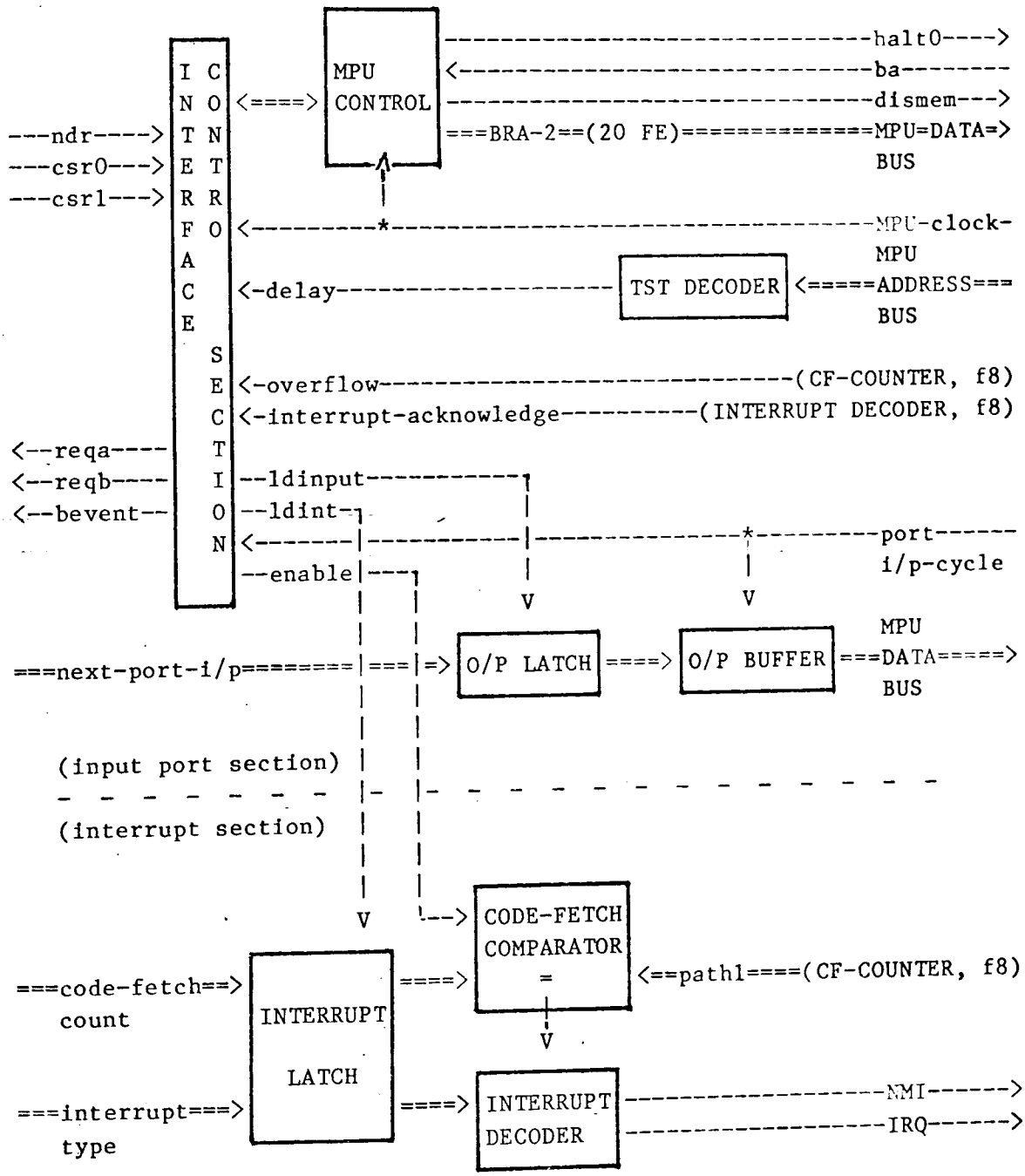


**FIGURE 9** : Action-replaying Interface (located at DSL board B).

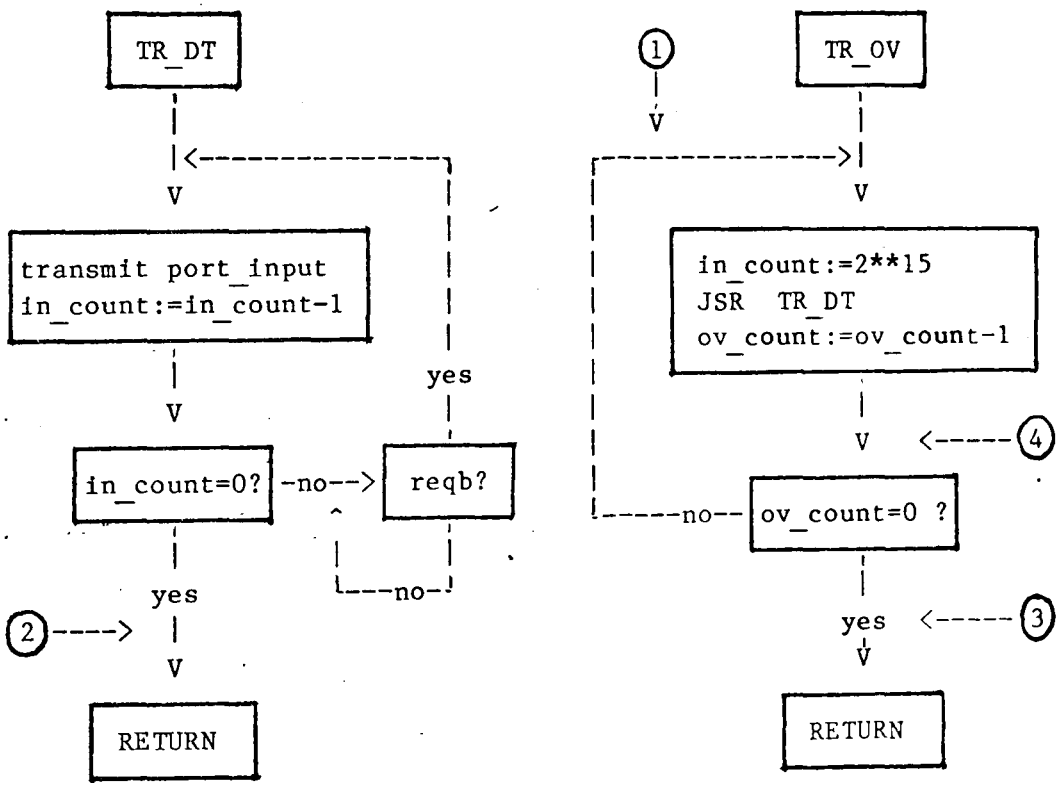


**Figure 10 :** Flowcharts for the TR_DT and TR_OV subroutines.

## BIBLIOGRAPHY


[1] Aspinall, D. The Microprocessor and its Application. Cambridge Univ. Press 1978.

[2] Balzer, R.M. EXDAMS : EXtendable Debugging And Monitoring System. Proc. AFIPS 1969, vol. 34.

[3] Bass C. Software Development Strategy for Microcomputers. ZILOG System Software Dept., Tech. R. no 2, July 1977.

[4] Boyd, D.L. and Pizzarello, A. Introduction to the WELLMADE Design Methodology. IEEE Trans. Software Eng., July 1978.

[5] Brown, A.R. and Sampson, W.R. "Program Debugging", Macdonald, 1973.

[6] CAP MicroSoft LTD, MicroAde, 1977.

[7] Dasai, T. et al. High Level Process Control Language "ESPRINT" and its Source Level Debugging System "SOLDA". Real Time Programming 1977, IFAC 1977.

[8] Dijkstra, E.W. Notes on Structured Programming. "Structured Programming", Academic Press 1972.

[9] Dobson, J.E. The Action Replay Software Mechanism. MARI working paper, ref. A029/2.6, October 1980.

[10] Dobson, J.E. and Ghani, N. Real-time Microprocessor Development Techniques. MARI ref. A029/3.2, 1981.

[11] Fagan, M.E. Design and Code Inspections to Reduce Errors in Program Development. IBM Syst. J., vol. 14-15, 1976.

[12] Fairley, R.E. ALADDIN : Assembly Language Assertion Driven Debugging Interpreter. IEEE Trans. Software Eng., vol SE-5, July 1979.

[13] Farrell, E. and Kanellopoulos, N.G.K. Debugging Aids for

Microprocessor Systems. Microprocessors, April 1978.

[14] Francis, R. and Teitzel, R. Real Time Prototype Analysis as a Microprocessor Design Aid. Computer Design, vol. 17/12, December 1978.

[15] Gains, R.S. The Debugging of Computer Programs. Ph.D. thesis, Princeton Univ. 1969.

[16] Ghani, N. and Farrell, E. Microprocessor System Debugging. Research Studies Press, 1980.

[17] Ghani, N. and Givens, J.G. A Teaching Laboratory for Digital Systems. University of Newcastle Upon Tyne, Tech.Rep. no 109, 1977.

[18] Ghani, N. Action Replay 8085 : Initial Hardware Design Considerations. MARI working paper, ref. A029/2.13, April 1981.

[19] Gould, J.D. and Drongowski, P. An Exploratory Study of Computer Debugging. Human Factors 16,3, 1974.

[20] Grishman, R. The Debugging System AIDS. SJCC, pp 59-64, 1970.

[21] Groves, L.J. The Provision of Debugging Facilities for High Level Languages. Massey University Computer Unit, report no. 18, May 1975.

[22] Henderson, P. An Experiment in Structured Programming. BIT 12, 1972.

[23] Henderson, P. The TOPD System. Computing Lab. Tech. Rep. no 77, University of Newcastle Upon Tyne, September 1977.

[24] Hennessy, J. Symbolic Debugging of Optimised Code. Ph.D. Thesis, Computer Systems Laboratory, Stanford University.

[25] Hewlett Packard. The 64000 Logic Development System, 1979.

[26] Howden, W.E. Theoretical and Empirical Studies of Program Testing. IEEE Trans. Software Eng., vol. SE-5, July 1978.

[27] INTEL Corp., MLS-80 User's Manual, October 1977.

[28] Kanellopoulos, N.G.K. An Investigation into Hardware, Firmware and Software Techniques for Providing a Generalised Console for a Microcomputer System. M.Sc. Thesis, University of Newcastle Upon Tyne, 1978.

[29] Kanellopoulos, N.G.K., et al. Apparatus for Assisting Fault-finding in Data Processing Systems. Pending patent specification, application number : 8110676-6/April/1981.

[30] Kline, B. et al. The In-circuit Approach to the Development of Microcomputer-based Products. Proceedings of the IEEE, vol. 64, June 1976.

[31] Lauesen, S. Debugging Techniques. Software Practice and Experience. vol. 9, 1979.

[32] Ledgard, H.F. The Case for Structured Programming. BIT 14, 1974.

[33] McCracken, D. Hybrid Tool for Universal Microprocessor Development. Computer Design, April 1980.

[34] Millennium Systems. Microsystems Emulator Manual, 1980.

[35] Model, M.L. Monitoring System Behaviour in a Complex Computational Environment. XEROX, CSL-79-1, Jan. 1979.

[36] Motorola Inc. M6800 Microcomputer System Design Data, 1976.

[37] Motorola Inc. M6800 Microcomputer Programming Manual, 1975.

[38] Mullin, F.J. Considerations for a Successful Software Test Program. TRW Software Series, Jan. 1977.

[39] Myers, B.A. Displaying Data Structures for Interactive

Debugging. XEROX, CSL-80-7, June 1980.

[40] Myers, G.J. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. CACM, vol. 21, September 1978.

[41] Pierce, R.H. Source Language Debugging on a Small Computer. Comp.J. vol 17, no4, 1974.

[42] Santoni, A. Microprocessor Testers Survey. Electronics, December 1976.

[43] Satterthwaite, E. Debugging Tools For High Level Languages, University of Newcastle Upon Tyne, Tech.Rep. no 29, December 1971.

[44] Systems Designers LTD. The CONTEXT microprocessor development system, 1979.

[45] Tektronix. The 8002/8001 Microprocessor Labs, 1977.

[46] Texas Instruments. AMPL Microprocessor Prototyping Lab. TI Computer News.

[47] Van Leer P. Top-down Development Using a Program Design Language. IBM Syst. J. vol. 14-15, 1976.

[48] Wilkes, M.V. Software Engineering and Structured Programming. IEEE Trans. Software Eng., vol. SE-2, December 1976.

[49] Wirth, N. Program Development by Stepwise Refinement. CACM,vol. 14, April 1971.

[50] Yannacopoulos, N.A. et al. Performance Measurements of the MU5 Primary Instruction Pipeline. IFIP, 1977.

[51] Yeh, R.T. et al. Current Trends in Programming Methodology, vol. 2, Prentice Hall 1977.

[52] DEC, Microcomputer Processor Handbook, 1979.

[53] University of Michigan. The Michigan Terminal System, vol.1, December 1979.

# A P P E N D I C E S

# A P P E N D I X A

## "Monitoring Program Paths"

The obvious way of recording a program path is to record only the addresses corresponding to instruction opcodes. This, however, not only is a processor dependent process, since the "opcode fetch" machine-cycle must be identified first, but generates a large amount of trace data (150kbytes/s on average for an M6800 MPU driven at 1MHz clock rate).

An alternative method is to record only program-nodes; these are addresses shown on the address bus just before a conditional/unconditional execution-control transfer. Program-node identification is not difficult and, furthermore, it is processor independent. Assembler generated data can be analysed and the address boundaries of those memory blocks which hold program code and those which hold program data can be derived. A comparison at the hardware level indicates whether the current contents of the address bus are Program-Addresses (PA) or Data-Addresses (DA). Then,

if **PAn =/= PA(n-1)+1** then **PA(n-1) = program node.**

A statistical study of one of the DSL M6800 programs (9kbytes of code) concluded that on average there are 2.43 bytes per instruction (422 instructions/kbyte), 4 machine cycles per instruction, 600kbytes of executable code per second and 137 potential control transfers per kbytes (1 every 3 instructions). The number of successful control transfers is obviously less when measured dynamically. A performance evaluation study [50] concluded that a typical control transfer rate

in most Von Neumann structured processors is 1 in every 7 to 10 instructions. Assuming a similar case for the M6800 machine (say 1 control transfer every 8 instructions), the average number of nodes per kbyte of executable code is 52 (that is, 422 instructions/kbyte divided by 8 instructions/control_transfer). Therefore, the average amount of traced information is 60kbytes/s (that is, 600kbytes/s * 52nodes/kbyte * 2bytes) which can be minimised even further via the following two techniques :

**1)** if the maximum number of different nodes permitted within a program module is 256 (i.e., the maximum module size to be debugged is 256*8=2048 instructions which corresponds to 4.86kbytes of program code), each node address can be encoded into an 8-bit number. This information is then stored in the trace memory instead of the corresponding 16-bit address. The average trace information per second is therefore brought down to 30kbytes, which corresponds to an execution of 600kbytes program code.

**2)** further minimisation may be achieved by compressing repeated information which arises, for example, when executing a program loop (or even nested loops). An identical technique to that employed to input port data reduction (# 3.1.1) seems to be appropriate here.

## A P P E N D I X B

### "Notes on discussions with Nikos concerning the debugging of the M6800 paper-tape-reader (PTR) "fault"

**Presented problem** : While reading paper tape and with an interrupt source whose period between interrupts was controlled by a pseudo-random-number, the tape was not read properly. Sometimes the tape would be partially read before the processor displayed a "checksum" error or a "non-ASCII" error. Sometimes the tape operation would stop, sometimes would not stop and usually RAM memory would be corrupted. It was also revealed that in fact the "busy" period of the PTR had been shortened with the result that the processor read data before it was valid. I was supposed to analyse the above program behaviour.

**Debugging Process Employed** : There was one fairly obvious clue in the way the paper tape jerked through the reader instead of running through smoothly; but this might have been a function of the random period interrupt routine. It was conceded (rather than observed) that single stepping through the reader routine would have resulted in tape being correctly read. Assuming that I did not Know that the reader speed had been increased beyond the maximum permitted limit, this second clue would have pointed to a timing problem, and an oscilloscope observation of the reader interface logic, when matched against the reader specifications, would have revealed the "busy" period problem. Time might have been spent in first verifying the mechanical adjustment of the reader. In practice, most time would probably have been spent in locating the

reader specifications and then reading and understanding them.

Then, it was revealed that, even if the timing correction had been made, a second-order effect would still remain; it was stated that good tapes (read successfully before) could be loaded correctly, but if the odd hole was obscured, again RAM could be overwritten.

My approach was to go to the reader program to try to understand what the program did in the event of misreading. This revealed nothing that seemed very probable; for instance, in the special case of the byte count being obscured, and hence misread, it was conceivable that up to 50 inches of tape could be read, but since some of these rows should have contained non-ASCII characters, the loader program should have stopped early.

The problem was then revealed to me to be the result of the background interrupt routine releasing the reader program from the "WAIT" state.

The next approach (not carried out) would have been to use a logic analyser to step through chunks of program. This could be expected to reveal (from the MAP display) that the program did not stop when an error was found, as it was expected to do. This in turn would have pointed back at the program and would have led to deeper investigations of the operations of particular instructions (setting of condition codes, etc.). The secondary problem would thus almost certainly have been solved with a combination of tracing and an understanding of the program.

It is extremely difficult to supply times for phases of debugging since these must depend on such factors as familiarity with equipment (e.g., logic analyser), programs and instruction sets, and on interruptions at critical times. My estimate would range from 2 hours to a full day.

Keith Heron ._