

https://dspace.org/

Debugging techniques for real-time and process-control software

Helps, Cedric Richard Gordon

2015-01-26
http://hdl.handle.net/10539/16754

Downloaded from a DSpace repository. https://dspace.org/

DEBUGGING TECHNIQUES FOR REAL-TIME AND PROCESS-CONTROL SOFTWARE

C. Richard G. Helps

A project report submitted to the University of the Witwatersrand, Johannesburg, in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

Johannesburg 1985

Declaration

I declare that this project report is my own unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

ABSTRACT

This investigation was carried out to determine the state of the art in debugging techniques for real-time and process control software. Debugging tools for real-time software tend to be primitive. The data processing field is much more sophisticated. Debugging tools are difficult to create due to the unique aspects of real-time and process control software. The newer real-time systems becoming available provide opportunities for improved debugging tools. Existing debugging methods and tools are described and a conceptual design of a real-time debugging system is proposed.

Acknowledgements

Thanks to my wife Louise for her continual sustaining and encouragement. I would also like to acknowledge the assistance of many of my colleagues in providing advice and material for this report.

CONTENTS

1. INTRODUCTION

2. THE STATE OF THE ART (Literature Survey)
2.1 Problem Identification
2.2 Real-time environments and languages
2.3 General Testing Debugging and Verification Techniques
2.4 Real-time testing and debugging
2.5 Debugging concepts and ideas
2.6 Summary

3. REAL-TIME LANGUAGES AND ENVIRONMENTS

4. DEBUGGING METHODS

5. DEBUGGING TOOLS
5.1 A Selection of tools
5.2 Extensibility
5.3 Patching of software
5.4 DOS Debugging Tools
5.5 A Programmer's Workbench

6. SPECIFICATION OF A PROPOSED DEBUGGER
6.1 General
6.2 Overview
6.3 The Controller
6.4 Debugger Hardware
6.5 Debugger Functions (Software)
6.6 Summary

7. CONCLUSION

APPENDICES

A. REFERENCES

1.0 INTRODUCTION

The use of computers for data-processing (dp) is now a mature industry and has developed a strong organisational hierarchy and a sophisticated approach to program development and debugging. In contrast to this the real-time and, in particular, the process control software industries are still developing rapidly. The debugging tools available in these latter areas tend to be primitive and to vary widely from one development system to another.

High level languages such as Fortran or PL/1 do not provide structures or functions for handling real-time responses. They are run as batch jobs whose timing is controlled by cpu load-scheduling priorities rather than by any real world synchronisation requirements. Furthermore traditional high level languages do not provide ways of communicating with real world inputs and outputs other than standard peripherals such as printers, vdus, etc., which are catered for in the compiler. When it comes to debugging the programs the repair cycle, which is sometimes referred to as the edit-compile-run-curse cycle, is very long.

As a result of these limitations, and also due to the traditionally high price of the hardware, real-time systems tended to be relatively small imbedded systems coded in assembler or some other low-level language. This made it possible for the programmer to take care of the fast interrupt and multi-tasking requirements. The entire program was usually written by a single programmer which again made it possible for him to keep track of all sections of the system. Debugging of these systems was done directly in the machine code and patched on line. As needs for real-time systems have grown, so have the operating systems and languages with which they are implemented. One aspect of the field that has lagged somewhat, however, is the debugging tools and techniques used to commission and maintain the systems.

Real-time considerations are extremely significant. Real-time processing requires that real world events be able to gain the control system's attention and be attended to within in a defined and guaranteed maximum period. For example a conveyor belt control system which detects that one of the belts has jammed must shut down before motors burn out and material piles up. The reaction time is also sometimes related to clock time (time of day), such

as starting remote equipment at the beginning of a shift or tagging time onto events recorded. The period is usually short - varying from a few seconds in a chemical processing plant to a few milliseconds fr.' a missile guidance or protection system.

This guaranteed real-world reaction time is not only required for single events or occurrences but must also be achieved when many events occur together. A quotation (Wolfe 1985) from a recent journal illustrates the problem. "Allan Wallack ... notes that with standard applications, when throughput lags behind predictions, the user simply waits a little longer for output data. But in real time, processing will back up. The system, unable to keep up with new input data, will 'cave in'. So when systems touted for real time don't live up to their claim, 'I get my machine back because the customer can't do his job.', says Wallack"

Process control software is a subset of real-time software. In the process control field a system is being automated. Process control systems are characterised by having a great number of inputs and outputs of diverse types - usually vr ", currents, resistances and other (itable For connecting to conventional comp

Specialized interface hardware is required. The inputs are almost invariably asynchronous and uncontrolled. This is the essence of the real-time (process control) problem. The solution lies in a combination of specialized hardware and software.

The real-time development cycle is shown in figure 1.1 . Debugging is not an isolated activity but due to the iterative nature of the cycle effects all stages of development. Consideration of debugging requirements at all stages leads to greater efficiency and shorter development times.

This report first examines what research is being done in the area of real-time debugging tools. Few researchers have concentrated on this specific field. This is unfortunate as significant productivity gains can be achieved in this area. Reliable statistics are hard to come by, and estimates vary widely, but it appears that debugging represents between 40 and 80 percent of software writing effort.

Having covered the work being done already the report then goes on to detail the unique aspects of real-time languages. The various debugging methods are then examined, followed by a discussion on

Debugging of Real-time Software

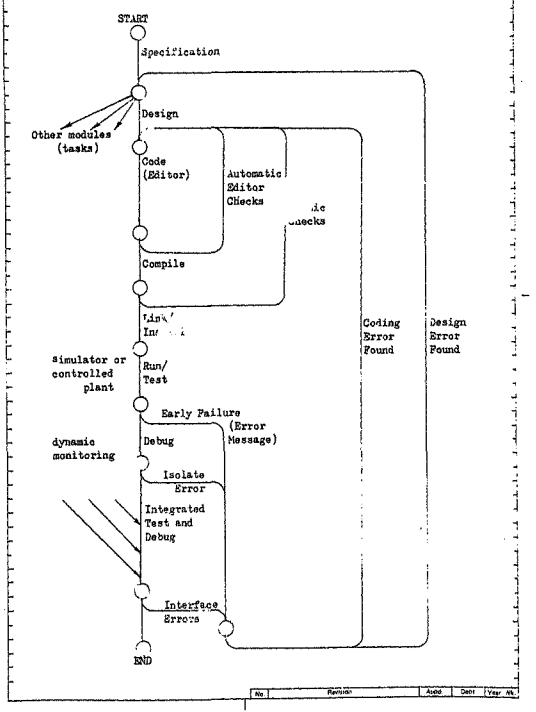
Software Development Cycle
for Real-time and Process Control Systems

figure 1.1

Sheet 1.4a
Date 1.5
Year Week

Drawing checked by RH

Drawn by RH



No	Revision	Audd	Date	Year	Wk

various debugging tools. A debugging system is proposed which overcomes most of the problems in this field and finally the benefits of the proposed system are discussed.

2.0 THE STATE OF THE ART (Literature Survey)

As stated previously debugging of real-time software is only one segment of the software field. Not many researchers have concentrated specifically on this field. There are, however, a number of articles and books in related fields which have been evaluated for this report. The papers and books surveyed can be loosely classified as :-

1) Problem identification
2) Real-time environments, systems and languages.
3) General debugging, testing and verification techniques.
4) Real-time testing and debugging.
5) Debugging concepts and ideas.

The literature will be discussed in terms of these classifications

2.1 Problem Identification

Probably the most significant paper acknowledging and identifying the problem of real-time debugging was written by Glass(1980) in which he specifically describes the problem. His analysis of 20 large

real-time projects revealed that the debugging tools were primarily machine code dumps, some breakpoint and trace facilities (machine code) and similar low-level tools. He proposes that tools be implemented on the same level as the high-level languages used for writing the original programs. He further indicates that a host computer, linked to the target system by a communication line, should be used for debugging. In a later paper [Glass(1983c)] he also discusses the increasing use of patching of machine code in real-time systems. He lists the reasons why this is undesirable.

Wirth(1977) indicates that one of the reasons for the lack of tools is that real-time programming is amongst the most complex, and therefore real-time tools are the most difficult to create. This problem is aggravated by financial considerations. Real-time projects tend to revolve around an automation system which has to be installed and commissioned according to a tight schedule and budget. Resources are seldom allocated [Glass(1980)] for developing software debugging tools.

Astrom (unpublished paper) highlights another aspect of the problem. His historical exposition shows how

the real-time (process control) industry developed out of the chemical and mechanical environments. Microprocessor technology was adopted as an available tool, but since this was a case of chemical and mechanical specialists adopting and modifying a new technology, rather than the computer field expanding in a new direction, many of the niceties (necessities) of development environments were not appreciated. Also those early systems tended to be very small, and could be handled by a single programmer without sophisticated tools, and thus the pattern was set for the situation we have today. To be fair it must be stated that adapting debugging tools from conventional software environments (batch-oriented, sequential programming) to real-time and process control is not a trivial task. The lack of development facilities has also been aggravated by the fact that systems are still usually implemented on mini- or micro-processors. These small systems cannot readily support the sophisticated tools which run on mainframe cpus.

More recent developments [Wolfe(1985)] indicate that, with more powerful systems becoming easily available, operating systems and development environments are being upgraded to match those

available in other fields of software.

2.2 Real-time environments and languages.

Debugging techniques are very strongly tied to specific environments and languages. There are several types of language used in this field. The paper by Gertler and Sedlak(1983) contains an excellent breakdown of what is available. Similarly Pike(1970) lists the different types of languages used, viz:-

1) Conventional languages (procedural or algorithmic).
2) Real-time languages (algorithmic).
3) Application oriented languages (graphic, fixed format etc.)

Initially work was done either in assembler or in algorithmic languages such as Fortran and Algol60. Because programmers are so familiar with them, languages like Fortran, Basic and Pascal are still used. Usually they are extended to provide for real-time control eg Purdue Fortran [see Gertler and Sedlak(1983)], Autran [Pike(1970)] and Prosic (derived from Basic [Meher(1976)]).

Later, special purpose real-time languages were developed, such as Coral66 (derived from Algol60) [Meek(1982)], Pearl and Procol [Gertler and Sedlak(1983)] and RTL/2 [Barnes(1980) and ICI-RTL/2 Language Specification(1974)]. Even these, however, did not go far enough. RTL/2 for example, is just a *conventional language supplemented by some real-time* synchronization facilities. Other language requirements such as structured programming, maintainability, security [MacLeod(1982)] as well as variable typing and many other developments have prompted the development of new real-time languages. Languages such as Module and Ada [Wirth( ), Barnes(1982a,b), Bishop(1985)] have fi of these needs and have provided much more sive real-time facilities.

Forth is an interesting but difficult to classify language. Devotees of Forth claim it is the best language for any programming task. Without doubt it has been successfully applied to real-time and process control tasks. [Moore and Rather(1977), Brodie(1984)]. It offers some useful features for real-time work, such as interactive programming and extensibility.

A completely different type of language has been

developed by the process control suppliers. These range from simple "ladder logic" languages [eg Telemechanique(1984)] to sophisticated multi-tasking systems with structured software, development environments, and data processing capabilities (as opposed to simple input/output control) [eg ASEA(1984)]. It has been suggested that these process control languages are not languages at all, merely application packages aimed at the non-programmer [Gertler and Sedlak(1983)]. This may be true for the simpler "ladder logic" interlocking systems and the "fill in the blanks" chemical processing systems produced by traditional control companies such as Kent, Fisher, Foxboro etc. It certainly is not true of the newer languages and systems available today which require the programmer to handle structures, databases, tasks etc. - although these are sometimes presented with different names. These newer languages provide analogue control (PID etc.), digital control (interlocking and sequencing) and data processing (text and data manipulation). They are sufficiently generalised to implement complete automation systems including control, calculation, reporting and man-machine interface functions.

2.3 General Debugging, Testing and Verification Techniques.

To begin with a distinction needs to be made between debugging, which is the process of seeking, isolating and correcting faults in the system, testing, which is running a program with different stimuli to see if it can perform its intended task, and verification, which is proving (eg mathematically) that a program is correct. Note that debugging can be included in the testing process.

2.3.1 Debugging

There is extensive literature on the practice of debugging. Articles such as Williams(1985) and Rushby(1980a,b) give a good practical introduction to the subject while Myer(1979) and others go into it in more depth. One of the points frequently made is the necessity of writing the program in an error resistant way (defensive coding). The use of structure, modularization, firewalling and similar techniques are promoted. [see eg Rushby(1979)]. Another common precaution is using debugging software or status reporting routines embedded in

the application program. [see eg Williams(1985) or Perdum(1983)]. Perdum(1983) puts forward a concise summary of debugging. Debugging has three stages:

1) detecting errors
2) isolating errors
3) correcting errors.

Brodie(1984) discusses the need to be able to move through these stages, and indeed through the whole programming cycle quickly. He therefore advocates the use of interpreted languages (specifically Forth) as being superior for debugging.

Ryback(1982) and IBM - VTAM Debug Guide(1978) are specific examples of how debugging is done on mainframe computers. Some interesting points from this area are that dedicated debugging software is written for large application programs, and even sophisticated tools like these include low level facilities like dumps. However a considerable amount of automation is included for analysing the dumps. Other techniques used are discussed in section 2.5

An example of the kind of debugging that is prevalent in the real-time world and which this report is proposing we move away from, can be found

in the paper by Landers and Tiberghien[1978]. This article typifies problems discussed in this report. It assumes (for the situation described) that debugging will be done on the target system, that debugging will be done in machine code and that a minimum of external hardware (logic analysers and in-circuit emulators) *should be used*. The article also discusses the problem of degrading the real-time response of the system by the use of debugging *hardware and software*. The situation has changed somewhat since this article was written, in that there is a tendency to use much more powerful 16 and 32 bit processors for contro . combined with multi-tasking operating systems. These give much greater assurance of achieving the *desired real-time* response. They allow inclusion of monitoring and communication software which runs at a lower *priority than* the real-time routines and thus cannot degrade response. Finally they have sufficient capacity to include sophisticated real-time routines such as time synchronisation [see Lamport[1978]] to further ensure high performance.

2.3.2 Testing

The workshop report by Miller[1978] gives a fair summary of what is required for testing of software.

The Workshop was concerned with batch-oriented programming, not real-time; nevertheless it makes a number of relevant points. Miller comments on the fact that testing is a somewhat neglected field and suggests that this is due to it being more difficult than other aspects of software. He then goes on to describe some essential aspects of software testing viz:-

1) Tests must be documented

2) The test data must be carefully selected according to the objectives of the test.

3) He makes the comment that".. the 'art' (of software testing) is in fairly good shape but the 'science' needs much more attention." Gerhart(1979) then goes on to state that much of the difficulty and expense of engineering test tools is due to a lack of a theoretical framework.

This lack partially explains why sophisticated test tools are not available for real-time software. Furthermore as was pointed out earlier with the Landers and Tiberghien(1978) article, real-time systems were (often still are) small systems programmed by a single person. As a result the need for sophisticated tools was often not perceived and even if it was, budgets would not permit their development. This same point is made by Branstad et

al (1988) in their article on the needs of the individual programmer in testing.

The Branstad article gives a very good summary of the requirements and problems of test data. This is a key problem in the field of software testing. To quote from the above article "Frequently, program domains are infinite, or so large as to make the testing of each element infeasible. There is also the problem of deriving, for an exhaustive input set, the expected (correct) responses, a task at least as difficult as (and possibly equivalent to) writing the program itself. The goal of a testing methodology is, therefore, to reduce the potentially infinite exhaustive testing process to a finite testing process. This is done by choosing a test data set (a subset of the domain) to exercise features of the problem or of the program. Thus the crux of the testing problem is finding a test data set that covers the domain yet is small enough for practical use."

Thus it can be seen that software testing (especially real-time) tends to be pragmatic rather than theoretical. This is further discussed in the section on verification below.

Other problems with testing, and with software management in general, are described by Brooks[1975]. His entertaining essays describe very real and common problems.

2.3.3 Verification

The field of program proving (verification) is attracting a great deal of research effort. The concept of proving a program correct, preferably automatically, is very appealing as it represents a major step in moving programming from an 'art' to a 'science'. It also offers immense benefits, obviously, in areas such as reliability, maintainability etc. However, despite the efforts of researchers like Gries[1981] and others, program proving is not yet practical for programs of significant size and complexity. Some progress has been made with real-time software [Bernstein and Harter(1981)] but once again it is a long way from being a practical software development tool. In my opinion a complete new generation of operating systems and languages will have to be developed before this technique can conveniently be used.

2.4 *Real-time* testing and debugging

In the real-time *field*, as with conventional *software*, there are two approaches to debugging *and* indeed, to program design.

The first approach can be described *as the* perfectionist *approach*. *The* objective is to produce *a program* which fills the specification *precisely* and is as thoroughly tested *as possible*. Ideally a program *should* be proved to be correct (verified). Redundant code is avoided as being *unnecessary* and also because it complicates the task of testing. *Elegant* software solutions are sought.

The second *approach* is more pragmatic. It is assumed that it is practically *impossible to* completely test *all the code* and highly probable (*virtually* certain) that there will be undetected errors in the system after *commissioning*. The system, *therefore*, is designed to be fault-tolerant. Working on this basis the system is *designed to fail* 'gracefully' or to *detect errors* in a running system and *automatically* repair them or work around them. The objective is to provide a *high probability* of the system running, although possibly with a

slightly degraded performance, despite software and hardware failures. These objectives are achieved by introducing redundant software (and hardware) to handle the error detection and correction requirements of the system. Note that since the fault tolerating software is not used to achieve the primary function of the system it is, by definition, redundant. In fault tolerant terms however, the additional code is not redundant but essential. Also note that this approach does not imply that the system should be less carefully tested. On the contrary, there are additional failure conditions and degraded-performance running conditions to be considered and tested. The testing is, however, of a different nature and is not as exacting as it would be for a perfectionist system.

The paper of Bernstein and Harter(1982) contains some of the early work being done in extending program verification principles into real-time software. Potgieter and Davidtz(1985) in an academically interesting experiment followed this philosophy without extensive mathematical analysis. They took a running system, disassembled it and then proceeded to reverse-engineer the software (partially automated) - querying and checking each segment of code. Finally they arrived at a

functional specification which they then compared to the functional specification for the original design. This method is hardly practical for larger systems; indeed, on being questioned, the authors explained that they had applied the technique to stand alone 8 bit processors only, with no operating system or firmware. Much more prevalent in the real-time field is the fault-tolerant approach. Kopetz(1975) and Saenger and Bahre(1981) both justify this approach and explain how it can be applied in practice.

One of the problems of error detection in a running system is that by the time the problem manifests itself the cause of the fault may be concealed because of many subsequent program statements having been executed, especially in multi-tasking systems. Schlichting and Schneider(1983) discuss the concept of error detection in running systems and postulate that the failed processor should stop immediately so that the error is isolated. Other parallel processors could take over the running of the system while the fault is being diagnosed and repaired.

Tsuruta et al(1981) have a different approach to this problem of errors being concealed by subsequent processing. They propose that additional code (and

hardware) be installed in the target system to detect and log error-related information as it occurs. Thus the difficult problem (especially in real-time systems) of duplicating the exact circumstances which caused a failure is largely avoided. There are some problems with this approach in that it requires mass storage on the running target system for the error log. The logging code must also be carefully debugged and precautions must be taken to ensure that the error log cannot be corrupted by the system going out of control when it fails.

An unusual feature of real-time systems is the programmers involvement with the hardware. Not all debugging can be done from a keyboard: other test instruments are sometimes required, for example logic and software analysers as described by Hamilton(1983) and Small(1985). This combined use of hardware and software for debugging highlights one of the unique aspects of real-time software testing and debugging. Glass(1983a) discusses the use of a host computer to input and test code which is later to be loaded into a target system. He lists the following problems in working with real-time software where a host and target system are used.

1) Target-sensitive timing errors.

2) Timing problems (the code is too slow to meet specifications).
3) Sizing problems (the code is too large to meet specifications).
4) Interface problems (the hardware behaves differently from the environment simulator).
5) Accuracy problems (the target's usually shorter word length causes too much loss of algorithmic accuracy).
6) Support software errors (the compiler's target code generator is often bug ridden).
7) Target hardware errors (these are sometimes astoundingly prevalent)

The above problems suggest that the software should be developed on the target - but with all the development facilities of the host system.

Stimoff(1983) presents the need for sophisticated real-time debugging tools and points out that testing low-probability situations is difficult, if not impossible, without a comprehensive debugger. He goes on to describe an application-specific debugging environment. One outstanding feature of the system he describes is its sophisticated trigger functions, (similar to breakpoints) which can be activated by a wide range of events and can perform

many actions, including modification of values and setting up other triggers. As can be expected the system also has good data collection facilities. These features, however, are still targeted at low-level debugging ie. absolute *memory*, *machine code* etc. Phillips[1975] also describes a test environment created for a very large project (*750 000 lines of real-time code*). He makes the point that patching of code during testing, as a function of the debugger, was one one of its most useful features and that it was essential to keep the project on schedule. This is in sharp contrast to Glass'[1983c] views on the subject. *Patching* will be discussed later in this report.

Van *der Linden and* Wilson[1985] describe the success of a general purpose debugging environment based on the Forth language. The benefits listed include:

1) Common language for all stages of testing.
2) Easy for hardware engineers to create drivers to test units.
3) The debug system is small enough to be installed and to remain in the target system.
4) The system *is generalized and* can be applied to any application.
5) Testing is interactive and dynamic.
6) The system is extendable; each programmer can

create tools to suit himself or the application. The authors also make relevant comments on faults and debugging in general for real-time, distributed and multi-tasking systems.

One example of a real-time debugging system is that of ASEA(1984). Their process control system has some interesting *debugging features.*

1) Debugging is done on the development system linked to the target system via a communications cable. (companion cpu)

2) The control system is multi-tasking and prioritized. This ensures that plant control takes priority over monitoring and debugging if necessary. *The high* level language allows the desired real-time response to be specified. Response is not just dependant on 'best possible scan time' *and therefore* response time is fixed not variable.

3) Monitoring of status in several tasks simultaneously is possible.

4) Debugging and monitoring effectively takes place on the source code (interpreter system). Patching is done directly in the source code.

Other features of this system will be discussed elsewhere in this report where relevant.

2.5 Debugging Concepts and ideas.

Some ideas not covered in previous sections are worth noting.

In real-time programming the concepts of time delays, *time measurement and synchronization* of processes is important. The work done by Lamport(1978) is significant in that he presents algorithms and equations for guaranteed synchronization between processes. With the use of the newer 16 and 32 microprocessors in control applications it becomes feasible to implement these schemes in target and debugging systems.

Automatic detection of *faults offers* attractive possibilities as has already been mentioned in connection with the work of Tsuruta et al(1981) (Non repeated runs debugging) and Schlichting and Schneider(1983) (Fail stop processors) and in general in the whole field of fault-tolerant computing. The classical cycle of fault-tolerant *computing* is

Fault detection

Diagnosis

Repair / reconfiguration

State recovery

Continue processing

Of significance to the first two stages of this cycle is the theoretical work by Kopetz(1982) with his fault-failure model.

The ability to interact with the system while it is running is an extremely powerful concept in real-time debugging. The Forth system created by Van der Linden and Wilson(1985), mentioned previously, shows this well. Another dimension is added to this concept when we consider the graphical programming concepts presented by Brown et al(1985), London and Duisberg(1985) and Moricconi and Hare(1985). Although this work was not specifically aimed at debugging the ideas are applicable. These three papers all discuss various forms of graphical programming and monitoring. Graphical representations of structures and algorithms can be easily understood and therfore easily debugged. These papers show how static flow diagrams and sketches of program structure can be turned into dynamic programming environments. Industrial process control systems started using graphical debugging techniques for logical circuits some time ago (eg Modicon, Allen Bradley, ASEA, Siemens and others) and they have proved very successful. The

concept is being extended to include other forms of programming and control.

2.6 Summary

A number of papers, articles and books are available on this subject, as this survey shows. Unfortunately this area has not been as thoroughly researched as other areas of software. Some of the references are fairly informal. The potential for increased performance and productivity justifies much more research than is currently being undertaken.

In summary, debugging tools are still primitive and require considerable development. Real-time and process control environments have unique problems which have hindered the development of suitable tools. General debugging techniques are applicable but need to be extended in some cases. Theoretical verification techniques are not yet practically applicable. New developments in the microcomputer field could make debugging much easier.

3.0 REAL-TIME LANGUAGES AND ENVIRONMENTS

Debugging tools are of necessity closely related to the language and environment with which they are used. It is highly desirable to be able to use the same development and debugging tools regardless of which language the program is written in but this is not always possible. For debugging tools to be as versatile as possible one must take into consideration the general characteristics of real-time languages and their associated development environments. From the literature surveyed and from personal experience it is possible to draw up a list of desirable features of real-time languages. The debugging facilities proposed in this report are intended to work with both available and still developing programming languages . Accordingly the list of features given below is an extract of the features of several languages. Most, if not all, existing languages fall short of having all these features but if an improvement in the performance of debugging tools is to be achieved then all the features mentioned should be taken into consideration.

The literature survey describes and discusses three

basic types of real-time control languages. The first type is high-level languages incorporating real-time and input/output handling structures. The second is assembler-level languages and the third type is the process control languages developed by industrial control system vendors. The use of assembler is declining in this field as in other fields of software. As languages and hardware become faster and more powerful so the benefits of assembler are outweighed by the convenience of high-level languages.

The following features are required of a real-time process control language. (taken in part from McLeod (1982)). Some of those listed are dependent on the development environment as well as the language.

1. Powerful process related instruction set and/or extensible language. This includes standard modules or structures for handling the many and diverse inputs and outputs found in process control systems.

2. Free format. ie. Algorithms and overall program structure are chosen by the programmer to suit the application. Applications are not limited by a rigid and narrow programming structure.

A balance must be achieved between points 1 and 2 with Fill-in-the-blanks process control application packages on the one hand and completely generalised languages with no real-time or process control functions (eg standard Pascal) on the other.

3. Structured hierarchy; ie. The application program is broken into separate modules which can be written and tested in several sections by several people. Nesting of sub-modules and creation of user-defined functions are also notable features.

4. Inherent real-time structure ie. control over the real-time response from within the language.

5. Extensive checking of syntax and structure.

6. Commenting facilities. To be effective, comments must be interspersable anywhere in the source code.

7. Both global and local variables.

8. Variable typing with type checking.

9. Good, plain English error messages as well as

codes.

10. Exception handling facilities.

11. Multi-tasking with facilities to control tasks (start, suspend etc) and facilities for task to task communication.

12. "Help" facilities which are on-line during both development and commissioning.

13. Self documenting; ie Source code is extractable on the run time package by interpreter, disassembler or decompiler on-line.

14. Interactive programming. This is the ability to move quickly from coding to testing and back to coding again. It therefore implies that there is an on-line editor and compiler (or interpreter).

15. Off-line program entry (editor), compilation or interpretation, syntax checking and simulation. Simulation to be combined with suitable performance measuring tools. (hardware or software)

Debugging techniques must be designed to match the above features. To a certain extent the features

dictate the debugging tools required. For example the timing of operations is a critical factor in a real-time system (see point 4 above). The debugger should therefore have ways of monitoring and acting upon time differences, time delays and time of day. Another example is that in the debugging of a multi-tasking system the debugger should be able to monitor and interact with more than one task at a time.

Most of the features listed are similar to those for a conventional (batch oriented) programming language. The differences lie in two areas

1. The extensive handling of various types of inputs and outputs to the real world.
2. The time factor. Execution time and response to stimuli must be defined explicitly.

4.0 DEBUGGING METHODS

There are several approaches to the problem of debugging a large program. The method selected will depend on the application, system architecture and personal preference. For example static checks are applied during code input and compilation. When the program is first run the designer will rely on a combination of system messages to report gross errors, and use of "most likely" test data to see if the program will run at all. During development, stubs will be used to test modules of supervisory code and headers will be used to test hardware driver modules. Later on in the testing process more stringent methods will be used, such as critical data sets, program coverage and so on.

Different methods and different tools may also be used to debug different types of errors. Errors may be of the following types:-

1) Syntax errors
2) Logical errors

and for real-time and process control systems we add

3) Timing errors
4) Interface errors.

Errors of type 1 can usually be eliminated by static

checks before the program is executed. Type 2 errors may be isolated by running the program under a variety of test conditions either to try to find more errors or to prove that there are no more errors. Type 3 and 4 errors are the most difficult to eliminate. Methods such as performance monitoring and simulation are usually used.

Note that there is a distinction between a debugging method, which is a philosophical approach, and a debugging tool, which is a specific operation on a system.

Different methods, currently in use, are described here.

1. Static debugging :- Syntax checking, array bounds, type checking, reserved words, proper termination of loops etc. This is usually done by the compiler or interpreter. It can also be done, often more effectively, by humans. In this case it is called walk-throughs, or desk checking, or peer review. An automated check can see only if the rules for entering the program have been obeyed. Checking by other programmers will also identify many other errors; even subtle interactions between parts of the program.

2. Program coverage (error coverage):- An attempt is made to test the whole program logically. This can take several forms:

2.1 Statement testing; test each statement in the program at least once.

2.2 Branch testing; test every branch of the program at least once. This can be done by inserting a statement into each branch to increment an array and later checking the array for percentage of program coverage.

2.3 Path testing; test every logical path through the program (all permutations!). See Bransted et al (1968)

3. Test data :- A set of inputs to the program for which the desired outputs have already been calculated. Several types of data can be used.

3.1 Most likely data :-Use the data with which the program will most commonly be use' In real time or process control environments this is equivalent to operating the control system in the way in which it will normally be used.

3.2. Critical data sets :- Test data most likely to uncover (certain types) of errors. Critical data sets should test the extremes of the program and must include data sets which will just pass through the program as well as those that will cause the program to *fail in predefined ways*. A simple example in testing an integer arithmetic routine would be the use of negative, zero and fractional values.

3.3. Simulation :- A software and/or hardware system which simulates the operating environment upon which the target program is to operate. Simulation is a method of applying all forms of test data to a program with total *control over the controlled* plant. This allows otherwise impossible or dangerous scenarios to be tested.

4. Module testing :-

4.1. Stubs :- This is "top-down" debugging. The higher level routines are written first and simple pieces of code are written in place of the lower level routines to *allow the routines to be tested*. The program is built up by writing successively lower layers of the program and testing them.

4.2. *Headers* :- "Bottom up" debugging. This is the inverse process to the one above. The lowest level routines are written first and used as a base on which to build the upper layers. Frequently used by hardware designers to test hardware drivers.

5. On-line monitoring :-

5.1. Snapshots :- The running program calls a test *procedure which then* lists out all variable values and other program (and plant) status data.

5.2. Status logging :- All critical events and values are logged as they occur. In the event of a failure this log is analysed to determine the cause. This approach is the essence of the "Non repeated runs approach" described by Tsuruta et al.(1981)

6. *System messages :- Messages are generated* by the operating system and the kernel indicating some unlikely or impossible operation. In the better systems these messages are in plain English as well as having reference numbers. The newer languages and operating systems allow the user to define his own errors in the application program. These are handled and reported in the same way as the

predefined system errors.

7. Performance monitoring :-

7.1 Software techniques include counts, execution tracing (eg. by line number or by variable value), imbedded code, or, on a lower level, breakpoints or catchpoints and so on.

7.2 Hardware approaches include on-line analysers, in-circuit emulators, etc .

8. Program proving :- Each stage of the development and each module is proved to be complete, consistent and correct.

In debugging, as elsewhere, prevention is better than cure. A technique known as "firewalling" (strict modularisation of code) is very useful in real-time work. In fact, because of real world simultaneity and parallelism of events and operations, it is essential that the software be written in a highly modular fashion to enable it to be tested with any degree of confidence.

As far as process control software is concerned the above approaches need to be modified somewhat. For

example, instead of a file of test data values it is far more likely that an operator will be used to operate banks of switches and knobs while getting feedback from lamps and dials to simulate the operation of the plant and to create abnormal conditions for testing.

5.0 DEBUGGING TOOLS

5.1 A Selection of Tools

Whichever method is used for debugging, a v' iety of tools will be required. There are many different tools in use in different environments -both dp and real-time. This chapter highlights tools which are, or could be of use in real-time environments. Where there is a specific source for a particular tool or idea it is mentioned in brackets after the description.

Adapting these tools for use in real-time systems poses some special problems. Real-time and process control systems usually run in a target machine that does not have the sophisticated peripherals of the development environment. In fact it is common for the processor not to have even a vdu connected in its application configuration. *Several solutions* are possible. One approach is to have a portable debugging tool *which could be as simple as a* handheld vdu (in which case the debugging software *must reside in the target system*). The problem with this approach is the software overhead required in *the target system*. Problems can occur with both memory usage and cpu load. Another approach is to

have a complete development system with screen, floppy disks, printer ports and extensive software. This development system is connected to the target system via a special port. A variation on this, used with smaller systems, is in-circuit emulation where the cpu of the target system is removed and replaced with a plug and umbilical cable to the development system. Yet another approach can be used if the system is connected to some sort of network, as is becoming more common. In this case the development system can be hooked up to the network and can communicate over the bus. This presupposes, however, that the bus and the target system's communication software are fully operational. The tools below assume that one of these solutions or something similar is being used.

1. Error Messages related to source code.
For run time fatal crashes (task aborts) an error message must be provided, as well as a pointer to the source code which caused the error. This could then be extended to invoke an editor automatic y and position the cursor at the error in the source code. Since real time systems do not usually have the source code and editor loaded in the target machine, an alternative is for the error message to provide a code, which the editor uses to position

the cursor.

[This idea comes from Turbo Pascal]

2. Watchdog timer

This is already fairly standard in real time systems. A simple watchdog is a hardware timer which must be serviced by the software periodically (typically once per second). If the software should ever lose control and fail to service the watchdog *in time, then the timer* times out and causes a non-maskable interrupt, which causes the system to shut down in an orderly fashion and to activate *some sort* of alarm. This could be improved by extending the interrupt routine, such that it attempts to identify the task, address, register contents and pointer to source code (see above) at the time of the failure.

3. Symbolic trace

A *trace facility* provides the following features: Firstly the program counter is monitored and displayed. To be of practical use the line number displayed must relate to the source code program. This is fairly easy for programs written in BASIC or its derivatives as the line number is inherently part of the program structure. For other languages the current procedure or subprogram could be

identified and, in conjunction with the error message facility, display a pointer to the source code.

[This tool is implemented in several versions of Basic]

3.1. One interesting possibility raised by the advent of windowing software, which requires a fairly fast development system cpu, is the idea of having the source code displayed in a window with the cursor indicating the actual code within a task that is being executed at any one time. Of course to allow for an efficient compiler being used, it will probably be necessary to limit this to the area of code within one or two lines, rather than the actual word or function.

[This has been implemented in the C language with a product called Visible C]

3.2. The second feature required of a symbolic trace is the ability to trace changes in variable values. A table of values of interest is built up, complete with names and engineering units, and the values in this table are updated dynamically while the process is running.

[this idea comes from AssaMaster]

4. Dynamic memory map. Tasks are dynamically displayed on a memory map. This is most useful for systems using virtual memory techniques, and will generally be used in the early stages of the project. This idea could be modified to show a comprehensive picture of system loading.

[This idea comes from the AMO function in RSX-11M]

5. On-line interpreter or compiler. The advantages of patching source code in Basic are renowned. If these advantages can be combined with structured programming then a formidable tool emerges.

6. On line documentation. If source code documentation is obtainable from the running system then all arguments about validity of documentation are avoided. This facility is generally available in interpreted languages.

7. Transaction Log. A common method of debugging data-processing systems is inserting code into the program to monitor and report milestones and performance statistics. These are typically output

to the system console (system manager's terminal) or a printer or some other device not controlled by the application program. In real time, and particularly in process control, there is often no terminal attached to the target system. Therefore to implement this on real-time systems, the (run-time) operating system should have a logical unit or software port into which the target system can write. This device will be a file in memory or on disk, if available, which must be protected from being overwritten if the system "runs away". Data stored here can be extracted and analysed off-line.

5.2 Extensibility

If the language being used is extensible, as was suggested earlier in the section on real-time languages, then debugging becomes more difficult in some respects.

The debugging tools used should also be extensible. This allows them to evolve as real-time languages evolve and as applications demand. Programmers will also be able to hand-craft tools to suit themselves. For this reason there needs to be access to underlying routine and structures of the debugging environment; ie. it cannot be a "closed" system.

5.3 Patching of Software

Two opposing viewpoints on the subject of patching are found in the literature. The first is that of Glass who speaks out strongly against patching in his paper on the subject.[Glass(1983c)] He cites the following disadvantages:

"a. Patches are coded in a numeric and specialized language with which the programmer may be unfamiliar.

b. Patches must be assigned vacant storage in memory, a task which is easier said than done. For example, assignment of a patch to an already assigned patch area is a common problem.

c. Patches are only a temporary expedient. The 'real' correction probably must be made in program source code, meaning the correction is done twice, often not using the same algorithm.

d. Patch insertion into the computer requires unusual techniques. For example, patches may be entered into the computer from its console, an error-prone process in itself which also retains no written record. Or patches may be punched into 'binary' card decks, a process for which no proper equipment has ever been defined, and one often requiring the

circumventing of a card-reader error check process called 'checksumming'. "

The second viewpoint is that of Phillips (1975) who gives the following advantages of patching:

a. Patching frees the program tester from source recompilations.

b. *Patching provides a great* deal of flexibility.

c. For extensive test runs, patching allows small changes in programs and data to be done quickly between runs.

d. For the large project discussed in the paper Phillips felt that patching was an *essential* feature in keeping the project on schedule.

The problems that Glass describes are avoided if all patching is done in the source code (high level) and not in machine code. To gain the benefits described by Phillips it is also necessary both for the *compilation or interpretation* stage to be very fast, and for the editor to be on-line in the debugging system.

Overall, considering both the benefits and disadvantages, patching tools should be provided in a system but the limitations discussed must be borne in mind and applied to the design of the tools.

5.4 DOS Debugging Tools

Some special forms of debugging tool are found in disk operating systems (DOS). Disk operating systems usually have a built in debugging utility. *MS-DOS calls it DEBUG, CPM86 calls it DDT86,* DEC systems (RSX11 et al) call it ODT (Online Debugging Tool). These tools all have very similar commands and functions. One limitation that they all have in common is that, although some of them are part of multi-tasking and/or multi-user sytems they can each manipulate only one task or program section at a time. Table 5.1 lists some sets of commands grouped by generic function.

It is interesting to note the symbolic debugging features of the ASEA debug routines (Pascal breakpoints, etc). *The Unix operating system has a* similar facility. Unix includes a command word "adb" which stands for "symbolic debugger" (Graff and Weinberg 1983). It includes features like breakpoints, variable value tracing, single-step execution and even function execution and function call tracing - all of which are related to the high-level language C, which is running on the operating

Table 5.1 DOS Debugging Tools

	DEBUG MS-DOS	ODT RSX 11	DDT86 CPM/86	ASEA Debug
Memory Manipulation	Dump Compare Enter Fill Move Register Search	Oper/Display/ Modify (relative & absolute addr) Fill List Status word Directive word	Display Fill Set Move X (reg` Block comp	Read (rel, abs, task) Write Open
I/O (port)	Input Output	-	-	-
Arithmetic	*Hex*	*Offset* +, -, =	*Hex*	-
Execution & Monitoring	Go Trace Assemble Unassemble	Go Proceed Single Break points(8)	Go (brk) Input cmd Trace Untrace (mon) Assemble List (disasm)	Debug Points -Define (abs, rel, type, counter) type=Pascal, Assembler catch data, trace, brk -Enable, Disable -Proceed -List, Move Task log
Disk Utilities	Load Write Name		E (load) Read Verify Write	use DOS commands
Terminate	Quit	X	^C	n/a

system. In other words the variables, statements, functions etc are part of the high level code but are accessible from the *the operating system.*

5.5 A Programmers Workbench

A number of different tools have been discussed in this chapter. These tools are, or could be, used in real-time systems. There are also many other tools and ideas which could be *relevant in this field.* The best and most appropriate of these tools and ideas must be selected, modified if necessary, and *combined* into a single integrated system. This then is the concept of a programmers workbench. All the tools should be easily available and could be applied to any problem at hand. This also forms the basis for the debugging system proposed in the next chapter.

8.0 FUNCTIONAL SPECIFICATION OF A PROPOSED DEBUGGER.

8.1 General

This specification is only a conceptual design exercise, in that the system has not been built. The purpose is to define clear directions for future projects and research effort. The system is defined in the form of a functional specification of the debugging tools that are desirable for real-time and process control software. The operating environment is also defined, as this is necessary for the tools to work together as an integrated system, rather than just having a collection of separate debugging utilities.

This specification describes a debugging facility for real-time software. This facility is hereafter referred to as "the debugger". It is intended that the debugging unit be connected to control processor(s) for the purpose of testing, monitoring and modifying software. The debugger requires both hardware and software to be provided.

A major requirement of the debugger is consistency. The same language or command syntax should be used by

the operator for all debugger functio .. The debugger is to be used for debugging real-time *software* in the remote control system. It is assumed that a minimal level of fully operative communication software resides in the target control system. Sophisticated communication routines and operating systems could be downloaded and tested from the debugger if necessary.

In specifying the debugger almost all the components *required* for a complete development system have been specified. This is incidental and this specification considers only the debugging requirements. This implies firstly, that this specification could be extended to cover a complete development system, secondly, that an *existing development system* could be extended to cover the requirements of this specification.

6.2 Overview.

A schematic of the system is shown in Fig. 6.1. There are three *major sections*.

6.2.1 The process controller(s). This is the 'target' system and does not form part of the

Debugging of Real-time Software

Proposed Debugger

Overview

figure 6.1

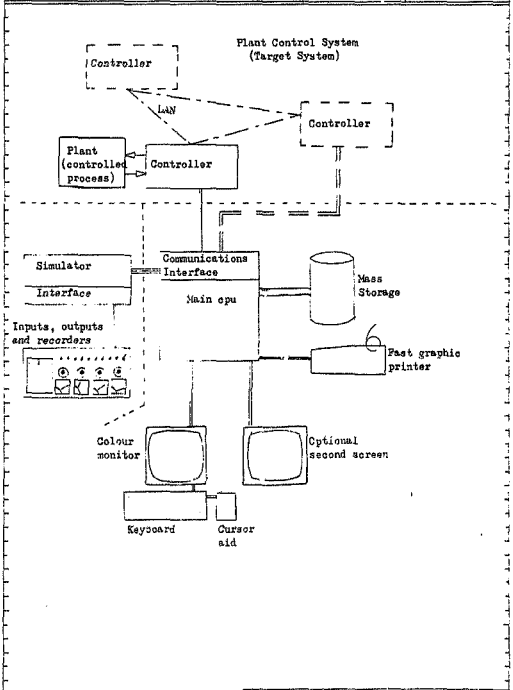
Sheet 6.2a

Contr. 6.3

Year Week

Drawing checked by RH

Drawn by RH



No.	Revision	Appr.	Dept.	Year Wk

debugger.

6.2.2 The debugger. This is the main unit containing central control, operator and communication interfaces, storage and all relevant softwere.

6.2.3 Simulator. This is a second phase extension to the debugging unit. It is used for simulating the plant and designing and testing control strategies .

6.3 The Controller.

The controller (target system) interfaces with, and controls, the process. The details of this system are not defined by this specification, but a typical control system suitable for use with the debugger will have the following characteristics:

6.3.1 It will be a real-time system controlling a process. This process is not specified but could be a plant or laboratory automation system, machine control, communications network supervisor, aerospace tracking and control system or any other form of real -time control system.

6.3.2 The controller will be multi-tasking. This feature is required to enable the system to control the process and communicate with the debugger. It is assumed that the controller will have its own operating system. As a minimum, the controller should have *firmware* to allow the operating system to be downloaded from the debugger and activated.

6.3.3 The controller may be a multi-processor system and/or may be linked to other control units on a local area network (LAN). IF a LAN is available then the debugger should be able to access any unit on the network via the LAN.

6.3.4 The controller must be able to communicate with the debugger over a high speed communication link. Communication handlers may need to be written for the controller. This communication link may be synchronous or asynchronous, serial or parallel. The controller must be able to recognise a high priority interrupt over the communication link. The communication interface should preferably be implemented in hardware as far as possible to obviate the need of debugging communication software.

6.3.5 To interface with the debugger while running, the controller may require some additional tweaks.

Tasks such as event handlers, task monitors etc. could be written and downloaded via the debugger.

6.3.6 The software to be debugged will be written in a suitable high-level real-time language. Compilers or interpreters for this will be available in the debugger.

6.3.7 The controller will preferably have a standard operating system and high-level language.

6.4 Debugger Hardware.

The specific hardware for the debugger is not defined but left for the *implementor* to *choose*. The facilities required are defined and recommendations *are made as* to suitable *hardware* units. The implementor is free to make substitutions if *appropriate*. For instance it is recommended that mass storage be done on a Winchester disk. If appropriate and available, bubble memories could be substituted.

To satisfy the functional requirements of the system a general purpose computer is required. It is recommended that a high performance personal computer

be used. It is strongly recommended that a standard system is used rather than a special purpose one. Implementation of the debugger will provide a sufficient challenge in itself without adding the further problems of designing and building a general purpose computer system at the same time. The facilities required from the system are described below.

6.4.1 Operator Interface. The operator needs to communicate through a keyboard. A minimum of ten softkeys is required for the operator to set up special commands to the control system. The operator will require a colour-graphic vdu for output with the option of adding a second vdu to provide greater output flexibility.

6.4.2. Cursor aids. For faster input and manipulation of data, a cursor control aid is required. This should be a 'mouse', lightpen, touchscreen or some other device which is reliable and available for the hardware being used.

6.4.3 Communications interface. The debugger requires a communications interface to match the one in the controller. See 6.3.4.

6.4.4 Mass storage is required for

storage of programs - source and object code

storage of logged data from the target system

storage of event configurations and output formats.

storage of user-written debug tools.

To satisfy the fast monitoring requirements the unit must be a fast access type. It must also be able to store large amounts of logged data for analysis.

Recommended device is a Winchester-type fixed disk with an associated floppy disk for backup etc.

6.4.5 The unit must be equipped with a battery-backed real-time clock and must either be equipped with hardware timers (to provide 1 millisecond resolution) or must be able to support such timers in software.

6.4.6 The unit must be expandable. One expansion that is already planned is the addition of an on-line simulator which will be connected to a test panel. The test panel will provide digital, analogue, and any other form of I/O required to allow an operator to simulate the operation of a plant. The simulator will also allow complete target system scenarios to

be simulated and tested in software. Expansion ports or something similar must be available for this interfacing.

6.5 Debugger Functions (Software).

The proposed architecture of the software is shown in Fig. 6.2. Each block represents a functional unit. How these are broken into tasks, procedures, data structures etc. is not specified herein. The function of each subsection of the system is specified.

6.5.1 Kernel: The kernel contains the operating system, supervises communication between functional units, handles allocation of resources and synchronises the system as a whole. It is recommended that the operating system be identical to, or compatible with the operating system in the controller(s) (target). This will greatly simplify passing of *files, messages etc.*

6.5.2 Communications with controller: A communications driver is required to act as an interface with the main controller. The following features must be implemented:

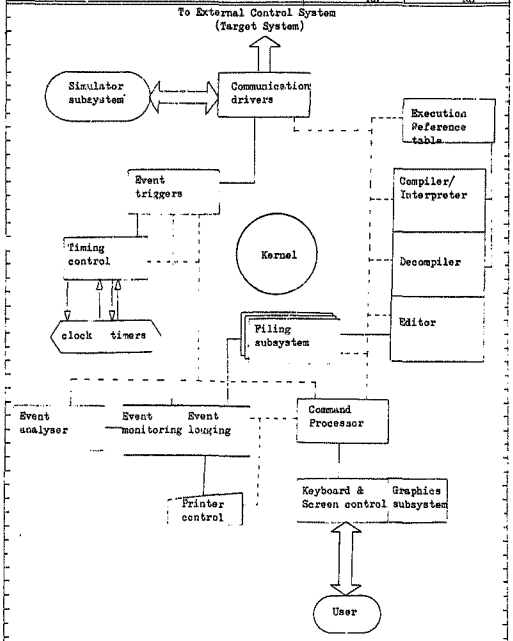
Debugging of Real-time Software

figure 6.2

Sheet 6.8a Cont. 6.9

Proposed Debugger Software Architecture

Drawing checked by RH | Drawn by RH



No. | Revision | Appd. | Date | Year Wk

6.5.2.1 Communication can be in either direction and be initiated by either host or target.

6.5.2.2 The communication system must be a full duplex system and must guarantee no loss of data.

6.5.2.3 The debugger must be able to generate a high priority interrupt to gain the attention of the controller if necessary.

6.5.3 Event Triggers: The debugger needs to be able to initiate actions when events take place in the controller.
The events that can be monitored include:

Start up a task
Stop a task (or task aborts)
Application program breakpoints, defined wrt. the application programming language
Change of status of I/O values in the controller i.e. *some controlled process changed state.*
Reading of state can also be initiated by the debugger based on operator commands, clock time or *timer.*
Any boolean combination (AND, OR, XOR, NOT) of the above events.

The action that can be initiated when an event occurs include the following:

Pass a message to the logging/printing routine (logger)

Trace a routine and output the results to the printer or console

Start a timer

Stop a timer, record elapsed time and pass a message to the logger

Execute a sequence of debugger commands (macro) which allows it to manipulate other event settings, temporarily take control of the screen, manipulate timers, manipulate the logger.

Every event will automatically be time-tagged with time-of-day from the real-time clock. The event handlers will also be able to interface with the clock and timers for polled (scanned) updating. A *link will also be required to the execution reference* table (see 6.5.7) to provide event trigger with run-time information on where to find application program events.

6.5.4 Event Logging (the "logger") The logger will

accept commands from the user to log events to disk, printer, screen or a combination. Formats for logging (e.g. explanatory text, colours, audible alarm triggering, numbers in decimal or hex etc.) can be set up for specific events or classes of event. This facility will also be used to specify files to be used for data storage. Associated with the logger will be an event analyser which can be set up by the user to search for patterns in the incoming data (sorting routines). Sorting will be on the basis of time or event type, or a combination.

6.5.5 Editor: A full screen editor is required to edit:

Application program source files
Event-log files (logged data)
Logging formats (see 6.5.4)
Event-trigger set-ups (see 6.5.3)
Keyboard softkey set-ups

The editor should include editing features such as time and date tagging of edited files, automatic creation of backup files when saving edited files, the ability to have more than one file open at the same time and to pass data between files, the ability to work in "windows" on the screen and finally, the

ability to execute "macros" of edit commands.

The editor will also be closely tied to the compiler/decompiler, such that system messages from running tasks can be vectored back into source code files, via the decompiler or execution reference table.

6.5.6 Compiler/Decompiler (or Interpreter): The choice of language is strongly dependent upon the target controller and therefore has not been specified here. The debugger, however, has some requirements in this area.

6.5.6.1 The language could be compiled or interpreted but if compiled then compilation should be very fast. Debugging is an interactive and creative process and lengthy compile-link cycles are detrimental to efficient working.

If the language is interpreted the interpreter will have to do a certain amount of syntax and other checking before allowing the routine to run (see 6.5.6.3).

6.5.6.2 Whether the language is compiled or

interpreted, it should include the following features:

- Structured or procedure oriented
- Variable typing and type checking
- Full alphanumeric variable and procedure names (Some reasonable limits are acceptable eg name must begin with an alphabetic character and be less than 12 characters long)
- Extensible i.e. the user can add his own functions, data structures and types.

6.5.6.3 The debugger should be as automated as possible. Therefore the compiler or interpreter should do as many static checks on a program as possible, conversant with fast turn-around. Static checks should include

- Syntax checking (aim to identify 100% of typing errors)
- Type-checking of variables, constants, records arrays, etc
- Flag undefined or uninitialised input variables
- Check for termination of loops, procedures etc

6.5.6.4 The compiler or interpreter shall create an execution reference table for each object module

compiled (see 6.5.7). Basically the purpose of this table is to provide a link between running programs and source code. Compiled modules must be tagged with a reference label so that a link can automatically be established between a running program and the appropriate execution table.

Note that these last two requirements preclude the use of a "pure" interpreter. Some pre-execution processing is necessary.

6.5.6.5 A "decompiler" is required. The exact nature of this depends strongly on the compiler/interpreter and language. The decompiler could be integrated with the execution reference table (see 6.5.6.4 and 6.5.7). Regardless of how it is implemented the following facilities are required.

Monitoring of a program, tracing of execution and variables is done with reference to the source code. The decompiler may be needed for the system to be able to display source code names and statements.

Source code listings of programs will be produced from the running programs to guarantee up to date documentation. This will require the use of a decompiler.

Patching of object modules is a requirement. *Since patching must be done* in source code the decompiler will probably be required while patching.

The definition of patching in this context is that a running program (in the control system) can be stopped, program statements inserted or deleted, values can be examined or changed and the program can be restarted or resumed, (with some limitations) *without affecting any other programs* running in the control system. Variables local to the altered program will be reset when the program *is restarted.* Variables global to the whole system or passed from other programs will not be affected in any way. Tasks in the control system communicating with the tasks being patched are not guaranteed or protected. In this case inter-task communications will be queued as defined by the normal capabilities of the control *system.*

Another way of defining this patching *facility* is that a copy of the running program can be loaded into the editor, patched and installed as a new program. Control is then swapped from the old version to the new version in a short time (say <10 milliseconds) and the the old version can then be removed from the

system.

6.5.7 Execution Reference Table: The function of the execution reference table is to provide a link between source code programs and running object modules for on-line monitoring.

It is unreasonable both in terms of storage space and execution time for the running object module to carry details of all variable names, procedure names and source code statements. This information is contained in a condensed and encoded form in the object module and in its task control block (TCB). It is therefore quite feasible to create a cross-reference table between the object module and the source code file. This table will probably be created at compile time as suggested in 6.5.6.4 .

Once this table is created, it will be used by the event trigger module to identify events in running application modules, in terms of their procedure and variable names supplied by the user.

It is possible that this reference table could more efficiently reside in the target system. This

arrangement, however, has two major disadvantages. Firstly it consumes memory space which is usually at a premium in the target system. Secondly it means that variables, programs etc must be referenced by *their full name* (ASCII string) over the communication link instead of referencing them as offsets in the TCB or similar. This places a much higher load on the *communication link*.

6.5.8 Command Processor: The command processor interprets all commands and input from the user and initiates actions such as invoking the editor, starting the compiler, etc. on the basis of those commands. Basic commands as described in the functions of the debugger will be built in. The user will be able to create and store new commands which will essentially be macros, made up of fundamental commands. The command processor will interface with the kernel.

6.5.9 Man-Machine Interface (MMI)

The man-machine interface unit overlaps the function of the command processor to a certain extent. The mmi handles all input from the keyboard and specialised devices and routes and controls all

output to screens and printer(s).

The mmi is seen as a lower level function to the command processor. It will handle functions like keyboard input buffering, "windows" on screens, (but the parameters will be set up by the command processor), interpretation of softkeys and so on. It could also handle dumping the screen to the printer.

6.5.9.1 Windows. The screen must be able to be split into a number of windows (up to 6 simultaneously). Each window can be used to run any operation of the debugger. Updating of information in windows will continue in all windows simultaneously, can be achieved by spawning multiple copies of the command processor task as new windows are activated. It must be possible to move between windows easily, for example by use of a function key or a control key sequence.

6.6 Summary

A list of desirable features has been identified in this chapter. Flexibility is provided for in the way the system is to be implemented. The hardware to be

used, for example, is specified only in terms of its functional requirements and is in no way limited to any specific system. Tools and their operating environment have been specified to ensure that the system is coordinated.

The system has been designed to cater for all stages of debugging, starting with static tests on program entry, going through various tracing and patching tools for testing and commissioning, and providing for up-to-date documentation at the end. The man-machine interface has also been provided for. The debugger is thus a complete and convenient system with facilities to handle any real-time software project.

7.Ø CONCLUSION

This report has detailed the status of real-time debugging, and a debugging system with a complete set of tools has been proposed. The proposed system has many benefits. It is flexible and the programmer can build his own special tools as required. All commands and operations will be *consistent*. This is achieved by filtering them through the command processor. The use of a separate cpu for the debugger, combined with an interpreted language, means that a fast development cycle can be achieved. The separated debugging system also means that extensive *debugging* facilities can be provided without being limited by the target system's available time, space and peripherals. The critical question of timing has been addressed by the debugger, with its timing and real-time tools. Powerful event trapping facilities have also been provided. The event facilities also provide the necessary tools to allow a transaction log to be created, with all the benefits that accrue from being able to locate intermittent faults that occur in real time. The sorting and analysis tools are an important part of this facility. The ability to patch source code *contributes to the fast*

development cycle as does the ability to obtain source code documentation from the running system. Finally the use of windowing software,combined with a multi-tasking operating system, addresses the problem of testing more than one section of the program at a time. It even permits totally different kinds of tests to be applied simultaneously.

This report has highlighted an area of the software field that is all too frequently overlooked. Significant gains in productivity and performance can be achieved by more efficient and convenient debugging techniques and it is necessary that more attention be focussed on this area in future. Before further research work could be done it was necessary to identify the problem, to evaluate carefully what has been done already, and to define directions for new research work. In specifying debugging tools to be built it was necessary to place some limitations on them. The major limitation is that the debugger will tend to be tied to a particular language or family of languages. This is because debugging is inherently related to the language in which the program is written. This limitation can be overcome by adding compilers to the system for various languages. In practical

terms this may be equivalent to transporting the specified tools to a completely new language and environment.

Although the proposed set of tools is significantly more advanced than any existing system found, its implementation is very possible. Each of the tools has been developed out of existing tools on existing systems. While implementing the tools would certainly be a challenge it is entirely feasible.

Difficulties were experienced in preparing this report, mostly due to the lack of references in this specific field. A further problem is the lack of theoretical background in the field of debugging in general. While the use of theoretical analysis for debugging is attractive, it cannot yet be used as the basis for a practical debugger design. The design stage of the project was therefore a very practical exercise involving the evaluation of the relative merits of various tools and techniques based on considerations identified in the literature survey. The second part of the design problem was that of combining the tools into an integrated system.

The next stage of development in this area should be

based on implementing some of the specified tools and measuring their performance.

A1. REFERENCES

ASEA Electric, (1984) MP200 Users Manuals (6 vols). ASEA Electric AB Vasteras Sweden

ASTROM, K.J. () *Process Control - Past Present and Future.* Unpublished paper, Lund Institute of Technology, Lund Sweden

BARNES, J.G.P. (1980) An Introduction to Real-time Programming. Proc. Comp. Soc of S.A. Real-time programming tutorial. Johannesburg. Aug 1980

BARNES, J.G.P. (1982a) Programming in Ada. Addison Wesley, London

BARNES, J.G.P. (circa 1982b) Overview of Ada. SPL International, The Charter, Abingdon, Oxon, England.

BERNSTEIN, A. and HARTER, P.K.(Jr) (1981) Proving Real-time Properties of Programs with Temporal Logic. ACM 12/81

BISHOP, J.M. (1985) Ada - A Status Review. Paper presented at the 1985 AGM of the Real-time Users Group of South Africa (RUGSA) Johannesburg Oct(1985)

BRANSTAD, M., CHERNIAVSKY, J.C., ADRION, W.R. (1980) Validation, Verification and testing for the Individual Program . IEEE Computer, Dec 1980

BRODIE, L. (1984) Thinking FORTH. Prentice Hall. New Jersey.

BROOKS, F.P. (1975) The mythical man-month. Reading, Massachusetts, Addison-Wesley

BROWN, G.P, CARLING,R.T, HEROT, C.F, KRAMLICH, D.A and SOUZA, P.(1985) Program Visualisation: Graphical support for software developm,ent. IEEE Computer Vol. 18, No. 8, Aug 1985.

FREEMAN, A.L. and REES, R.A. (1977) Real time computer systems. Crane Russak and Co. Inc. USA

GARCIA-MOLINA, H., GERMANO, F.(Jr) and KOHLER, W.H. (1984) Debugging a Distibuted Computer System. IEEE Trans. software vol SE10 No. 2, Mar 1984.

GERHART, S.(1979) Testing theory: Field needs definition, experimentation. In workshop report: Software testing and test documentation. IEEE Computer, Mar1979.

GERTLER, J. and GEDLAK, J. (1983) Software for Process Control. published in Real Time Software by (Edited by R.L. Glass.) Published by Prentice Hall International.

GLASS, R.L. (1980) Real-Time: The "Lost World" of software debugging and testing. Communications of the ACM Vol. 23, No. 5, May 1980.

GLASS, R.L. (1983a) Real-Time Checkout: The "Source Error First" approach. published in Real Time Software (Edited by R.L. Glass.) Published by Prentice Hall International.

GLASS, R.L. (1983b) (Editor) Real Time Software. A collection of papers. Prentice Hall International.

GLASS, R.L. (1983c) Patching is alive and, lamentably, thriving in the real-time world. published in Real Time Software (Edited by R.L. Glass.) Published by Prentice Hall International.

GRAFF, J.R. and WEINBERG, P.N. (1983) Understanding Unix. Que Corporation. Indianapolis.

GRIES, D. (1981) The Science of Programming. Springer-Verlag, New York

HAMILTON, G. (1983) Logic analyser gives programmers real-time view of software performance. Electronic Design News. 5 May 1983.

MEHER, A.D. (1978) Real-time Interactive Multiprogramming. CSIR special report ELEK150. Pretoria, June 1978.

IBM Corp(1978) VTAM Debugging Guide, VTAM level 2. IBM Corp Kingston NY> Dec 1978.

ICI (1974) RTL/2 Language Specification. Reference :1 version :2. ICI Ltd

KOPETZ, H. (1975) Error detection in real-time software. Proc. 11th IFAC Real-time programming workshop. Boston.

KOPETZ, H.(1982) The Failure-Fault model. Proceedings of IEEE

KOPETZ, H. (1979) Software Reliability. Macmillan computer science series.

LAMPORT, L. (1978) Time, Clocks and the ordering of events in a distributed system. Comm. of the ACM vol 21, No. 7, July 1978

LAUBER, R.J.(1984) Software for Industrial Process Control. Guest editor's introduction to special issue. IEEE Computer, Feb 1984.

LENDERS, P. and TIBERGHIEN, J. (1978) Debugging aids for real-time microprocessor systems. Euromicro Journal, Vol. 4, No. 4, July 1978.

LEVESON, N.G. (1984) Software Safety in computer controlled systems. IEEE Computer vol 17, No. 2, Feb 1984.

LEVESON, R.L. and DUISBERG, R.A. (1985) Animating Programs Using Smalltalk. IEEE Computer, Vol. 18, No. 8, Aug. 1985.

MacLEOD, I.M. (1982) The application of modern software engineering techniques in the development of a process control package.

MATSUMOTO, Y. (1984) Management of Industrial Software Production. IEEE Computer vol 17, No. 2 Feb 1984.

MEEK, B. (1980) Real-time Programs. Published in Guide to Good Programming Practice. Edited by B. Meek and P. Heath. Ellis Horwood ltd. (John Wiley), Chichester, England.

MILLER (Chan) (1979) Software Testing and Test Documentation. Workshop report: Software testing and test documentation. IEEE Computer March 1979.

MOORE, C. and RATHER, (1977) Forth in Process Control. International '77 Mini-Micro Computer Conference, Geneva, May 1977.

MORICONI, M. and HARE, D.F. (1985) Visualising Program Designs Through Pegasys. IEEE Computer, Vol. 18, No. 8, Aug. 1985.

MYER, G.J. (1979) The Art Of Software Testing. J Wiley, USA

PARDUM, J. (1983) C Programming Guide. Que Corporation, Indianapolis.

PHILLIPS, A.K. (1975) Debugging a Real-time Multiprocessor System. Bell Technical Journal. Safeguard supplement.

PIKE, H.E. (Jr) (1970) Process Control Software. Proc. IEEE vol 58 No. 1 Jan 1970.

PLATTNER, B. (1984) Real-time Execution Monitoring. IEEE Trans. Software, vol SE10 No. 6, Nov 1984

POTGIETER, L. and DAVIDTZ, T. (1985) The SATS Software validation procedure. Paper presented at the 1985 AGM of the Real-time Users Group of South Africa (RUGSA) Johannesburg Oct(1985)

RUSHBY, M. (1982) Structured programming and error prevention. Published in Guide to Good Programming Practice. Edited by B. Meek and P. Heath. Ellis Horwood ltd. (John Wiley), Chichester, England.

RUSHBY, M. (1982) Testing and debugging. Published in Guide to Good Programming Practice. Edited by B. Meek and P. Heath. Ellis Horwood ltd. (John Wiley), Chichester, England.

RYBACK,M. (1982) MVS Debugging Aids, Part 1 - System. IBM Israel, CE Software Dept. (IBM internal document).

SAENGER, F. and BAHRE, R. [ ] Error Detection and Location methods in the Real-time and Fault-Tolerant Multi-Computer Sytem "ROC" - Realization and proposals for further Improvements. Unpublished report, Fraunhofer Institut.

SCHLICHTING, R.D. and SCHNEIDER, F.B. (1983) Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. ACM Transactions on Computing Systems, Vol. 1, No. 3, Aug. 1983.

SCHOEFFLER, J.D. (1984) Distributed Computer Systems for Industrial Process Control. IEEE Computer vol, 17 No. 2, Feb 1984.

SMALL, C.H. (1985) Software analyser traces high-level program execution in real time. Electronic Design News. April 18 1985.

SMALL, C.H. (1984) Integrated Development Tools. Electronic Design News. 29 Nov 1984.

SPL *International* (1980) *SMT Operating system, RTL/2* Reference :122, version :2. Published by SPL International, London, Feb 1980.

STEUSLOFF, H.U. (1984) Advanced Real-time

Languages for Distributed Industrial Control.IEEE Computer vol 17 No. 2 Feb 1984.

STINAFF, R.D. (1983) An Integrated Simulation and Debugging Facility for a Distributed Processing real-time Facility. Published in Real Time Software. (Edited by R.L. Glass) Prentice Hall International.

TELEMECHANIQUE. (1984) TSX47 Users Manual. Telemechanique France 1984.

TSURUTA, S., FUKOKA, K., HIYAMOTO, S., and MITSUMORI, S. (1981) Debugging Tool for Real-time Software - Non Repeated Runs Approach. 11th IFAC workshop on real-time programming. Kyoto Sept 1981

VAN DER LINDEN, F. and WILSON, I (1985) An Interactive Debugging Environment. IEEE Micro, Vol. 5, No. 4, Aug. 1985.

WALTER , C. (1984) Control Software Specification and Design: An Overview. IEEE Computer, vol 17 no. 2, Feb 1984

WEIDE, B.W., BROWN, M.E., RAMANATHAN, J. and SCHWAN, K. (1984) Process Control; Integration and Design

Methodology Support. IEEE Computer, vol 17 no. 2, Feb 1984

WILLIAMS, G. (1985) Debugging Techniques. Byte magazine, June 1985

WIRTH, N. (1983) *Towards a discipline in Real-time* Programming. Published in Glass, Real Time Software.

WOLFE, A. (1985) Real-time Operating Systems Join the Computing Mainstream. Electronics, Aug 15, 1985.

Author Helps Cedric Richard Gordon
Name of thesis Debugging Techniques For Real-time And Process-control Soft Ware. 1985

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2013

LEGAL NOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.