Computing with REAL-TIME SYSTEMS

Volume 2

Proceedings of the
Second European Seminar
University of Erlangen – Nürnberg

TRANSCRIPTA BOOKS

Computing with REAL-TIME SYSTEMS

Volume 2

Proceedings of the

Second European Seminar

University of Erlangen – Nürnberg

Edited by
I.C. PYLE
AERE Harwell
and P. ELZER
University of Erlangen

TRANSCRIPTA BOOKS

30 Craven Street, Strand, London WC2

Publisher: S. Chomet

Printed in the United Kingdom by Shortlands Press Ltd, London

CONTENTS

PREFACE

Following the First European Seminar on Real-Time Programming at Harwell in April 1971, this Second Seminar was held at the University of Erlangen, Nürnberg, in April 1972. It was organised by P. Elzer of the Physikalisches Institut of the University, assisted by Mrs Brehm, with further help from K. Pelz and P. Holleczek. As at the first seminar, special periods were allocated for 'workshop' discussion sessions covering particular areas, and notes on these discussions are included. (The absence of identification of the speakers in these discussions is deliberate, to emphasise that speakers' views are personal and do not necessarily represent those of their institutions.) The papers presented and workshop notes were prepared for publication by P. Elzer and I. C. Pyle.

GENERAL CONSIDERATIONS ON REAL-TIME PROGRAMMING

Chairman:
Prof. Dr R. LAUBER

EDITORS

IAN C. PYLE, MA, PhD (Cantab)
has been working with computer software since 1956, particularly concerned with programming systems (Hartran on the ICL Atlas Computer), multi-access systems (HUW on the IBM 360 computer at Harwell), and multi-computer command and control systems. He is now Professor of Computer Science at the University of York.

PETER ELZER, MA (Erlangen)
has been working with computers since 1966, at the Physics Institute of the University of Erlangen. His early work concerned the development of a programming system for application to nuclear physics experiments. He was one of the initiators of the PEARL project and is now leader of a PEARL implement team at the Physics Institute. Since 1971 he has now been involved with the LTPL project, of which he is now the European Chairman.

TRANSCRIPTA BOOKS

Experiences with languages for real-time programming

D.R. FROST
General Electric Co, Phoenix, Arizona, USA

At General Electric's Process Computer Operation in Phoenix our product consists of real-time process data acquisition and control systems, with applications including the automation of power plants, power distribution, refineries, chemical plants; such processes as discrete parts manufacturing, steel making; production of cement, non-ferrous metals, paper, textiles, and others. For these automation systems we provide the computer and process input/output. We frequently provide analysis and programming as well. Since economics dictate that we will get no more money for this service than it is worth to our customers, profitability depends upon our skill in keeping programming costs down. This paper is about our attempts to lower costs through the use of higher level programming languages and application packages. I will discuss:

1. What we expect from higher level languages and application packages.
2. Our specific experiences with these various types of languages and application packages and how successful they have been.
3. Where we are headed.
4. Lessons we have learned.

Expectations

The whole rationale for using higher level languages and application packages is to save system costs through lowering programming costs. There have been many successful tools for various other computer applications: FORTRAN, ALGOL, COBOL, LISP, various report generators, sort packages, and simulation languages are just a few. All of these tools have a common objective: to produce machine-independent source programs.

Machine independence is often equated with portability between computers. Yet the truly important facet of machine independence is not so much portability; of more importance, it allows the user to implement his solution without all the computer-related clutter that gets in his way. To a problem solver this clutter is overhead, with all of that word's connotation of cost – and he is right. He wants independence from the computer.

A good programming tool:

1. Gives the programmer a vocabulary that matches his application.
2. Takes care of hardware
   - storage allocation
   - controlling peripherals.
3. Provides
   - standard algorithms
   - data formats
   - character codes.
4. Promotes good design, and allows good design to be maintained as the system gradually evolves.
5. Has self-contained debugging aids, which
   - make mistakes less likely
   - make error correction more reliable.
6. Gets rid of 'patches' [machine-oriented program corrections] and the dependence on knowledge of the computer which 'patching' requires.
7. Aids maintenance of a design as the application changes.
8. Aids maintenance of code as errors are found and corrected.

Good design is probably the biggest cost saver in software production, and good tools encourage good design. An excellent example is the successful language COBOL, which forces the programmer to define and document his data base. (I suspect that this is actually more a key to COBOL's success than is its natural English-language-like syntax.)

Successful languages and application packages such as the ones mentioned exist in the comparatively orderly world of scientific and business data processing. They fit their applications very well. FORTRAN is a good language for mathematics and logic. The vocabulary of COBOL consists of business data processing terms in common use long before computers. The environment of a sort generator package is orderly files of data. But the term 'real-time programming' causes many of us to think simplistically. It makes us believe we will find a single, ultimate tool for 'real-time programming'. But real-time applications are just as diverse as those for which we use LISP, FORTRAN, and report

generators. As a result, we are finding and will continue to find a multiplicity of 'best' tools for real-time systems, depending on the particular applications.

Good tools for real-time programming must provide the same advantages as do existing successful tools for other applications:

1. Independence from the computer.
2. A problem-oriented vocabulary.
3. Encouragement of good software design.

Let us now look at some of the tools that have been used by employees and customers of General Electric's Process Computer Operation.

Our experiences

FORTRAN – a semi–success

Soon after FORTRAN swept to supremacy as the earliest and most used mathematical programming language in the United States our customers wanted to use it to program their process applications. In the early days there was considerable naïveté about the usefulness of FORTRAN for real-time programming. It 'seemed natural' that its acceptance for mathematical applications would be duplicated immediately in process control applications. Yet even now, more than ten years later, its use is light compared to assembler language. Without considerable work it would have been a total failure (and was for some time).

First, let us look at its shortcomings:

1. Real-time – FORTRAN has no capability of recognising the passage of time or of responding to real-time data arriving at unpredictable intervals and in varying volumes.
2. Data types – it does not handle large numbers of logical variables efficiently.
3. Data communication – COMMON storage (as used in FORTRAN) is monolithic: it provides total communication or none, which works against good program design. Besides, data bases are seldom small enough to fit into one COMMON pool in main memory.
4. Program organisation – FORTRAN's subroutine method does not provide control linkage or data communication between segments operating under a real-time operating system.
5. Input/Output – Formats fit neither the needs of process input nor communications with operator consoles and other specialised devices.

In spite of these shortcomings, FORTRAN is in use today. In some companies (and some application areas) it is used a great deal. This is possible because of various attempts which have been made to remedy FORTRAN's shortcomings. These various attempts have resulted in a proliferation of FORTRAN dialects and extensions. We have two ourselves, one predating the FORTRAN ANSI standard, the other based on it.

The real-time problem is solved through providing a real-time operating system outside of FORTRAN. System functions or subroutines are made available to the programmer for interrogating time, which is kept by the operating system.

Data types are augmented by allowing various forms of bit arrays and providing operations to work on them.

Data communication problems are also handled through various extensions to the language. In one case we changed the meaning of labelled COMMON to allow more than one common area. Named system variables are another useful technique.

Program organisation is dependent on the available operating system. A program with its own subroutines is usually (but not always) considered a program segment to the operating system. Special system subroutine calls can be used to schedule other program segments.

Input/Output is a very troublesome area. FORTRAN I/O is based on such a rigid approach that many improvements have been attempted, but most have proven unsatisfactory. For example, the idea of sharing a peripheral is foreign to FORTRAN. This causes a problem when very large operating logs must be collected and stored, then put out on a typer without being interrupted by other output requests to the same typer.

There is a hopeful development: as a result of the Workshops on Standardisation of Industrial Computer Languages held twice yearly at Purdue University, a standard for FORTRAN extensions is being worked out. Although the semantics of some of these extensions are operating-system-dependent and cannot be standardised, the syntax can. It is predicted that this effort will be successful.

In our own shop the use of FORTRAN is mostly limited to power plant performance calculations. I believe the problem here is that trading off savings in program production against an increase in program size and hardware cost is difficult to 'sell'. Hardware costs are easy to calculate, but software costs are nebulous; it is not always easy to demonstrate the advantages to financial management.

Therefore, I have to consider FORTRAN as being relatively unsuccessful.

PERCON and SCANCON — application package failures

PERCON (peripheral control) and SCANCON (process input control) were two application packages intended for our first process control computer with core memory, the GE-412. PERCON was really no more than a typical peripheral I/O package. SCANCON was a table driven process scanner much like that now in SEER.

These two packages never got off the ground. After considerable design work they were killed before coding began; thus we will never know their merits. They failed because they were conceived and designed by a group that was not actually involved in the application programs; and this is fatal. Analysis and design work must be done by those who will use the package. There are two reasons:

1. They probably know better what to do.
2. Even when they do not know, you have a nearly impossible political job in selling the package to them.

COOL — a language that failed

Even in the early days it was obvious to us that FORTRAN was not the right language for process computer programming, so we tried our hand at inventing one. COOL (Control Oriented Language) was a dialect of WIZOR, a little known (but excellent) language for system programming available on the GE-225 general-purpose computer. Although it looked like other computer languages, control of almost everything remained in the hands of the programmer. Its strong point was that it gave the programmer a good syntax for operations on scaled fixed point and logical variables. The compiler translated COOL into assembly language for input to the assembler.

The language was greeted with enthusiasm by our application programmers (a study had been made of the language before the translator was implemented, both for technical and political reasons), and it was immediately used. But it fell into disuse quite rapidly. What had gone wrong? Several things:

1. The compiler existed on our general-purpose computers only, which made language patching in the field a necessity.
2. With assembly language accessible to the programmer, re-assembly without recompilation was tempting, and this quickly made the COOL statements obsolete, indeed misleading.
3. Since COOL statements quickly became obsolete, they saved time only in the initial coding of programs, which is certainly a small part of the programming job. (Our study reported positively on the language mostly because of savings in coding time; we just did not then realise what a small part of the total work coding really is.)
4. We failed to do a good job of training the programmers in the intelligent use of the language.

Our lesson here is that coding aids are not enough, and training is essential. A macro assembler failed later for exactly the same reasons.

MACRO assemblers — abortive attempts

Macro assemblers have been frequently proposed, and one was implemented. It was written by an application programmer for application programmers, which is an approach that should lead to success. Yet it was little used. It failed for the same reasons that COOL failed. It did not do enough for the programmer, and reassemblies frequently bypassed the macro processor, and there was no training of any consequence.

SEER — a big success

As power consumption in the United States has mushroomed we have been providing a great many process computers for power plant monitoring and control. An application package for power plant monitoring, SEER, has been our most successful tool for saving programming costs. It is a system of functional modules for data scanning, logging, alarming, and console input and display. It also includes support routines for performance calculations. It provides a base system which can easily be expanded to include further plant monitoring as well as plant control.

Several factors contribute to its success:

1. All steam power plants are pretty much alike; thus we had an application narrow enough to fit an application package to.
2. Programming and hardware both are nearly always purchased by power companies from a limited number of traditional instrumentation suppliers; thus programming has not been fragmented among many teams of programmers.
3. SEER evolved slowly, starting as individual programming jobs where we carried program design and code from one job to the next. We thus learned to understand the application very well.
4. It was invented by the same group which was going to use it.
5. The programmers using it knew the software as well as the application, and could

thus use it intelligently.

6. Because of our heavy participation in the market, our approach has had good acceptance from major engineering firms that design plants for the utilities.
7. We waited until we had a *de facto* near standard before we standardised the application package.
8. A team of programmers with power plant experience was assigned to write the standard, guaranteeing political acceptability as well as continuity of knowledge, experience and technology.

Although our exact costs cannot be divulged, it can be said that, working in ratios, our first plant monitoring system took about 18 units. Before the final standardisation effort, the time was down to 3 units. We can now, with our present standard application package, produce one in a single unit of effort. (In all cases this does not include special unique application software.) We are now developing a SEER for the GE-PAC 3010 based on experience with three previous process computers, this time confidently without the trial-and-error approach.

TASC – an unsung hero

Although monitoring power plant operation is quite standard, control, especially in sequential operations such as start up and shut down of the boiler and turbine, is less so.

Early in our history one of our programmers invented a sequential control language called TASC (for Tabular Sequence and Control). TASC is a typical interpretive system with

1. A sequential control language.
2. A translator of TASC statements (one-to-one) to a compact code.
3. A simple interpreter which, based on TASC statements, selects subroutines for execution.
4. A library of subroutines which actually effect the control actions specified in the language.

The language and translator have remained much the same functionally through the ten years of their existence and through several computers. The control subroutines usually require some rewriting from job to job.

This is called an unsung here because it has never had much publicity to our customers or even our own management. It continues to be one of those valuable tools that no one much knows about except the programmers using it from day to day.

Direct Digital Control – success or failure?

When discussing 'direct digital control' one immediately runs into a semantics problem. The phrase has two meanings:

1. Digital (as opposed to analog) control of a process element in any kind of application.
2. An application package for control of continuous processes only, in which analog controllers are replaced with digital controllers.

The latter definition is used, because it is the application for which our ddc packages have been written.

Direct digital control was originally conceived as a money saver. It seemed that considerable savings in hardware cost could be realised through replacing analog controllers with digital controllers, moving control calculations into the computer. Since this has turned out not always to be true, ddc has not become the control method that its advocates had desired and expected. Indeed, interest appears to be waning.

In addition to a good many special ddc systems, we have produced two generalised ddc packages; one a large application package for general use by customers who, unlike electric utilities, usually do their own application programming. The other is a special package for a specific customer who has us to do his programming for him and has given us considerable repeat business.

The former has not been particularly successful: either deficiencies in our package or lack of success of the concept (large ddc systems) might be the reason. I suspect the latter.

The "special" ddc package has just recently evolved to a state where it is becoming standard. We have realised the same kind of cost savings on recent systems as we did with SEER just before the final standardising effort.

Other Utility Application Packages – evolution

Just as SEER gradually standardised and the special ddc package is on its way to becoming standardised, other utility packages (for monitoring power transmission networks to provide system security and for economic dispatch and power generation control) are also evolving towards a standard.

For power transmission network monitoring (called SCADA) we have moved so far that we are now 'standardising' the package. Economic dispatch and power generation control packages will probably reach the same stage within one to two years. We are already realising savings through re-use of design and code. We are confident that true standardisation will lower our costs per system even further.

BICEPS – a successful supervisory control system

BICEPS is a large system for supervisory control

[where process variable setpoints are sent to analog controllers (or to a ddc system)] of continuous processes. It provides the engineer with forms which he fills out to describe measured variables and control loops; it also provides a simple language for programming his control algorithms. Its files are available to programs for further optimising and control calculations. (These programs are usually written in FORTRAN.) It was designed in cooperation with the same company that had previously been instrumental in the design of IBM's PROSPRO system for the same application. We have delivered it to several other customers as well as to the original customer.

In my opinion, its success may be credited to the fact that supervisory control still remains the main reason for using a computer on a continuous process, and that this application package fits supervisory control well. (Note, however, that one can expect application packages like this to increase system overhead. The laws of hardware/software trade-off apply.)

What next?

From the evidence of our own successes and failures it appears that application packages have saved us considerable money, while new languages have done little for us.

Actually, the dividing line between languages and application packages is hazy at best. Thus the elusive problem-oriented 'languages' that the industry has been looking for are in truth taking the form of application packages like those we have been discussing. Consider the important characteristics of both:

1. Machine independence.
2. A syntax for describing the problem solution that
   - fits the problem
   - the problem solver can use with ease.

Although the syntax used with an application package does not look like a language (i.e. it is not algorithmic), I maintain that the differences are only syntactical and that we should not worry about what form our successful 'language' takes. It is the benefit that counts.

Therefore, my conclusion is that for applications where standardisation is feasible, we at General Electric will continue to define application packages to lower programming costs.

Unfortunately, not all applications can be standardised, usually because the number of similar applications is too small to make it economical. For these we must continue to search for an algorithmic language better than FORTRAN. FORTRAN is merely a stopgap until a really good algorithmic language comes along to take its place – and there are candidates coming from all directions. At present we are looking to the various system implementation languages that are being invented.

Earlier the necessary attributes for success were stated:

1. Machine independence (as it was defined then).
2. A problem-oriented vocabulary.
3. Encouragement of good software design.

Good implementation languages allow one machine independence when one wants it, but let one get as close to the hardware as one wishes when one needs to. Although an implementation language cannot give 'the best' problem-oriented vocabulary (since it will be applied to diverse problems), it can at least give a better algorithmic vocabulary than FORTRAN, and a good implementation language will encourage good program design.

We are working on one now.

Lessons we have learned

To sum up, our nearly fifteen years' experience in searching for better programming tools has taught us the following:

1. An application package can be a real money saver where there is a large enough population of jobs for the package.
2. Application packages are best developed by the people experienced and actually involved in the application. Yet members of this group must be set aside and be dedicated to final standardisation independent of any one specific application.
3. Do not expect standards to be final. They are subject to evolution.
4. The ideas for any languages designed for use by application programmers should come from them. As a poor second choice, there must be a careful, early study conducted by the system programmers to discover the application programmers' real needs. This study serves two purposes – it increases the probability of technical success and it helps to 'sell' the tool. And make no mistake, the acceptance of a tool by those destined to use it is one of the most critical requirements for success!
5. The programmers who will use a tool must be thoroughly trained in both the application and the software itself.
6. FORTRAN with appropriate extensions can help if one is willing to make the necessary hardware/software tradeoffs. Yet it is not really the right algorithmic language.
7. None of the candidates (and there are many) for the 'right' algorithmic language has yet proved successful (that is, none has become a widespread standard).
8. Tools that help merely in the coding stage of programming are not worth the trouble.
9. Any language should have support so that its translation can occur at the application site. Otherwise much of the expected savings fail to

materialise.

10. Good software design vs. poor software design will make more difference in programming cost than the language selected (except where the language fits the application particularly well).

11. An application package must be modular; considerable effort should be made to separate the relatively stable elements from those that will likely change from job to job.

For FORTRAN see:

- American Standard FORTRAN
- American Standard Basic FORTRAN
  both available from:
  United States of America Standards Institute,
  10 East 40th Street,
  New York, N.Y. 10016,
  USA.
- 'Clarification of FORTRAN standards – initial progress', ACM Communications, V12(5), (May 1969).
- 'Clarification of FORTRAN standards – second report', ACM Communications, V14(12), (October 1971).

For the Purdue proposed extensions see:

- Minutes of the Fourth Workshop on Standardization of Industrial Computer Languages (pp. 71 ff) available from
  Purdue Laboratory for Applied Industrial Control,
  Purdue University,
  Lafayette, Indiana 47907,
  USA.
  There have been minor amendments which have not been published. This work will result in an Instrument Society of America Standard.
- Also obsolete in some of the details but adequate to give you a feel for the style of the extensions is KELLY, E.A., 'FORTRAN in process control: standardizing extensions', Instrumentation Technology (May 1970).

For information on some of our application packages see:

- SEER System (GEA 8465)
- BICEPS Summary Manual (GET 3559)
- GE-DDC Summary Manual (GET 3558)
  also available from General Electric in Phoenix.

For discussions on the functional requirements for Industrial Computer Systems see:

- Minutes of the Fifth Workshop on Standardization of Industrial Computer Languages, also available from the Purdue Laboratory for Applied Industrial Control (see above). This work has been summarized in CURTIS, R.L., 'Functional requirements for industrial computer systems', Instrumentation Technology (November 1971).

For further discussions on the applicability of FORTRAN for process control see:

- FROST, D.R., 'Fortran for process control', Instrumentation Technology (April 1969).

For a good survey of process-oriented languages see:

- PIKE, H.E., Jr., 'Process control software', January 1970 Proceedings of the IEEE, a special issue on computers in industrial process control. This same issue contains many good articles, including SCHOEFFLER, J.D., and TEMPLE, R.H., 'Real-time language for industrial process control' – (a proposed algorithmic language).

For General Electric's FORTRAN see:

- GE-PAC 4000 Process FORTRAN Language Manual (PCP 122)
- GE-PAC 4000 FORTRAN IV Language Manual (GET 6051)
  both available from:
  General Electric Company,
  Utility and Process Automation Products Dept.,
  Technical Publications,
  2255 W. Desert Cove Road,
  Phoenix, Ariz. 85029,
  USA.

Standard software versus high-level languages

R.C. HUTTY
GEC-Elliott, Borehamwood, England

(Presented by P. Heywood)

1. Introduction

The purpose of this paper is to assess the merits of using Standard Software as against a High-Level Language for application programming. Although some of the points made in this paper are applicable generally to all fields of computing, they are intended to refer mainly to the computing field of process and industrial control.

2. Definitions

As the terms 'standard software' and 'high-level language' are often misused, their meanings, as used in this paper, will be outlined.

Standard software

Standard software is that software which is designed, coded and tested once, but used, with little or no alteration, in more than one computer system. Standard software for a particular computer is normally developed by the computer manufacturer. Standard software falls into three categories:

1. Basic standard software.
2. Application standard software.
3. Problem-orientated languages.

1. Basic standard software refers to standard software which is designed for a particular computer range and is general-purpose enough to be used in any computer system which utilises one of the computers in the range. Basic standard software, e.g. executives and operating systems, is used in almost every computer system. Other basic standard software, e.g. mathematical routines, although general-purpose, is not necessarily used in all systems.

2. Application standard software, normally used with the support of basic standard software, is designed to satisfy the software requirements for a particular application area. Application standard software is developed only once but can be used in any computer system required to perform the same kind of function, e.g. a DDC system. Normally the only application programming work required is the initialisation of parameter values relevant to a particular system and a requirement specification of the parts (usually optional) of the software required for the system.

3. Problem-orientated languages are similar to high-level languages except that they are defined for a specific field of application, e.g. sequence control, and are machine-dependent. The software for a system is written using the statements of the language. However, the set of statements available is restricted and designed only for the narrow field of application. It is therefore not likely to be useful for any other field of application.

High level languages

High-level languages are used for one-off application programming. High-level languages are now replacing assembly languages for one-off application programming. The advantages and disadvantages of using high-level languages instead of assembly languages are not discussed here.

High-level languages, e.g. FORTRAN and ALGOL, and their statements are designed for general use. Real-time languages such as RTL are included in this category.

High-level languages are not wholly general but are designed for very much wider fields of application than the narrow field for which problem-orientated languages are suitable. For example, real-time languages are designed for all industrial control and real-time applications, whereas a sequence control language would only be suitable for relevant industrial sequence control applications.

3. Specific comparisons

For a comparison between the use of standard software and a high-level language, every aspect of a computer manufacturer's business, from selling to commissioning, must be considered.

Marketing

The marketing of systems is enhanced by the availability of extensive standard software, especially application standard software and problem-orientated languages. It is possible to provide meaningful publicity describing the particular item of standard software and its area of application. Potential customers are able to identify their company's requirements with software designed to solve their own particular problem. Standard software packages lend themselves to more illuminating advertising languages. High-level languages can be advertised, but their impact on potential customers is not as great.

Selling

Standard software makes the task of selling systems easier. Sales teams can aim at definite market areas rather than a general area. Salesmen are able to sell fixed price packages for some types of application standard software and are able to communicate with the potential customers via the standard software because it is designed for the customer's application and is readily familiar to and understood by the customer.

Tendering

Tendering of software systems currently involves a higher degree of risk than is normal in an engineering environment and therefore anything which helps to reduce this risk is obviously advantageous. Estimating the software content of a contract is more accurate when standard software is available than with high-level languages. The problem of estimating storage for a system is reduced if standard software is used because a larger amount of store is pre-defined by standard software than languages.

Not only storage estimates, but also the manpower required (which is often proportional to the storage requirements), are easier to assess because a standard software package inherently defines what is required to be done to make it suitable for a particular system. Even if it is not possible for the whole of the software system to be standard, at least that part which is standard is likely to be estimated accurately. Standard software enables the function of a software system to be specified precisely. The customer therefore knows exactly what he will receive and the programmer who will implement the system will know what to provide.

Programming

Standard software reduces the programming skill required. This makes the software less expensive, more controllable and reduces the risk. Standard software should provide an over-all software cost saving over the use of high-level languages. A programmer's job is reduced when using standard software because it has been predefined and pre-structured. Programming may be just a matter of supplying data to application standard software.

It is likely that non-computer people will be able to 'program' a system using application standard software or problem-orientated languages. Thus, engineers who specialise in the application field for which the standard software is designed can 'program' the software for his own application, thereby providing a system which does exactly what he wants it to do. Therefore, the problem created by the difference between what an engineer actually wants and what a programmer actually provides is removed.

Commissioning

Commissioning problems are reduced when standard software is used because there are fewer bugs in well tried and proven standard software than a first-time one-off program.

Documentation

Standard software reduces the amount of documentation to be produced for each contract. Standard documentation is a natural by-product of standard software. However, high-level languages provide a good degree of self-documentation.

4. General comparisons

There are many points other than those associated with each phase of a contract, which are general and worth noting.

Staff changes

Staff changes during a contract pose fewer problems when standard software is used, because the software system and its requirements are capable of being well defined initially and do not

usually change.

Machine dependence

Standard software is usually machine-dependent and, as such, is designed to be as efficient as possible, with respect to both storage and execution time. However, high level languages are usually machine-independent, so when one of these languages is required to be used on a particular computer the object code produced by the compiler may be unsuitably inefficient – this depends a great deal on the particular language-computer combination.

Suitability

It is very difficult to define a high-level language which is suitable for all applications within the extensive areas of application for which they are designed. For example, the inefficiencies produced by high-level languages are likely to be too large to be used as an alternative to basic standard software. However, efficiency is not always of prime importance.

Unfortunately, it is usually the case that standard software, especially application standard software, does not provide exactly what is required for a particular application, even though the application is considered to be in the area intended to be covered by the software. This usually happens when a customer's requirements extend outside the norm for which the software has been designed. Where the requirements are not essential then it must be pointed out to the customer the extra costs that are incurred by either 'bending' the standard software or programming the system in a high-level language, as a one-off system, as against using the standard software intact. Where the requirements are essential then there is no option but to 'bend' the standard software or to use a high-level language.

Quality

The suitability of standard software depends upon the quality of its design. A knowledge of the functional requirements of standard software is an important necessity before its design can begin; this will ensure that the software is likely to satisfy more applications than it would otherwise. The design of standard software should ensure that it can be easily made to fulfil the requirements of a particular application without redundancy. This is often currently achieved by using a modular approach in the design of the software.

Tools for a good system construction facility are necessary for modularly constructed software. It is better to 'bend' standard software than to use only a high-level language since at least some of the software will remain intact and as such be well proven. The bending should be performed, if possible, by the team who implemented the software, in order to reduce the risk of errors.

Development cost

Standard software development costs must be compared with the cost of producing a compiler for a high-level language. If any appreciable amount of standard software is developed then its cost will be greater than the cost of developing a compiler. However, it is usual to have available both standard software and a high-level language in order to completely cover all possible applications, and therefore development costs are required for both.

5. Conclusion

It would seem that there will always be a requirement for high-level languages and standard software because of the shortcomings of standard software. Although standard software, where it can be used, will usually be less expensive and involve less risk than a high-level language, it is only in simple applications that a system could be made up completely of standard software. Hence, in most systems, standard software requires to be either supplemented, or replaced, by a high-level language. It is important that a high-level language be compatible with the standard software.

As a general rule, wherever possible, standard software should be used in place of high-level languages.

Discussion

Q. In what level of programming language is your standard software written and specified?

A. It is written in assembly language, and specified in terms of the facilities it provides (and the implications thereof) in assembly language terms.

Q. How stable do you find your standards?

A. Packages are always modified, but only slightly.

Q. Are not modifications to standard software difficult to test?

A. Possibly so for an individual system, but not essentially worse than testing the original package.

Q. Suppose a piece of standard software (let us call it S) is slightly bent to form something we shall call S'. Then henceforth, is S or S' the new standard software?

A. The software S' may be eligible for standardisation, but is certainly not new standard software automatically.

Which programming languages for minicomputers in process control?

V. HAASE
UNICOMP, Blankenloch, W. Germany

A

Computer involvement in industrial and scientific process control shows a very significant increase in the application of very small computers. This is due to the fact that hardware costs are falling, whereas the cost of designing, installing and maintaining special-purpose systems is not becoming lower. This calls for general-purpose electronics in measurement and control: the minicomputer. This is the reason for the fact that this end of the computer spectrum is not populated so much by small brothers of medium- or large-scale real-time computer systems, but by special systems designed for – and in many cases by – the engineer used to working with special-purpose control hardware. This hardware is now replaced by a 4 to 16 K, 8- to 24-bit machine with storage cycles between 0.5 and 5 μsecs, in most cases capable of, but not necessarily equipped with, all kinds of conventional and process-control capabilities which medium-scale computers have as peripherals. The whole thing costs no more than a special-purpose hard-wired system.

Nevertheless, the lack of relationship between minis and larger systems often causes a software gap between minis and midi- resp. maxi computers. Hardware-mindedness of both designers and users, and the fact that minis are used in environments which are not frequently changed, have led to the development of special-purpose software (instead of the special-purpose hardware that existed before!) – nonmodular code including both operating system and application program functions.

Programming was (and is) done in some kind of assembly language. In the course of time, here and there, FORTRAN and BASIC compilers or interpreters were developed (non-real-time-applicable) and the idea of indirect programming came up (minis were used more in changing tasks, laboratories, test-stands, etc.). As the indirect method is of little use if a larger computer is not at hand (which is often the case if a program has to be changed in the field), it is necessary to think about programming languages and compilers to solve real-time problems with small computers.

B

There is an extensive literature on real-time languages, but little implementation experience. The minicomputer has attracted few papers. We shall therefore try to express the apparent contradiction between the (unpublished) feelings of computer scientists and the experience of users, namely, that both kinds (small and big) of real-time computers should not be programmed in the same way, at least not using the same higher level language.

There exist special types of (real-time) programming languages that are especially suited for compilation and execution on minicomputers. They will be explained and proved as follows.

If we look at *higher level problem-oriented special-purpose* programming languages that could be used on minicomputers in process control, we may distinguish several groups of languages:

1. Macro assemblers.
2. So-called 'low-level' languages (e.g. PL/360).
3. Real-time dialects or subsets of procedural languages (e.g. R-T-FORTRAN).
4. System writing languages (e.g. POLYP).
5. Special real-time-languages (INDAC, PEARL) which may also be based on other languages (e.g. PAS).
6. Problem-oriented languages (e.g. fill-in-the-blanks language).

On the other hand we may lay down criteria (functional requirements) for real-time languages:

a. Easy to use.
b. Machine-independent (at least, 'controllable' machine-dependent).
c. Effective.
d. Timing in 'programmer's hands'.

among others. We add for the mini:

e. Executable.
f. Compilable on a machine with less than 16K working store (no mass store).

Everybody knows that items b. and c. (especially in connection with e. and f.) are contradictory. We therefore have to find a compromise matching the lists of language types and requirements:

1. Macro assemblers are effective (c), pro-

vide timing-control (d) and cope with the mini-requirements (e, f); they are not machine-independent ($\bar{b}$) and not always easy to use ($\bar{a}$).

2. Low-level languages can be designed to fulfil criteria a., c., d., e. and f., but they are explicitly machine-dependent ($\bar{b}$) and therefore, for example, very useful for system writing.

3. Real-time dialects and subsets of FORTRAN and similar languages have shown that they are not able to solve both real-time *and* efficiency problems, i.e. inherent problems of batch processing languages ($\bar{c}$, $\bar{d}$). Perhaps this could be achieved, but not in mini-environments.

4. System-languages are sometimes claimed to be the solution for process control and any real-time problems. They are, in fact, but only in the hands of very clever programmers. However, ease of use ($\bar{a}$) and timing and tasking features are not ideal ($\bar{d}$).

5. Newly designed real-time languages would, of course, be the best solution (if they fulfil the functional requirements). The problem of generality versus efficiency *can* be solved for larger systems, including backing storage (INDAC, PEARL), but for minis this is not possible. The design concept has to be modified so that both assembly type and procedural type languages are combined in a new language. Should this lead to a modular structure of the language itself, implementation costs could also be reduced (PL/1 + PAS/1, and PROCESS-BASIC, to be introduced here).

6. Special problem-oriented languages will not be considered here, as they are no common solution for real-time problems (and in many cases also designed in a descriptive type: no timing = $\bar{d}$).

C

If we combine the efficiency of an assembly language, handy macros for handling real-time functions (timing, interrupt-handling), and the machine-independence and ease-of-use of BASIC or FORTRAN – the whole of which can be compiled and executed on a mini – we obtain the language we are searching for. Of course, these properties cannot be present simultaneously in every language element (under our marginal conditions), but we can try to have them in an additive structure. This seems to be sufficient in the majority of cases, e.g. one frequently needs a special algorithm for data reduction (that is present in a higher level language library), but not the I/O driver for a special device used in some other installation (that has to be programmed very efficiently).

My solution for a programming language for minicomputers in process control is a composite language, the elements of which come from:

1. A procedural language (BASIC) that is easy to learn, use and compile.
2. An assembly language, covering both machine instructions and macro-instructions (if it is sufficient, the normal assembler of your machine).

It is necessary that these two components can be mixed at the statement level – not only at the module (= linkage editor) level – and it is to be provided that user-named items (data and labels) can be defined in one and used in the other statement type.

C1

Since these ideas came up when a special project was considered (the usefulness of a process-oriented programming language to be implemented on the UNICOMP 201) some aspects of this machine will be mentioned.

UNICOMP 201 is a 20-bit machine with working storage expandable up to 32K. Since the limited instruction repertoire, as far as arithmetic is concerned, is balanced by a great variety of operation- and addressing-modes (system-user states; literal, working store, indirect, external store and 'execute' operands), real-time programming can be done quite nicely.

A convenient macro-assembler makes use of 'supervisor-call' instructions, causing the execution of a subroutine package present at runtime that can be implemented as a 'firmware' module (a fast read-only-memory). These macro instructions include both fixed point (multiple precision, too) and floating point arithmetic, and interrupt and I/O handling instructions. For the design of a language with the desired features, the algorithmic part must be replaced by a common procedural language. We have chosen BASIC for reasons of easy learnability and compilability.

To improve efficiency, BASIC had to be expanded by a data-type INTEGER (INDEX) that is used for counting and field addressing operations. As labels are represented by line numbers in BASIC also, the assembly type statements of Process-Basic are allowed to have and to refer to integer labels. The name-syntax of BASIC is not changed (for compatibility reasons), so that all names in assembly type statements that do not consist only of one letter and one number are unique for the assembly part of the program.

The three levels of Process-Basic statements comprise:

Level 1 (BASIC):

INPUT, READ, PRINT, DATA, RESTORE	Input and Output
GOTO, IF, FOR, NEXT, GOSUB, RETURN, STOP, END	Control
DEFFN, DIM, INDEX	Definitions
LET	Assignment

Level 12 (Macros):

OUT,OTN,INP,INN,OSM,ISM,OLS,ILS,KTL,PLA,SAZ:

I/O Operations done in parallel or sequentially, with or without formatting, using symbolic or direct addressing.

ISS,ISL,RIT:

Interrupt answer definition, enable, disable.

Multiple precision arithmetic, block transfers, text editing and test instructions are also handled on macro level.

Level 13 (machine instructions):

Operations			Modifications
BRI	(load)		
UMS	(store)		
ADD			literal
ADU			working store address
KON	(and)		indirect address
EOR		X	I/O address
ERH	(+1)		execute
VER	(-1)		supervisor call
SPR	(jump)		
UPS	(subrout.)		
SAN	jmp =0		
SUN	jmp ≠ 0		
SKU	jmp car =0		

All three levels of statements can be mixed line-wise as shown in the following example:

```
      .EQUAL. TIMER='006'      device definition control statements
      .EQUAL. ADC1 ='5F1'      on assembler level
100   FOR I=1 TO 10
110   BRI 30 ADC1              loop coded in BASIC intermixed with
120   UMS 22 I                 machine code
122   NEXT I
      .PSW. '0100'             address of an interrupt routine
200   BRI 30 TIMER
210   DIV 00 333               macro instruction, time converted to
220   UMS 20 T1                secs
      :
      :
```

C2

The actual software (as planned) will be based on the existing UNICOMP software; it will consist of operating system, compiler, linking loader, test and text-editing routines.

The operating system (2 to 4K) is modular in the sense that a list-driven event (interrupt)-handling routine, the device driving package and supervisor-call-interpretation macros can be tailored by the user according to his special purposes. The compile-time-OS can be different from the execution-time-OS (the former including more string handling functions, the latter real-time macros).

As for the compiler implementation for small machines, two philosophies are applicable. First, for very small core size two pass systems are to be preferred; the object code will be punched. Certain on-line text-editing is possible, but no incremental compilation. The object code may be both relocatable and linkable. Secondly, if one can afford to hold both dictionary and object code in core, one pass systems which allow incremental compilation can be preferred. The object code can be either executable at once or be used as input into a linkage editor.

The compilers will be very similar, as the structure is defined by the general syntax parser. Directory- and code-generating routines are used as subroutines by the general routine. Certain optimisation (in the field of space versus time-efficiency) can be performed in that the code generator has two possibilities which can be chosen by the programmer: more in-line or more subroutine-like code (e.g. loop-heads, if-statements). The advantages of BASIC can be seen in the general expression definition, the name syntax allowing a directly addressable directory and the very clean statement type recognition.

The type of language described here is thus a good solution, both for the implementer (good price/performance ratio) and the user of a small machine (handyness and flexibility).

1. KEMENY, KURTZ, 'BASIC'.
2. FRÖHLICH, 'SAMMI, assembly language for UNICOMP 201'.

Discussion

Q. Why must the compiler run on the 8K machine?

A. Frequently the computer is out in the field at some plant, and the users will wish to be able to do their own program development. Also they may not have access to bigger machines.

Q. Will not the advent of satellite computers alter this?

A. It will still be awkward if you have to go to another computer which is remote.

Q. Line by line interleaving of assembly language and high level language statements implies that the compiler cannot attempt inter-statement optimisation. Do you accept this reduction in efficiency?

A. Yes, we are much happier that the programmer is aware of this and has to deal with inter-statement optimisation explicitly (e.g. by use of INDEX) than that he leave this to the compiler. We allow a testing phase using an interpreter to give debugging facilities, with only tested parts of the program compiled.

The influence of the structure of high-level languages on the efficiency of object code

G. MUSSTOPF
Scientific Control Systems GmbH, Hamburg, W.Germany

1. Introduction

Programming languages are a means of communication between man and machine. For each of these languages an alphabet (set of symbols), a syntax and a set of semantic rules are defined. Applying the syntax and semantics statements for a computer can be constructed from the symbols of the alphabet. If a language consists of only a few simple elements which are strongly adapted to the hardware of a given computer, it is called a low-level language. Algorithms which are to be described using such languages must first be manually prepared, i.e. translated. Other languages have a substantially larger set of elements and rules. The representation of formulae in such languages, for example, is that used by the mathematician. These languages are called high-level languages. The translation into a form acceptable to the computer is carried out by a special computer program. It is noteworthy that no measure exists for the level of a programming language. It is also important to note that an arbitrary number of levels exists between the two extremes.

One of the aims of developing a program with the help of a language is to perform the computation defined by the program. The object program as end product of the development, whether from manual pretranslation or from automatic translation, can be of varying quality. This can be seen, for example, from the speed and/or size of the object program. The quality of an object program is influenced by the quality of the translator, the language and the source program.

People who program computers have the wish to express themselves at as high a level as possible, i.e. they want to use the jargon with which they are familiar. The advantages of being able to do this are:

1. Short training period.
2. Short program development time.
3. High legibility of the source text.
4. Low test and maintenance effort.
5. Later modification easily carried out.

The comparison of an object program, which has been generated automatically by a translator from a high-level language, with a functionally equivalent object program which has been optimally written in a low-level language shows a large difference in the quality, i.e. efficiency, of the object programs. The size of this factor with respect to speed or size depends also on the structure of the computer (in addition to the reasons given above).

For many applications this loss is rightly accepted. There exist, however, apart from system software, very many problems in the field of real-time applications which require that the object programs be of high quality. The assertions in the following sections should be considered in connection with these problems.

2. Programming languages and programs

The classical machine and assembler languages are considered here solely as objects of comparison for the quality of object programs. This section classifies languages such as PL360 [1], PS440 [2], CORAL66 [3], BLISS [4], PASCAL [5] and POLYP [6].

A program consists of two different kinds of elements. The first kind is used to describe the application (application elements), e.g. testing measurements for critical boundary values. The second kind is required owing to the 'general or specific structure of the computer' (EDP elements), e.g. the specification of the lengths of items of information or the use of address variables. The use of this kind of element helps the translator to produce a better object program.

These two classes of elements cannot be distinguished by the operators of operands they contain.

The minus sign in

```
TN - TO
```

is used to represent a temperature difference, whereas that in

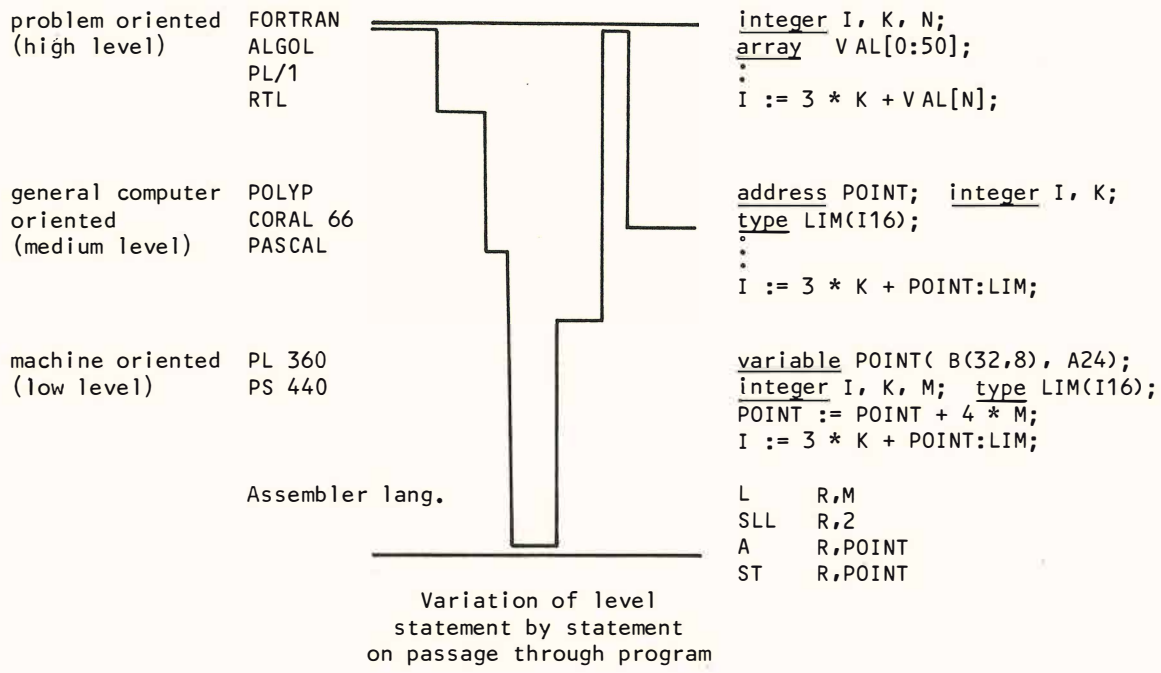


Fig. 1 Language levels

is used to modify an address constant. From this it can be seen that one must speak not only of the level of language but also of the level of a program. If address arithmetic is available in a language it does not mean in general that the programmer is forced to use it in his programs.

The EDP elements of a program can be divided into three subclasses:

1. Elements which refer to the general structure of computers (e.g. data overlay).
2. Elements which refer to the properties of a specific computer but which only influence the efficiency of the program (e.g. use of pointers as operand).
3. Elements which refer to the properties of a specific computer and which lead to errors (among other things) if the same source text is used when the program is transferred to another computer.

Programs which only contain applications elements and EDP elements of the first class are called machine-independent, those which in addition contain EDP elements of the second class are called conditionally machine-dependent and those containing EDP elements of the third class are called machine-dependent.

This classification proves useful when the languages themselves are analysed. The positioning of information in POLYP is taken as an example. The declaration:

```
variable AV(B(32,8),A24);
```

defines a variable with the name AV of type address and of length 24 bits (A24). The specification B(32,8) states that the address variable should have a bit address which is modulus 32 plus 8. For a word machine (smallest addressable storage unit is a word) with a 32-bit word the declaration above means that the variable is to be positioned into the rightmost 24 bits of a word. The declaration is also interpretable for a 16- or 8-bit machine.

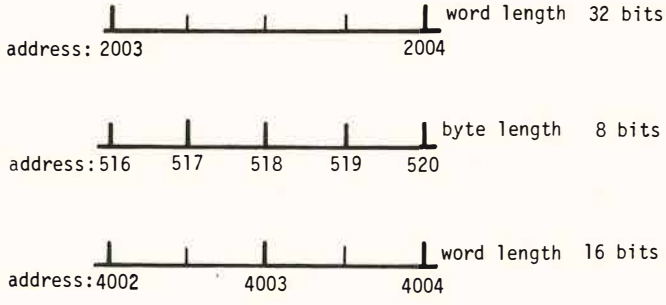


Fig. 2 Positioning

The language elements used in the declaration above are machine-independent since the syntactic and semantic definition makes no reference to a particular computer. Their occurrence in program elements leads to conditionally machine-dependent or even to machine-independent programs. The notion machine-independence is connected with the notion transferability of programs. Languages like FORTRAN, ALGOL 60 and COBOL were developed in order to allow for the exchange of programs regardless of the

computers being used. A program was transferable if it produced identical results when translated and run on two different computers. This condition is in general not sufficient for real-time applications. In addition, timing conditions (reaction time) must be fulfilled. This is, however, difficult to attain.

Another difficulty leads to similar wishes. Small and medium-sized real-time computers often have peripherals whose performance is too low and software which is not extensive enough for the development of medium and large program systems. For this reason many users are in favour of transferring the development work onto a large computer. This should be possible without having to use a simulator. It is mostly sufficient if the logic of the program modules can be tested on the large computer. Furthermore, it is not only possible to produce translators for such computers more cheaply, but they are also capable of accepting the full language.

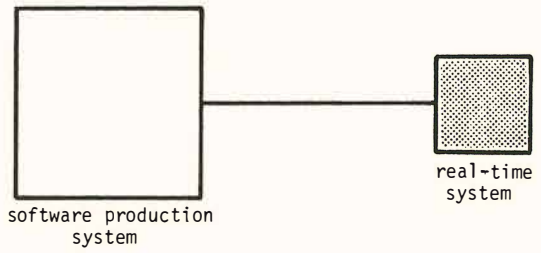


Fig. 3

If an object code of high quality is required it is necessary to make use of additional aids such as general macro processors.

3. Compilers

For the following considerations it is necessary to say something about the structure and the development of compilers. It is the job of a compiler to analyse a source text and translate it into a form which is directly or indirectly executable by the hardware. It is useful to differentiate between those problems which are associated with 'compile time' and those which are associated with 'run time'.

A compiler must be integrated into an operating system which, for example, takes over the management of data files. It is important that the compiler must be able to differentiate between various kinds of language elements (as opposed to the programmer). For example, the compiler must normally be able to distinguish between calls to user procedures, calls to standard procedures and calls to I/O procedures. One reason for this is that often different instruction sequences must be generated in each of the three cases. In order to make the distinction possible, the compiler must have a list of the names of all standard and I/O procedures (e.g. in its identifier list).

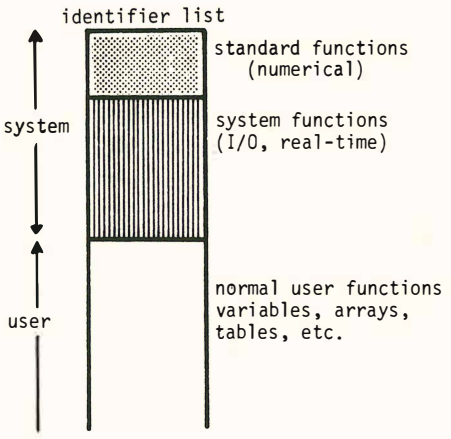


Fig. 4 Functions

The program which is generated by the compiler must normally run under the control of an operating system. This run-time operating system need not necessarily be identical to the compile time operating system. An existing operating system is usually used, whereby interface routines and extensions are necessary. This often requires an effort which is half the total effort of producing the compiler. Also, adaptation causes a reduction in the efficiency of the system. This problem occurs especially with real-time systems where the advantages of a universal system are discarded in favour of the higher efficiency of numerous small systems, each of which is tailored to the demands of a given application. Thus the necessity arises of adapting a language and its compiler to a given operating system. This should be possible without having to modify the compiler. It is possible to meet this demand if a distinction can be made at the definition level between the classes of procedure mentioned above [6]. In the compiler only the call interface need be defined. For each procedure class a different instruction sequence for the call is generated. The number and nature of the individual standard and I/O (or system) procedures need not be known to the compiler.

From the programmer's point of view, the main nucleus together with the procedures must represent a unit. By this method a system independence can be achieved which, at least for the next few years, exists in no process control programming language, compiler or operating system.

4. The reasons for bad object programs

It is often thought and claimed that the reason for

the low quality of object programs which have been generated from source programs written in high-level languages lies in the fact that the optimisation methods used in the compilers are not good enough. Much could certainly be done to improve these methods but optimising alone cannot solve all of the problems being discussed. For example, the problem of choosing between an optimisation which produces a fast object program and one which produces a short object program cannot be solved satisfactorily since the compiler sees the source program as a static unit and has little or no infromation about its dynamic nature. In the following considerations it will be assumed that the compilers considered all perform a sufficient degree of optimisation. Three typical properties of program elements have been chosen which exist in languages such as FORTRAN.

In the first example the definition of information (data) is analysed. Information types such as integer, real, boolean and string are usually available. The information length is normally predefined although alternatives such as double length are often allowed. In considering the information type boolean, which is most interesting in real-time applications, the question of the representation of a boolean value immediately arises. Should it be represented by a bit, a byte or maybe a short word? The information can be stored so as to optimise access time or so as to optimise storage requirement. The source text of the program must be analysed in order to determine which optimisation method should be used. Although it is relatively easy to calculate the storage requirement of a program, it is usually impossible to discover much about the effect of an access time optimisation, since the dynamic properties of the program cannot in general be determined. As already mentioned, it is also necessary, along with the type and length, to specify the positioning of an item of information. This is especially of interest in the case of tables, where the elements possess various types and lengths.

The second example is the sequential processing of information. Program cycles or loops (e.g. for statement), in which rows or columns of arrays or tables are processed, are mostly used for this purpose. The well known methods of optimising the organisation of the loop do not give the solution to the problem. More critical is the access to the information which is processed within the loop. The use of index registers relies on the fact that the loop controlled variable is incremented/decremented by a fixed constant value for each circuit of the loop, one condition for which is that its value is neither directly nor indirectly (side effects of procedures!) altered within the loop. If a static analysis of the source program shows even a remote possibility of this occurring, index registers cannot be used. The sequential processing of information outside loops is normally left unexamined. The problem can be solved in critical cases only by using address variables for which an address arithmetic is provided. An improvement can be achieved in POLYP [6], however, by using the vectorial statements which are an extension of the PL/1 array cross-section principle.

The last example is the call of a procedure with the transfer of parameters. Procedures are one of the most important aids in producing program systems and at the same time one of the most common causes of low quality in object programs. The reason can be found in the universality of the procedure concept. The parameter list in the object program consists, with hardly an exception, of a vector of pointers. The possibility of transferring values in registers or in special lists does not exist. The volume of instructions required for a procedure call and in the prolog and epilog of the procedure is out of proportion for short procedures.

The reason is that the construction and management of the parameter lists must be partly generated by the compiler and partly undertaken by run-time routines. It is completely sufficient for critical applications if an address variable as parameter is allowed. Thus the organisation and management of parameter lists becomes flexible and can be programmed to suit the situation. For example, the same parameter list can be used for different procedures, or the parameter list can consist of a mixture of pointers and values.

5 Conclusion

Considering the demand for language elements for the positioning of information, address variables, address arithmetic and address variables as parameters in procedure calls, it can be seen that the price to be paid for an improvement in the quality of object programs is the necessity for a better knowledge of the hardware properties of the computer(s) being programmed. It is not, however, required that the syntax and semantics of the assembler languages in question be known.

The problems mentioned are often solved by inserting assembler language text into the high-level source. This, however, does nothing to improve the legibility of the source program. The solution adapted by languages such as BLISS, PASCAL and POLYP is to provide language elements like those discussed above which can be used to improve the declarations and in some cases which can be used at critical positions in the dynamic parts of the program. The use of a second language is thus not necessary (Fig. 1). A more exact examination shows that a relatively small number of fixed or special positionings of information are required for a given computer or project. The generality of, and thus the amount to be written for, a given declaration can be

disturbing. This difficulty can be removed by introducing a general control (or macro) language. With the aid of a suitable library the required 'modifications' can be made to the source text.

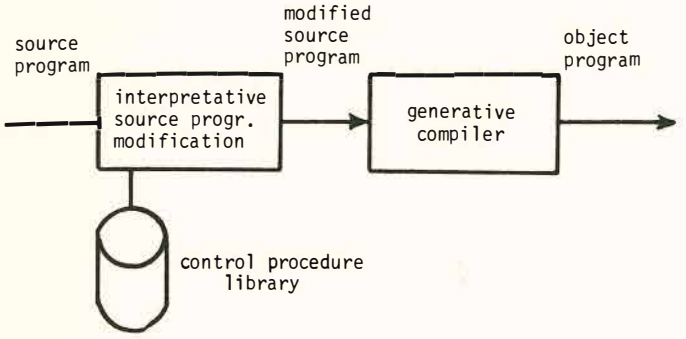


Fig. 5 General macro processor

This method can be used to great advantage, together with the system procedures mentioned in Section 3, to extend the language, without changing the compiler or operating system, with real-time oriented function calls. It is also possible, after the design phase of a project, to construct a control procedure library (Fig. 7) so that during the programming phase programs can be written without special knowledge of the computer and which can be translated into efficient object programs.

The methods described here also allow the development of real-time program systems using larger computers (software production system). It is only necessary that the required control procedure libraries be available.

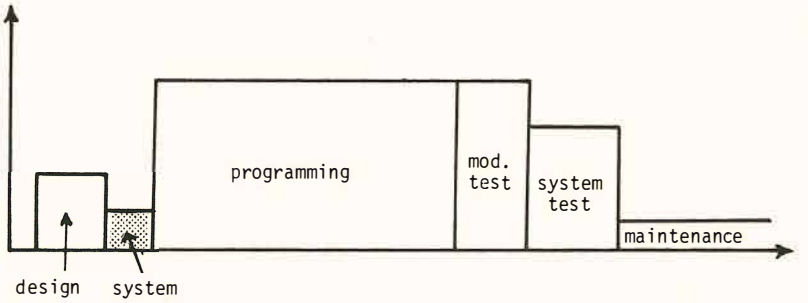


Fig. 6 Project phases

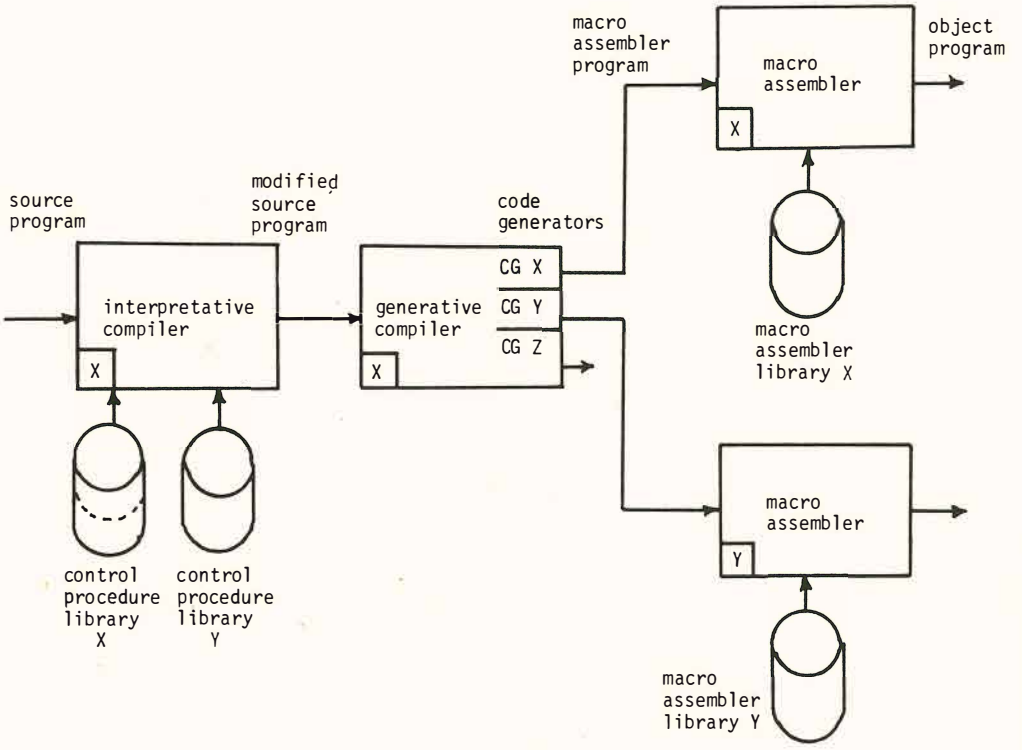


Fig. 7 Full compiler

1. WIRTH, N., 'PL 360, a programming language for the 360 computers', Journ. ACM, V15, pp. 37-74 (1968).
2. GOOS, G., LAGALLY, K., and SAPPER, G., 'PS440 – Eine niedere Programmiersprache', Bericht 7002 RZ der TH München.
3. Inter-establishment Committee for Computer Applications, 'Official definition of CORAL 66' (1970).
4. WULF, W.A., RUSSELL, D.B., and HABERMANN, A.N., 'BLISS: a language for systems programming', Comm. of the ACM, V14(12), pp. 780-790 (December 1971).
5. WIRTH, N., 'The programming language Pascal', Acta Informatica, V1, pp. 35-63 (1971).
6. MUSSTOPF, G., 'Definition der Programmiersprache POLYP' (noch unveröffentlicht).
7. MUSSTOPF, G., 'Sprachgesteuerte Modifikationen von Quell-programmen', GI-Tagung (März 1972).

Discussion

Q. In Fig. 1, what does the horizontal axis denote?

A. The stage during execution of a program – different statements may be at different levels.

Q. In Fig. 6 (project phases), how significant are the relative sizes?

A. It depends on the problem.

Q. Is a description of POLYP available?

A. Only in German (see reference [6]).

Efficiency of programming in higher-level languages

H. MITTENDORF
Karlsruhe, W. Germany

1. Idea of efficiency in computer programming

The increase in software costs as compared with the cost of the hardware devices themselves has forced both vendors and users of computers to become more and more conscious of the problem of effective programming. Difficulties then arise because many people use the following definitions side by side:

a. Effective (i.e. as short as possible) program-code.
b. Effective (i.e. as little as possible) reaction-time of the program.
c. Effective labour supply, if the reserve of man-power is pruned.

Therefore, the 'effectiveness' problem can be treated exactly only if we define a relative reference value for all participants. This shall be the 'sum of all costs' of a computer installation (hardware + software). Effectiveness in this sense means the minimisation of total costs by suitable combination of the hardware configuration with a particular programming technique.

2. Historical development of programming technique

The first computers that appeared on the industrial market possessed so little core storage that programs could only be produced directly in machine code (MC). The middle fifties saw the invention of symbolic programming with the Assembler Code (e.g. in 1956 the language SOAP = Symbolic Optimal Assembly Programming at the IBM's 650). This new method of symbolic addressing was very convenient at the time. The significant sign was the 1:1 – conversion between the assembler and the machine code. For many years most people believed that this conversion was the only method of producing optimal programs in the sense of minimal code length.

The development of FORTRAN, enforced by IBM, was the beginning of a programming technique independent of the special features of a particular machine. At the same time, the concept of FORTRAN was made in such a way that an effective MC could be generated out of the source code. Some 'risky' features of high-level languages remained forbidden (e.g. very generalized I/O procedures, bit- and byte-oriented data-types, general data structures and language features for general loop-modifications). It is only now that such extensions are being discussed in conjunction with the new languages PL/1 and ALGOL 68.

There has been a general feeling that the reference value man-power would remain sufficient for a long time, i.e., it would be possible to define 'effectiveness' or 'efficiency' independently of the 'market' of system programmers. However, it is becoming very clear that it is easy to build computers but very difficult to educate and train persons capable of using these computers.

3. Programming technique today

Current developments or developments in the immediate future are indicated by the following typical signs:

a) Prices of core store
A clear tendency to lower costs can be seen in this sector. This is a result of new production techniques (wire store, later on MOS store, in the future holographic mass memories) which will bring much more saving. At present, we must assume prices of 0.15 to 0.20 DM per bit, but between 1975 and 1978 we may expect 0.05 to 0.10 DM per bit for machines with a cycle time under 1 μsec. By 1980 at the latest we may expect that the price for storage capacity will fall by a further factor of 10 or more. Perhaps by then we shall have the cheap Tera-Bit memory for each computer (with 10^{12} to 10^{14} bit and a read/write cycle time of 20 nsec).

b) Capacity of internal stores
As a consequence of falling prices, the store capacity will generally increase (also for process computers). A new method of addressing by the paging mechanism will be of great importance in this context. This method will enable the addressing of very large data blocks in spite of small data units (e.g. 16-bit words). In the immediate future we will also have for process computers a maxi-

mum store capacity of 1 million words or more. As a consequence of these facts, we can also produce core-store-resident programs by higher level languages. If a demand paging mechanism is available, one can renounce the overlay principle for large programs.

c) Programs on background stores
We also believe that external stores will become considerably cheaper than internal stores, and therefore we shall need a reasonable roll-out technique. Because of the greater capacity of these internal stores it is possible for greater parts of the software belonging to this technique to be taken over by the operating system. Therefore the application programmer will be unburdened. As a consequence, in the future it will also be possible to produce large programming systems by means of higher level languages.

d) Use of high-level languages
After the introduction of FORTRAN, a few further high-level languages (ALGOL, COBOL, PL/1) were developed. These languages can facilitate the work of the application programmer, especially in mathematical (ALGOL) and economic (COBOL) problems.

Many groups throughout the world are working on the development of special real-time languages (e.g. Process-FORTRAN, PEARL, LTPL). The newest and most modern of these languages provide for very comfortable bit-, byte- and string-handling, and for definition of special data structures. Therefore many of the tasks of the application programmers are considerably reduced. In special cases, the time for the development and testing of programs will be lowered by a factor ten.

On the one hand we find, in general, considerable facilitation of the programming technique, but, on the other hand, there are many difficulties which arise from the increasing use of computers for solving more and more diverse problems. A few of these problems are listed below:

a) 'Mathematization' of the tasks
Many interesting problems (e.g. weather-forecasting by computer, computer-aided design, linear- and dynamic-programming) now involve greater mathematical effort. The same problems arise from the urgency for mathematical treatment of many problems in commercial and management information systems. Effective effort is possible only if the highly specialized men engaged in solving these problems are economically employed and relieved of routine tasks. For this, the use of higher level languages is very important.

b) Larger volume of data
Usually this is a result of more complex problems. The higher level languages (especially ALGOL and PL/1) can bring about an important improvement in data management.

c) Higher data-structures
Problems connected with the increasing use of interactive displays working with real-time computer systems can only be overcome if we introduce new data-types – the so-called 'structures' and 'pointer-variables'. With these new elements we can solve the problem of referencing from one data-list to another connected with it.

PL/1 (using the concept of 'pointers') and ALGOL 68 (with the much more useful reference concept) give very serviceable means to the programmer. Programs for storage and treatment of complicated pictures (by suitable data-structures) can be prepared and tested much faster by these means.

d) Coordination problems in real-time-programming
An additional complication in real-time-programming is the (temporal) coordination of simultaneously running programs. For this we have the Dijkstra-Semaphores, which allow coordination of single programs. 'Higher coordinating functions', giving a real facilitation to the user, will be developed. In every case, the possibilities of coordination in a higher-level language give a formal simplification to the user. This can increase the clarity of the problem and consequently prevent the appearance of errors, especially deadlocks, in programming. All this will contribute to a more effective programming technique.

However, improvements due to revised programming techniques will be nearly counterbalanced by the increase in the complexity of new tasks to be programmed. Nevertheless, very considerable progress has been made, as compared with programming techniques at the beginning of computer development: we now have excellent higher programming languages. With these we are able to solve very extensive tasks such as computer-aided design, numerical weather-forecasting, complicated real-time process control, and so on.

4. Consequences

Maximum efficiency, in the sense of minimisation of total cost, can be reached for a given problem and given hardware installation only by careful considerations of, on the one hand, higher programming effort and, on the other hand, larger core store. Unfortunately, there are a number of basic obstacles which are not precisely measur-

able but could hinder the application of higher level languages.

a) User prejudice against higher level languages
Frequently, this follows from deficient knowledge, insufficient training and erroneous generalisation. In many cases, the very abstract and sophisticated specification (here one thinks of the ALGOL 68 report) will hinder the user, because he believes that this language is as difficult as the understanding of the language specification. The well-trained system analyser does know that the contrary is true, but the unskilled application programmer (and the number of these is much greater!) will not use such a language. Therefore in this field of application, much explanation, clarification and education must be done in the future.

b) Alleged uselessness of higher level languages for solving real-time-problems
Unfortunately, objections of this kind are often justified. We must, however, think not only of existing means (e.g. supervisor calls in FORTRAN), but we must also take into consideration that latest developments (e.g. PEARL and LTPL). When compilers are available, e.g. for PEARL, and when the usefulness of this high-level language is demonstrated for practical problems, the most conservative user will adapt himself to programming in higher level languages.

c) Hardware limitations
Core stores *will* become cheaper, but their size cannot be raised as much as one would like (because of addressing problems). This is especially true for very small computers (e.g. computers for measurement instrumentation or other special purposes); in these cases, the application of higher level languages cannot as yet be recommended.

A greater part of these objections will have lost its importance in the course of the next few years. We will therefore neglect these objections in the following quantitative considerations (especially the limitation on core stores). We can then say that we are able to perform easier and faster programming for a given problem by using a high-level language. We can thus introduce the following assumptions for a program of length L_A (made in Assembler language):

1. By using a higher level language we need only a fraction $1/n$ of the time needed for programming in assembler language ($0 < 1/n \leqslant 1$, n real).
2. By the use of this high-level language the object program is lengthened statically by a factor $V_s > 1$. In the case of very long programs, we can reach $V_s \leqslant 1$, but this is only a theoretical limitation, because it is almost impossible to make very long programs in assembler language at all. Therefore we neglect, for the moment, the dependence of V_s on program length L_A.
3. The written program shall run or be installed f times.
4. We assume fixed software costs K_{SA} in assembler programming for given program length (K_{SC} is the value for higher level languages).
5. In the same manner we assume fixed hardware costs. For this we apply the price of the additional core store in DM per word.
6. If we further consider the program dynamically running, we see that the running time can become longer by a factor V_t.
7. Finally, if the program runs in a computing centre, we additionally have to consider

M_{KSPW} = leasing cost for one word (ca. 0.0015 DM/hr)

M_{ZE} = leasing cost for central processor unit (ca. 250 DM/hr), and

t_A = absolute running time of the program (written in assembler code).

In a quantitative analysis we must compare the savings of manpower costs, that is

$$A_S = (K_{SA} - K_{SC}) \cdot L_A = K_{SA} \cdot L_A \cdot (1 - 1/n) \quad , \qquad (1)$$

with the increased hardware costs or leasing costs. Therefore, we have to consider two different cases:

A. Increase of hardware costs

If a vendor desires to sell a computer f times together with a special application program, he must find the most economical solution for the whole system. For this, the vendor could write the program in a compiler language (to save time) and pay himself for the (perhaps greater) additional hardware cost. This additional cost is for f applications of the same program:

$$A_H = f(V_s - 1) \cdot K_H \cdot L_A \qquad (2)$$

Here is a numerical example (for 24-bit word, 5000 FW/NY):

K_{SA} = 20 DM/FW K_H = 5 DM/FW
n = 3 V_s = 1.7 f = 3.

Then from Eq. (1)

$20 \cdot (1 - 1/3) = 13.34$

and from Eq. (2)

$3 \cdot (1.7 - 1) \cdot 5 = 10.5.$

Therefore, A_S is greater than A_H and the use of a compiler language (higher level language) is

more effective.

B. Increase of leasing costs

When programs in higher level languages are to be used in the closed shop of a computing centre, the costs will be higher both statically (V_s) and dynamically (V_t) (as compared with assembler programs). These higher level language programs, therefore, will need more core-storage (in units of leasing costs: $M_{KSPW} \cdot L_A \cdot (V_t \cdot V_s - 1)$) and more computing time (given by $M_{ZE} \cdot (V_t - 1)$). The sum of these two effects must be multiplied by the absolute running time t_A and by the number of applications, f. Therefore, the increase in leasing costs is then given by

$$A_M = f \cdot t_A [L_A \cdot M_{KSPW} (V_t V_s - 1) + M_{ZE}(V_t - 1)] \quad (3)$$

The value of M_{KSPW} is very low. If we assume K_H = 5 DM/FW and 20,000 total working hours (time of amortization, about 5 years), we get $M_{KSPW} = 2.5 \cdot 10^{-4}$ DM/hr.

Now consider two numerical examples:

(1) K_{SA} = 20 DM/FW (see above)

L_A = 2,000 FW

n = 3 (therefore $1 - 1/n = 0.67$).

Then, from Eq.(1),

$A_s = 2 \cdot 10^3 \cdot 20 \cdot 0.67$,

$= \underline{\underline{27{,}000 \text{ DM}}}$

f = 20

t_A = 12 min = 0.2 hr $\qquad M_{KSPW} = 2.5 \cdot 10^{-4}$ DM/hr

V_t = 1.2

V_s = 1.7 $\qquad M_{ZE} = 2.5 \cdot 10^2$ DM/hr

Then, from Eq.(3),

$$A_M = 20 \cdot 0.2 [2 \cdot 10^3 \cdot 2.5 \cdot 10^{-4} \cdot (1.2 \cdot 1.7 - 1) + 2.5 \cdot 10^2 \cdot 0.2]$$

$$= 4 [0.5(2.04 - 1) + 0.5 \cdot 10^2]$$

$$= 4(0.52 + 50)$$

$$\approx \underline{\underline{200 \text{ DM}}}$$

In this example we can neglect the additional leasing costs A_M (the first term is nearly zero).

(2) If we now increase L_A to 200,000 FW, $V_t = V_s$ to 2, t_A to 1 hr, and f to 50, we reach a very unfavourable case.

We now get

$$A_M = 50[2 \cdot 10^5 \cdot 2.5 \cdot 10^{-4} (4 - 1) + 250 \cdot 1]$$

$$= 50(50 \cdot 3 + 250)$$

$$= 50 \cdot 400$$

$$= \underline{\underline{20{,}000 \text{ DM}}}$$

Now the additional leasing costs are high, but in this case everybody will use a compiler-language (just because of the length of the program!). Beyond that, the savings of software costs are, numerically (because $L_A = 2 \cdot 10^5$ FW),

$$A_S = \underline{\underline{2\,700\,000 \text{ DM}}}$$

Summarizing: Only if t_A and/or f are large enough is it worth programming by assembler language. Quantitatively, for this purpose with t_A = 1 hr, the value of f should be 5,000 to 10,000.

It should be emphasized that, besides the economical reasons, there are many other points of view calling for the use of a compiler language, e.g.:

1. Much better form of documentation.
2. Saving of know-how when a new computer family with the same software package is used.
3. Simple changeability of the program.

For the user-programmer and the system analyst:

4. Faster programming and relieving of routine tasks.

For the producer and vendor of the data processing system:

5. Relief of own software deliveries specific for certain application problems for the customer, provided that excellent compilers for an effective high-level language are available.

For the immediate future, assembler programming will remain for only two major software fields:

On the one hand, for frequently used system programs, as the operating system itself and the various compilers, and, on the other hand, for frequently needed application programs which must have a very short code. One can find this in interrupt programs with very short response times or in the field of very small computers (e.g. computers for measurement instrumentation with a 'software modular system').

Final observations

Effectiveness, that is, minimum cost of the whole installation, especially the installation of a process computer, can only be reached by careful

consideration of all cost components. For this, we will need more and more higher level languages. But as assembler programming loses its influence, we must develop more efficient higher level languages for real-time programming. This is a very wide field for computer scientists in industry and the various research centres for computer science. Extraordinary efforts will be needed for a break-through in this field. In spite of all the difficulties, we should attempt to reach a world-wide standardization of such a language. But we must expect that the user-programmer will carefully examine these new language features and really use such language if possible. Only if both sides – the application programmer and the software designer of the language – have enough openmindedness and understanding of the problems can we expect a solution of general satisfaction.

Discussion

C. I am sure the author of this paper does not feel he has the final answer on efficiency measures, but I applaud attempts like this. They are better than none – you at least have something to make individual judgements from – and also something to change and refine as your understanding improves.

Workshop on general considerations on real-time programming

Chairman: **Prof. Dr R. LAUBER**
Stuttgart, W. Germany

The discussion included realistic assessments of the gains from using high-level languages, and the overhead penalties they carry. The benefits of standardisation were overshadowed by the practical difficulties of achieving it.

Comparisons of high level and low level programming

S. We should recognise that even when the goal is to program in a high-level language, some fraction of the programming will be in low-level. In my experience (an operating system for an automated radar system), the proportion of high-level to low-level programming is about 80: 20. This is not simply to achieve efficiency – high-level programs can be tuned up. A second look at high-level programs often produces a highly efficient object code, although the initial version may have been poor. In one case the high-level language version had 50% less code than the original assembly language version (perhaps because the compiler-writer himself coded it!).

L. In a real-time data gathering system we found that 85% of the code could be written in Real-time FORTRAN, supported by an operating system written in assembler language. Coding in assembly language may be necessary to reduce the size of a program. We had a 12 K FORTRAN program, of which 2.5 K words were data, and converted it into assembler. This brought the size down to 8 K still including the 2.5 K words of data, so indicates a reduction of code itself to 60% of the high-level version.

P. At the Royal Radar Establishment we developed programs for a computer controlled radar in assembly language. Later when a compiler became available the same problem was reprogrammed in CORAL 66. The expansion in code resulting from the use of high-level language was 20%. The total program length was 10 K and less than 10% of this was written as code inserts to optimise speed at critical points.

K. TPL1 at Essex has been used to program a stored program controlled telephone exchange. It had substantial overheads: 90% on space in main store, and 35% on running time. It has been studied to find the reasons for the inefficiencies, and a second language TPL2 has been designed to remove them. This iteration of language design is very necessary but elapsed time must exist to permit it.

H. CONTRAN has been used on a process control application (cement kilns) in England. Unfortunately, it introduced considerable overheads in additional storage requirements and execution time. This suggests the question whether application demands still make it necessary to squeeze the best performance possible from current hardware. There still appears to be no spare power or capacity in computers to tolerate the reduction in efficiency introduced by a programming in a high-level language.

Standardisation

S. It is desirable that we have a generally accepted real-time language. However, this is unlikely to happen until it is promoted and supported by one or more large computer manufacturers. In addition it must not be too specific to a particular computer. What is needed is a virtual processor whose primitives are generally agreed and can be realized on most hardware. PL/1 is too close to SYSTEM/360.

D. ANSI committee X3J1.4 is working on PL/1 Standardisation, and has admitted some important principles:

1. It starts from the user's point of view, not the manufacturer's.
2. You cannot standardise what is not developed.
3. You must not neglect the Europeans..

Y. Is standardisation by ANSI more successful than by a major manufacturer?

D. ANSI cooperate with ECMA, so it should be helpful for both sides.

Y. There are plenty of languages all competing for use. A sort of natural selection applies between them.

R. ALGOL 60 did not come into existence in this way.

F. A standard language would gradually lead to a standard operating system and then to a standard computer. This is not allowed!

Y. CORAL 66 was defined by users, and has been used for many applications, including an operating system.

P. CORAL 66 compilers are available on a number of machines in the UK now. These compilers conform to the definition of the language published by the Stationery Office. They have been implemented in some cases by the computer manufacturing firm and in other cases by the RRE. The compiler typically needs a simple machine with 16 K of store. It is planned in the future to allow modular extensions to the facilities of CORAL 66 so that where more space is available a more powerful compiler can be used.

General comments

T. There is a converse effect of machine architecture on programming languages. For example:

1. The Burroughs B6700 is an ALGOL machine and there is no assembler language for it: extended ALGOL is the lowest language that can be used. Because of the addressing structure which has to be implemented, there are extra core memory accesses, which make it a slower machine than the equivalent 360.
2. The DEC system 10 (formerly the PDP-10) has two pairs of protection/relocation (base/limit) registers in addition to its index registers. This means that the code and data can be addressed using different base registers, and so one can easily product re-entrant code. Making use of this facility, it is possible to produce a real-time system in which the applications programs are re-entrant, even though written in COBOL!

C. Concerning high-level languages and standard software packages, both are needed in different circumstances.

We should distinguish at least two stages of evolution:

In stage 1, the engineer has just a rough idea what the process control computer has to do, and he is experimenting to determine the appropriate algorithm. For this stage he needs a flexible general-purpose high-level language.

In stage 2, problems are well settled, and satisfactory algorithms known. Software packages containing these algorithms can be provided for the user.

J. The level of language is not important if you have good people, but it is when your staff are not so good. They will meet deadlines, but the quality of the programs cannot be guaranteed.

BASIC DESIGN PRINCIPLES OF OPERATING SYSTEMS

Chairman:
Dr I.C. PYLE

Adaptive operating systems

H. WETTSTEIN
Institut für Informatik, Universität Karlsruhe, W. Germany

1. Introduction

In conventional data processing a great many computer users are disillusioned with the behaviour and performance of their equipment. Compilation times are too long, access routines too slow and processors are idle most of the time. Frustration has spread because "computers are not living up to their potential" [1]. It has, for example, come to the author's attention that in computing centres the policy is sometimes adopted not to use more than one work file in a data processing program. If the logical structure of a problem is such that more than one sequentially organized working area is needed, the recommendation is propagated to open a directly accessed file and to partition it into several sequentially organized subareas, the access mechanisms in this case being established by the programmer himself. This approach obviously involves a repetition of efforts already being invested in the implementation of the operating system itself. The reason for this is the apparent slowness of the operation of several sequential files concurrently.

It is the author's opinion that such and similar weaknesses of operating systems is due to the fact that those systems generally comprise far more capabilities than an individual installation really needs. Operating systems are intended to serve a great, and especially a heterogeneous, community. Therefore, it has been the manufacturers' policy to incorporate as many features as possible in order to increase the probability that the needs of an eventual customer can be fulfilled with the existing product. This is an acceptable policy from the manufacturers' point of view; it is drastically inefficient from the customer's point of view.

What can we do to overcome this conflict? An obvious solution, satisfying at least the user, would be to implement an operating system for every single environment. Of course, this conclusion can only be followed if the fabrication of an operating system can take place automatically. What is therefore needed is a kind of master or primary system including all possible features and capabilities from which special systems can be deduced and adapted to the requirements of individual installations. Such a master system we like to call an adaptive operating system.

2. Analogies in the area of control computers

The above-mentioned considerations led to a certain conclusion following an economic reasoning. However, within the area of control computers we come to a similar conclusion from a different point of view. Control equipment is usually sold in far less quantity than conventional processing equipment. In contrast, however, control equipment shows at least three additional limitations:

1. It is usually resource-bounded, which means that efficiency in software plays a much bigger rôle than anywhere else.
2. Performance with respect to switching times between processing or availability rates is a very important factor.
3. The operating environment varies greatly from one installation to another, but is constant within the same installation.

These restraints make it a necessity that almost every installation be equipped with its own specific operating system. Under those circumstances it is a matter of survival to produce the individual systems by automatic adaptation.

3. Previous approach

Of course, the idea of individual adaptation is not new. There are several examples of systems where the possib lity exists of selecting predetermined components from a collection of modules kept in a library. Such selection procedures have successfully been used to incorporate for instance:

1. The drivers for specific I/O-equipment.
2. Transition areas, work space, etc.
3. Certain macro functions.
4. Or even certain language processors.

A selection process is usually called 'system generation' and is generally executed when a new installation has been built up or when a configuration has changed. The system generator needs as an input a description of the configuration and a list of certain wishes. Sometimes the possible alternatives are all drawn on an order sheet as a flow diagram and the customer has to mark a

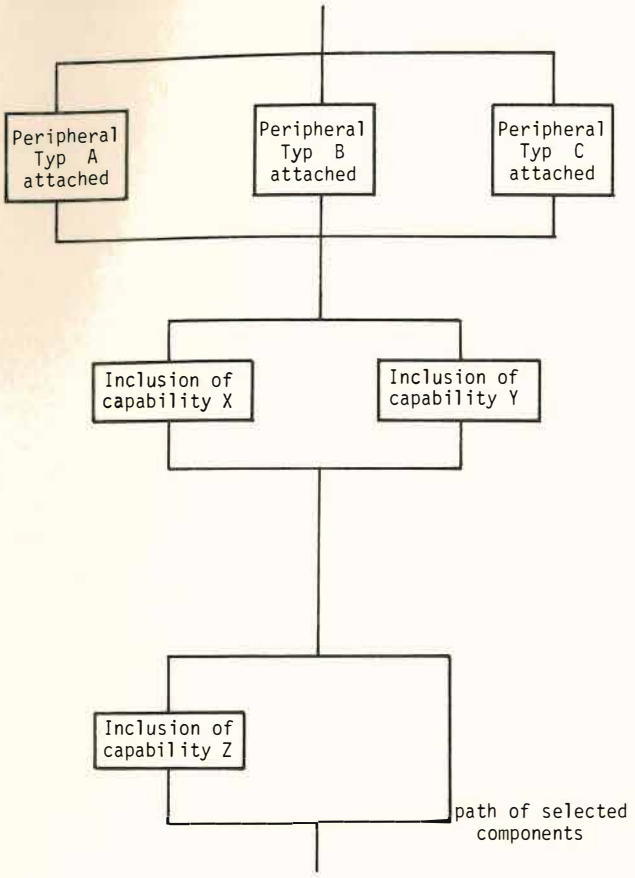


Fig.1 Selection of components for a system generation

single path through the various blocks (Fig. 1). By this means, selection of mutually exclusive components can easily be avoided. A good example of this philosophy is the system ordering and generation procedure of the SIEMENS 300 series. In contrast to what follows, we will call this kind of modification component adaptation.

4. New approach

Component adaptation cannot comply with the strict demands for optimality. Surely it is possible to exclude parts not needed in a system? But, nevertheless, the basic structure, the ground philosophy of the systems, remains the same. If, for instance, a system is designed to manage n processes concurrently, it is not an easy task to modify it for the management of $n+1$ processes. Or, if a system manages a certain list dynamically with capabilities for insertions and deletions, it is impossible to simplify this part to a static handling when the operating environment is definitely fixed. However, only modifications of this kind justify calling a master system a really flexible and powerful tool. Especially timing factors can effectively be influenced. We like to call this mechanism a structural adaptation. From there, we propagate the following strategy

1. Design an operating system as general as possible.
2. Incorporate as many features you can think of in all degrees of freedom.
3. Take the necessary precautions for reduction of the generic components to their simplest versions.

5. Formulation

Having recognized that our aim is not a component adaptation, it is clear that a system cannot be generated by binding together selected members from an object module library. We can imagine that differences between variations of components or references to other, possibly missing, components will very often show up only within short instruction sequences. Therefore, the object module is not the proper unit to keep an element of an adaptive operating system. What is needed is an implementation level where address references have not yet been resolved or, in other words, are still in symbolic form. Such a level is some form of programming language, in this case, preferably a special-purpose language for system implementation similar to PL 360 [2].

However, this higher-level language approach solves only one side of the affair. Nothing has yet been said about the composition or selection procedure itself. Again, the problem is how to know whether a certain sequence of statements should be incorporated in a target system or not. Obviously, a description from the outside via control statements is not feasible. Therefore, the relevant parts must themselves bear appropriate markings which tell the generating process the conditions for incorporation. Since those conditions may be very complex, it is apparently helpful to use again the means of a higher-level language. For instance, the conditional incorporation of a ⟨piece of program⟩ could be described by a sentence like

```
IF ⟨conditions⟩ THEN ⟨piece of program⟩ .
```

Later, it will be seen that the repeated incorporation of program pieces, possibly slightly but systematically changing from repetition to repetition, is also very useful. This can be expressed as

```
FOR ⟨repetition specification⟩ DO ⟨slightly
                    changing piece of program⟩
```

The ⟨condition⟩ and ⟨repetition specification⟩ in the examples above contain variables which actually control the composition mechanism. They can be manipulated. Especially by assigning values from outside (via READ-statements), the entire composition process is under external control. We finally see that all the approved language constructs can successfully be applied

to describe the composition scheme. Thus the composition scheme becomes itself a program, which, last but not least, can be executed. In fact, this execution represents the act of composing a target system. If the composition process is fairly complex, one can imagine the amount of variability among the target systems. Nevertheless, the composition process can be kept machine-independent to a reasonably high degree.

6. Composition compiler

What do we really mean by 'execution of the composition program'? Obviously, the ⟨pieces of program⟩ in the examples above are not intended to be executed because they constitute a part of the target system. We must adopt a new kind of interpretation of a statement's meaning as follows. In scanning the program text from left to right and obeying the rules laid down by this program, every time we come across a string of characters that do not belong to the composition scheme we attach them to an output stream. When the execution of the composition program comes to an end its output stream forms the generated target system.

Of course, the ⟨pieces of program⟩ consist of phrases similar to the composition language. This raises the problem of distinguishing between two linguistic levels.

Musstopf [3] has recently discussed this question. He proposed defining a different set of delimiters for the composition language. This could partially be done by labelling existing delimiter words, by replacing delimiters with other characters not hitherto used, or by newly introduced delimiter words. Examples are

Normal language	Composition language
FOR	CONTROL FOR
PROCEDURE	CONTROL PROCEDURE
;	$
+	ADD

The exchange of delimiters would also be accomplished by switching states. For instance

```
A   ADD   B  $
```

could be expressed as

```
CONTROL  (A + B;)
```

Of course, if the fraction of composition elements in the over-all picture prevails, it is easier to switch states at the beginning and end of normal passages, e.g.

```
IF  B  THEN  ATTACH ((piece of program));
```

This, indeed, reads much more smoothly because the delimiter ATTACH can be considered as a standard procedure call with ⟨piece of program⟩ as a parameter and with the function of outputting its parameter string. This is exactly our interpretation mentioned above.

In order not to make a language decision at this moment, we have chosen in the following examples to accentuate the composition parts by underlining.

7 Examples

The following two examples may be considered as two tiny cut-outs of a hopefully adaptive system.

Suppose we want to reserve space for a varying number of process control blocks (PCB), but only in the case when the processing environment is predetermined, that is when the list of the PCB is static. Each PCB consists of the following components (fields):

Contents	Data Type
- continuation address (CNTADR)	ADR
- ready indicator	BIT
- activation indicator (only if activation should take place individually)	BIT
- safe area for arithmetic register (REG) (only if process is interruptable at random places)	WORD

As we can see, the composition of a PCB is dependent on three different control parameters. We call them PRLIST (possible values: 'STATIC', 'DYNAMIC'), INTRPT (possible values: 'YES', 'NO'), and ACT (possible values: 'INDIVIDUAL', 'GLOBAL'). With these conventions we could formulate

```
DECLARE (I,N) INTEGER, (PRLIST,INTRPT,ACT) STRING;
READ (PRLIST,INTRPT,ACT);
IF PRLIST = 'STATIC' THEN
   BEGIN
   READ(N);
   FOR I := 1 STEP 1 UNTIL N DO
        DECLARE PCB | I
                    (CNTADR  ADR,
                     RDYIND  BIT
                     IF ACT = 'INDIVIDUAL' THEN ,
                                          ACTIND BIT;
                     IF INTRPT = 'YES' THEN  , REG
                                          WORD; ); ;
END;
```

If it happens that PRLIST = 'STATIC', INTRPT = 'NO', ACT = 'INDIVIDUAL', and N = 3, then the output of the above composition program would be:

```
DECLARE PCB1 (CNTADR ADR , RDYIND BIT , ACTIND BIT);
DECLARE PCB2 (CNTADR ADR , RDYIND BIT , ACTIND BIT);
DECLARE PCB3 (CNTADR ADR , RDYIND BIT , ACTIND BIT);
```

This example confirms also the former assumption that the fraction of composition text could outweigh the normal text.

The next example shows the construction of a simple scheduler. Starting from a current element of the PCB-list pointed to by a pointer P the scheduler is intended to search cyclically for the next process ready for execution. It should be adaptable to the structure and contents of the PCB-list, the latter being assumed to be either a dense list or a cyclically linked list with a pointer to the successor element in a field PCB.PTR of each element. The scheduler could read as follows:

```
NEXTPROCESS:
   IF PRLIST = 'STATIC' THEN BEGIN   P := P+1;
                                     IF P = N THEN
                                           P := Ø;
                             END
                       ELSE P : = PCB.PTR [P];  ;
   IF PCB.RDYIND [P] = Ø
      IF ACT = 'INDIVIDUAL' THEN OR PCB.ACTIND = Ø;
IF INTRPT = 'YES' THEN REGISTER : = PCB.REG [P];  ;
GOTO PCB.CNTADR[P];
```

Given the same situation as before the composition run generates

```
NEXTPROCESS:
   P := P + 1; IF P = 3 THEN  P := Ø;
   IF PCB.RDYIND[P]=Ø  OR PCB.ACTIND=Ø THEN
                                 GOTO NEXTPROCESS;
   GOTO PCB.CNTADR[P];
```

8. Further problems

Besides giving the impression of the formulation picture, the last example demonstrates the question of resolution of data structures. The composition scheme contains a statement for stepping on a list pointer, the stepping function being dependent upon the list structure. Since the allowance for various data structures will probably constitute a major part of an adaptive operating system, one can imagine the numerous places where similar primitive operations will occur. Structure-dependent differentiation, however, should not appear as a dominant part of the procedural division of the operating system. It will therefore be necessary to make available list-handling operators like 'step', 'insert', 'attach', 'delete', etc., being defined for any of the relevant data structures. The resolution of these operators would be deferred until the final translation phase.

1. MORTON, K.W., 'What the software engineer can do for the computer user', Advanced Course on Software Engineering, Munich (1972).
2. WIRTH, N., 'PL 360, a programming language for the 360 computers', Journ. ACM, V15, pp. 37-74 (1968).
3. MUSSTOPF, G., 'Sprachgesteuerte Modifikation von Quell-programmen', Zweite Fachtagung über Programmiersprachen der Gesellschaft für Informatik, Bericht Nr. 4 (1972).

Discussion

Q. This proposal seems to be an extension of the conditional assembly feature used for example in the PDP 10. The big problem is in debugging the resulting operating system. This can be very expensive, and in practice is not always done. The result is that when one generates a 'tailored' system it does not work. Might this apply to your method?

A. I am not particularly interested in the two language levels concerned, but in the security of the system components which this method encourages. The structure of the system gives built-in reliability.

Description of operating systems in terms of a virtual machine

H-P. ROST
Allgemeine Elektricitäts-Gesellschaft, AEG-Telefunken, W. Germany

After the impetuous course of compiler development in the first and second generation, after many roundabout ways of proving outstanding technological methods, the third generation was apparently successful in consolidating the procedures and by this the terms of computer knowledge. Well defined slogans were introduced to release unbiased information. Everybody knows today the meaning of multi-address instructions, byte-string manipulation, decimal arithmetic, context and index registers. With great effort, these terms and their usage were standardized on an international basis. The possibility of description and comparison of digital computer systems seems to be based on their so well-defined data.

But that is a delusion. The understanding ends at the border of the physical device. This means that only the comparison of physical devices is possible. But is what the manufacturer develops and what the user buys only a central processor? A central processor unit without standard software is saleable only with mini-computers. At the same time, the meaning of standard software and the description of it is far from standardized. Even an agreement as to the functions an operating system has to fulfil is not firmly settled.

From the user's viewpoint it is irrelevant whether a function he needs is realized in hardware or in software, provided that it solves his problem quantitatively. I shall discuss this matter by taking floating-point arithmetic as an example. The following solutions are possible:

Hardware processor
Micro programmed processor
Interpretation of non-defined operations code
Assembler-generated built-in-macro
Assembler-generated library procedure.

Each solution owns by equal quality a different working speed and need of working memory; there are differences in quantity. The system (hardware and software) is in all these cases able to use floating-point arithmetic, although the physical central processor in some cases is inefficient to do so.

Naturally this train of thoughts is meaningful only in the case where the user is able to perceive not only the qualitative requirement of a computer function, but also the quantitative exigency. The latter is not true for technical-scientific or commercial computer systems, because not all the programs to be executed are known from the first. But in the greater part of the process computer installations the situation is better. Here the structure of the program system is given by the technology of the plant and will not change for a long time. In our example the user of a process computer knows whether his problems need a great amount of floating-point arithmetic or not.

These thoughts lead us to the following statement. A computer system behaves like a central processor which is able to execute certain functions. The addition of software to a physical central processor manifests itself like using an imagined central processor unit with extended functions. This imagined central processor unit I call a virtual computer. Its features can be defined and described in hardware terms.

This interpretation of terms seems particularly serious if the physical description alone would give a false impression. Examples of this are often found in descriptions of interrupt units. The interrupt unit often consists of modules for the collection of events, which generate an interrupt for the module as a whole and prepare a status word identifying the source of the interrupt. The reaction time for the representative interrupt (shown in the design paper on the physical central processor) is unimportant as long as the time necessary to read in and interpret the status word is unknown. The latter is done by a program, the so-called interrupt handler. But its consumption of time is not shown in the normal data sheets of a computer. For a virtual computer the behaviour of the interrupt unit can be described in the following way:

For each interrupt input it is definable whether it works on a channel with low or high speed. The performance of the channel consists in identifying the interrupt input, interrupting the running program and starting a code routine attached to the interrupt input. Each interrupt is followed by a time interval in which other interrupts are only marked down. These are not served in the order of input sequence, but in a fixed priority order.

In most cases, the above code routines handle peripheral devices and are called drivers. Usually peripheral devices are coupled with the central processor by a hierarchy of hardware units named data channel, channel multiplexor or device controller. The physical description of these hardware units shows whether the device will work in character or block mode, whether data or command chaining is allowed, whether control operations are realized by special commands or control characters.

If the driver is planned to output a new character after each interrupt, to interpret certain control characters by special actions, the device works for a user behind the intersection line of the driver equivalent to a device with a very comfortable controller. This being part of the manufacturer's standard software, the user is not in a position to differentiate between hardware and software solutions.

In the virtual description, this situation is stated by the fact that all peripheral devices are operated by the same I/O programs. Following this outline, the virtual computer will translate between external and internal codes and bridge the difference between the relatively slow peripheral devices and the central processor by external buffering. The question is: where is this procedure to be terminated?

I have shown that all required functions not realized in the physical central processor are attainable by software imbedded in the standard operating system. On the other hand, all functions generated by supplemental software are describable in terms of a virtual computer. The advanced standardization in the hardware field shows that it is now preferable to describe structures and attributes of operating systems in this way.

Let us now try such a description for a central processor with a basic monitor including an I/O system as an example. (These undefined terms show that no two people will agree on their usage, and defining them would lead to other undefined terms, and so on.)

In the example, the virtual central processor serves a great number of devices connected with it, some being peripherals like teletype or tape punch, some process-elements like digital output or analogue input, some background memories, serial like magnetic tape or random like drum and disc, and most being programs. For the programs running in this virtual processor all these devices act equally. This means that they are started with the same virtual I/O commands. This feature is very valuable in industrial process systems because you can replace, in certain periods of time, one peripheral by another, by a bulk memory or by another program. Structuring and restructuring of the program system will be made easy by this.

When executing the I/O command this virtual processor unburdens the source program of the parameters coupled with every I/O action by storing them in itself. In hardware terms, the control parameters are transferred to the I/O processor of the specified channel. This means that the source program is with this action ready for working again. This has great influence on the design of program systems for stochastic processes, where noone is able to predict the next program start.

It will be clear that this device-independence of I/O commands is a strong tool for testing programs, structuring program systems and widening the field of application by changing devices without modifying the programs.

The virtual devices of the virtual processor work off some multiplexors: the peripheral devices on the I/O multiplexor, the background memories on the memory multiplexor and the programs on a (virtual) multiplexor of the arithmetic and control unit. The last works in so-called burst mode overlayed by a priority strategy. (This is the sufficient hardware description of a multiprogramming operating system.)

A virtual computer for the control of a stochastic processor should have the facility of command chaining of I/O operations for all devices. Interpreting this for the programs, we speak of parameter buffering for many calls of one program without confusing the order of sequence.

In this description, no special position is held for the interrupts. In the virtual computer, interrupts terminate in calls for programs, undistinguishable syntactically, but only semantically from calls from other programs. Therefore it is clear that all interrupts can be simulated in test or in case of failure by certain programs.

The AU multiplexor not only switches from one program to another, but also switches the surroundings of the program. This means that every device is able to communicate with other devices only by means of the virtual I/O statements. For every program-type device therefore it seems like being the only program on an own central processor in an environment of devices, some of them also being programs.

This feature of the virtual computer means resource allocation strategies, a swapping mechanism and virtual addressing.

The above account should give some idea of the hardware description of a virtual processor. Once the qualitative aspects are clarified, it will be an easy task to give quantitative answers in the same manner. It seems that this method of description, definition and comparison of different process computer systems is much easier than the definition in special software terms. Last but not least, with new developments, such as micro-programming on the one hand, and mult-minis on the other, the hardware borders have become so soft that only unified descriptions of systems as a whole, including the above concept of the virtual computer, are able to provide a

base for standardization and comparison.

Discussion

Q. How do you deal with drivers for non-standard devices?

A. For every device including non-standard ones, the driver has to implement the appropriate virtual I/O statements.

Interrupt handling in real-time control systems

R. BAUMANN
Technical University of Munich, W. Germany

Introduction

Typical Third Generation Computing Systems are time sharing systems, teleprocessing systems, information and service systems, interactive systems and computer networks, as well as real-time control systems. They are designed to provide the user with a wide range of problem-solving facilities on a low-cost basis through sharing resources and information. The programmer especially wants to find an efficient software environment for development, debugging and execution of his programs.

Owing to the complexity of these systems, general software must be designed very carefully, following sound engineering principles similar to those of hardware design. The structure of the systems considered is determined primarily by functional components independently of the realisation which can actually be given either by hardware or software units.

Since properties like concurrency of programs, automatic resource allocation, sharing of resources, multi-programming, multi-tasking and multi-processing are common to third generation computing systems, general software for these systems will also have much in common. We may therefore ask for components typical of real-time control systems only, which will not be present in other computing systems.

A real-time control system is determined by its ability to synchronise with a technical process. It must therefore contain facilities to:

1. Accept data from, and deliver data to, the technical process at given real-time intervals.
2. Act immediately in response to foreseen or unforeseen events in the technical process.

From the point of view of the computing system, this implies that there must be a functional unit, which can (at least temporarily) interrupt processor activities and alter the programmed sequence of processes due to external events. Emphasis is on the adjective 'external', since interrupts caused by internal events occur in every computing system which realises concurrency of programs, sharing of resources or even some modest fail-soft mechanism.

We therefore conclude that mainly external interrupt handling facilities distinguish real-time control systems from other third generation computing systems. A typical component of a real-time control system is one that is dedicated to handling external interrupt signals. Nevertheless, for practical reasons, external interrupt handling is usually embedded in general interrupt handling facilities.

Following this guideline, a working group of Unterausschuss 'Programmierung' of VDE /VDI Fachausschuss 'Prozessrechner zum Messen, Steuern, Regeln' began to describe hardware and software features of an interrupt input device (Appendix). This effort can also be considered as preparatory work for future standardisation within the scope of Arbeitskreis C of Fachnormenausschuss Informationsverarbeitung (FNI) im Deutschen Normenausschuss (DNA).

1. Classification of interrupts

Owing to their urgency, interrupts can be classified according to their origin as follows:

1. Emergencies (e.g. power supply failure).
2. Traps (e.g. every kind of error occurring in the central processing unit).
3. Peripheral interrupts (messages from peripheral devices).
4. External interrupts (messages from outside).

Emergency interrupts are always served instantly in order to save as much information as possible in case of a breakdown. Traps should be checked before any other interrupt is served (except for emergencies) in order to ensure correct functioning of the computing system itself. Hence interrupts of class (1) and (2) may have immediate access to the central processing unit, while those of class (3) and (4) are handled by a distinct interrupt device. These should only compete with other activities of the computing system on the same level of urgency.

2. Interrupt device

2.1 The interrupt device has to perform the following functions:

a. Accept the interrupt signals.

b. Attach and evaluate interrupt priorities.
c. Assign start addresses of interrupt response programs.

Interrupt signals enter the interrupt device by interrupt entries, which can consist of three flip-flops, the outer mask element, the interrupt location (holding the signal) and the inner mask element (see Fig. 1, Appendix).

2.2 The set of interrupt entries can be structured in different ways by forming subsets according to common properties such as:

a. Equal interrupt priority.
b. Common mask elements (groupwise locking).
c. Common read/write instructions for the mask elements.
d. Common response program.

The subsets defined in this way are usually called groups, except for the case of equal interrupt priority, where the term 'level' is used. Obviously different structures can be realised simultaneously in a given interrupt device (Figs 3 and 4, Appendix). An interrupt entry can be:

a. Armed or disarmed by setting the outer mask element.
b. Enabled or disabled by setting the inner mask element.

These functions can also be performed groupwise.

2.3 Since it must be assumed that, in reality, several interrupt signals will compete for execution simultaneously, an interrupt priority scheme is needed to determine the order of execution.

When the decision as to whether or not to interrupt the processor's activity is to be influenced by the priority scheme, priorities are to be attached not only to interrupt entries but also to the central processor, according to the piece of program occupying it. In that case the priority scheme is much stronger, since not only the single selection of an interrupt request, but also the interruptibility of the affiliated response program is within its scope. Here is the chance to weld together external interrupt handling with other activities of the computing system, combining the priority-allocation scheme for internal system activities and external interrupt handling.

Concerning interrupt priority we have to differentiate between:

a. Fixed priority (attached to every interrupt entry or interrupt level by hardware).
b. Variable priority (attached by software).
c. Subpriority (due to a fixed sequence of serving interrupt entries of a given interrupt level).

The interrupt device has to put in order the interrupt signals standing at enabled entries, according to their priority, and has to offer the interrupt request of highest priority for execution.

3. Interrupt execution

3.1 The interrupt is actually effected if the following conditions are fulfilled on the processor side:

a. The state of main scheduling or supervisory program controlling the processor's activity must permit the interrupt (e.g. according to a comparison between processor priority and interrupt priority).
b. The piece of program occupying the processor must be in an interruptible state.
c. The instruction the processor is working on must be interruptible.

In multi-processor systems, interrupt requests can be dispatched to all equal processors on the basis of a comparison between highest interrupt priority and lowest processor priority. Such a schedule makes the system more flexible than dedicating a single processor to interrupt handling.

3.2 In the case of normal competition [interrupts belonging to class (3) or (4), see Section 1], it is desirable that the interrupted program can later re-enter the processor without avoidable loss of time and information. This implies that, before releasing the processor, the contents of all registers and other locations relevant for continuation are saved by putting them in store or by just switching to another register block. Admittedly, in the latter case, the processor has to provide for as many register blocks as different programs are to be served simultaneously by it.

In several process control systems there is a one-to-one correspondence of interrupt levels to register blocks. In other systems the register blocks are scheduled by the supervisory program.

4. Characteristic time intervals

In order to judge the capacitance of a real-time control system and to compare the efficiency of its performance, the user and the programmer need information on time spans consumed in interrupt handling. For reasons of correct comparison, time intervals and related activities should be defined clearly.

When tracing an interrupt request from the origin of the interrupt signal to the completion of the responding action, the following instants are of interest (Fig. 5, Appendix):

P_0 – the interrupt signal enters the interrupt device,

P_1 – the interrupt request is announced to the processor,
P_2 – the interrupt request is accepted by the processor for execution,
P_3 – the processor starts to execute the first instruction of the specific response program,
P_4 – the processor finishes the last instruction of the specific response program,
P_5 – the processor starts to execute the first instruction of the next scheduled program.

For the intervals T_i defined by

T_i = time between P_{i-1} and P_i

the following denotations are recommended:

T_1 – transition time
T_2 – latency time
T_3 – recognition time
T_4 – execution time
T_5 – return time.

Additional recommendations are:

$T_1 + T_2$	– waiting time
$T_1 + T_2 + T_3$	– reaction time
$T_3 + T_4 + T_5$	– interrupt time
$T_1 + T_2 + T_3 + T_4$	– response time
$T_3 + T_5$	– organisational time.

It must be admitted that a considerable amount of philosophy is hidden behind these denotations. Nevertheless, they may be helpful for comparing the interrupt handling facilities of different real-time control systems.

Conclusion

Real-time control systems can be expected to provide for interrupt handling facilities on different stages. Hard interrupts will have immediate processor access while soft interrupt requests are handled via the operating system, competing with other activities on a priority basis.

APPENDIX

Editors note: This appendix has not been translated since it is concerned with precise terminology.

Entwurf für eine einheitliche Beschreibung von Unterbrechungsvorgängen

1. Übersicht
2. Unterbrechungseingänge
3. Zustandsbezeichnungen von Unterbrechungsgängen
4. Unterbrechungsgruppen
5. Auswertung der Priorität der Unterbrechungssignale
6. Definition charakteristischer Zeitintervalle

ausgearbeitet von der Arbeitsgruppe "Interruptwerke" des Unterausschusses "Programmiertechnik" des VDI/VDE-Ausschusses "Prozessrechner zum Messen, Steuern, Regeln".

Mitarbeiter:

Prof. Dr R. Baumann, Mathematisches Institut der Technischen Universität München
Dipl-Phys. J. Brandes, Institut für Informatik, Universität Karlsruhe
Dr G. Heller, BASF Ludwigshafen
Dipl-Ing. G. Hepke, IDT der GfK Karlsruhe
Dr P. Höhne, Siemens ZL Munchen
Dipl-Phys. E. Huber, MBB Ottobrunn
Dr M. Prasser, AEG-Telefunken Konstanz
Dipl-Phys. W. Rüb, Mathematisches Institut der Technischen Universität München.

1. Übersicht

Die Verarbeitung von Unterbrechungssignalen in einem Rechnersystem wird durch eine Unterbrechungseinheit veranlasst.

Der Unterbrechungseinheit fallen dabei folgende Hauptfunktionen zu:

1. Annahme der Unterbrechungssignale
2. Zuordnung und Auswertung der Unterbrechungspriorität, und
3. Zuordnung von Anfangsadressen der Antwortprogramme.

Dabei bleibt offen, ob die einzelnen Funktionen ganz oder teilweise durch Hardware oder Software realisiert sind.

Es wird davon ausgegangen, dass die Unterbrechnungsanforderung durch ein binäre Unterbrechungssignal gestellt wird. Die Übertragung zusätzlicher Information über die Unterbrechungsursache auf anderen Datenwegen wird hier nicht betrachtet.

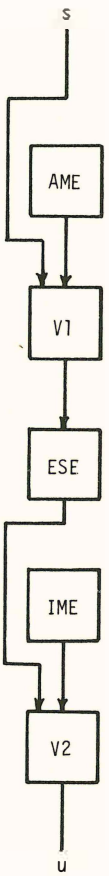
Zur Beurteilung des Verhaltens von Rechnersystemen bei Unterbrechungsanforderungen muss auch der zeitliche Ablauf bei der Bearbeitung angegeben werden.

2. Unterbrechungseingänge

Ein Unterbrechungseingang hat die Aufgabe, ein ankommendes Unterbrechungssignal wahrzunehmen, zu speichern und gegebenenfalls gegenüber dem Rechner abzusperren.

Ein Unterbrechungseingang kann aus Aussenmaskenelement, Eingangsspeicherelement sowie

Innenmaskenelement bestehen (s. Bild 1). Jedes dieser Elemente kann den Wert L oder O speichern.



Abkürzungen:

AME Aussenmasken-Element
ESE Eingangsspeicher-Element
IME Innenmasken-Element
s,u Unterbrechungs-Signale
V1 Verknüpfungsglied 1
V2 Verknüpfungsglied 2

Bild 1 Unterbrechungseingang

Aussenmaskenelement und Eingangsspeicherelement sowie Eingangsspeicherelement und Innenmaskenelement sind je durch ein Verknüpfungsglied verbunden. Bei geeigneter Besetzung des Aussenmaskenelements verhindert das Verknüpfungsglied V1 den Durchgang eines ankommenden Signals zum Eingangsspeicherelement, bei geeigneter Besetzung des Innenmaskenelements verhindert das Verknüpfungsglied V2 die Weitergabe des im Eingangsspeicherelement angekommenen Signals.

Aussenmasken- und Innenmaskenelement werden nur vom Rechner (DIN 44300) her gesetzt und gelöscht. Das Eingangsspeicherelement kann von einer externen Signalquelle (bei geeigneter Besetzung des Aussenmaskenelements) oder vom Rechner her gesetzt, jedoch im allgemeinen nur vom Rechner her gelöscht werden.

3. Zustandsbezeichnungen von Unterbrechungseingängen

Für die Zustände der Elemente eines Unterbrechungseingangs werden folgende Bezeichnungen vorgeschlagen:

TABLE 1

	Bezeichnung
Aussenmaskenelement	aussensperrend
	nicht aussensperrend
Innenmaskenelement	innensperrend
	nicht innensperrend
Eingangspeicher-element	gesetzt
	nicht gesetzt

TABLE 2

Aussenmasken-element	Eingangs-speicherelement	Innenmasken-element	Zustand des Unterbrechungs-eingangs
aussensperrend	*	*	aussengesperrt
*	*	innensperrend	innengesperrt
nicht aussensperrend	nicht gesetzt	*	bereit
*	gesetzt	*	belegt

*bedeutet eine beliebige Besetzung des betreffenden Elements

Daruber hinaus werden für die 8 verschiedenen Besetzungsmöglichkeiten eines Unterbrechungseingangs 4 Zustandsbezeichnungen angegeben. Alle weiteren Zustände lassen sich als Kombination dieser Bezeichnungen beschreiben.

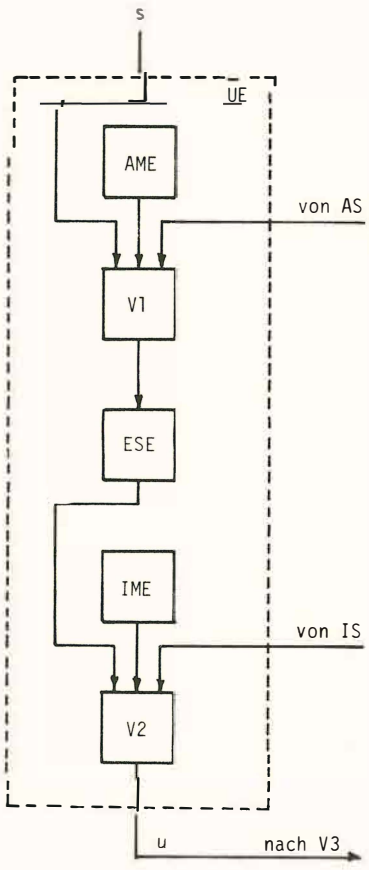
4. Unterbrechungsgruppen

Eine Unterbrechungsgruppe ist ein Satz von Unterbrechungseingängen, die nach einer gemeinsamen Eigenschaft zusammengefasst sind. Solche Eigenschaften können z.B. sein:

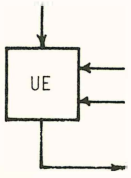
a) Gleiche Unterbrechungspriorität der Eingänge

Jedem Unterbrechungseingang ist eine Unterbrechungspriorität p zugeordnet (vergleiche 5).

a) Schaltung:



b) Schaltsymbol:

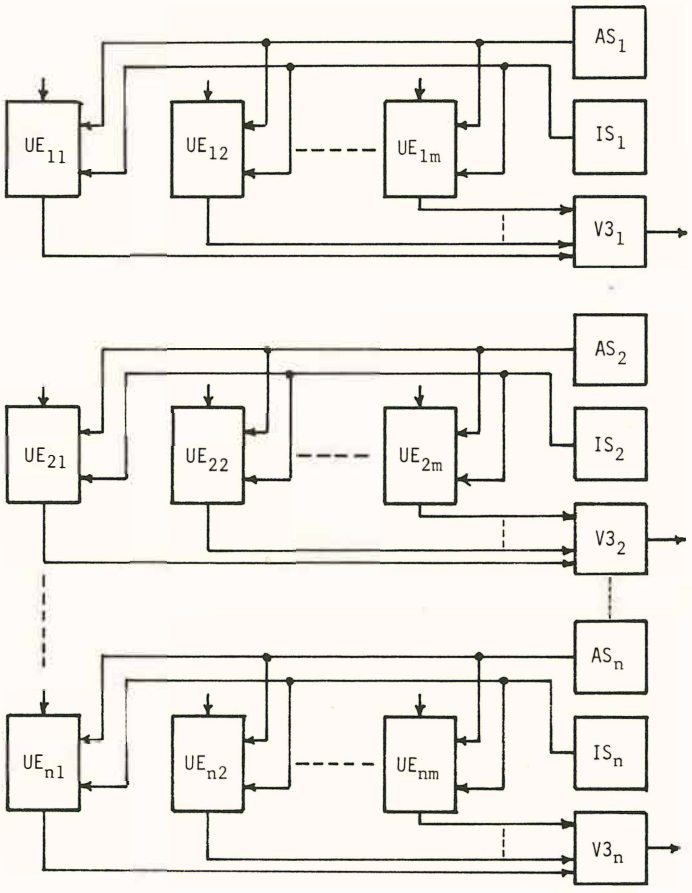


Abkürzungen:

AME	Aussenmasken-Element
AS	gruppenweise Aussensperre
ESE	Eingangsspeicher-Element
IME	Innenmasken-Element
IS	gruppenweise Innensperre
s,u	Unterbrechungs-Signale
UE	Unterbrechungs-Eingang
V1	Verknüpfungsglied 1
V2	Verknüpfungsglied 2
V3	Verknüpfungsglied 3

Bild 2 Element einer Unterbrechungsgruppe

Die Unterbrechungsprioritäten regeln den Vorrang der Unterbrechungseingänge. Die Bearbeitung einer Unterbrechungs-Anforderung der Priorität p kann nicht durch eine Unterbrechungs-Anforderung gleicher oder niedrigerer Priorität unterbrochen werden. Die Unterbrechungsgruppe mit Eingängen gleicher Priorität p heisst Unterbrechungsebene. Innerhalb einer Unterbrechungsebene wird die Auswahl der zu bearbeitenden Anforderung durch eine Subpriorität festgelegt.



Abkürzungen:

AS	Aussensperre
IS	Innensperre
UE	Unterbrechungs-Eingang
V3	Verknüpfungsglied 3 (für Eingänge gleicher Unterbrechungspriorität)

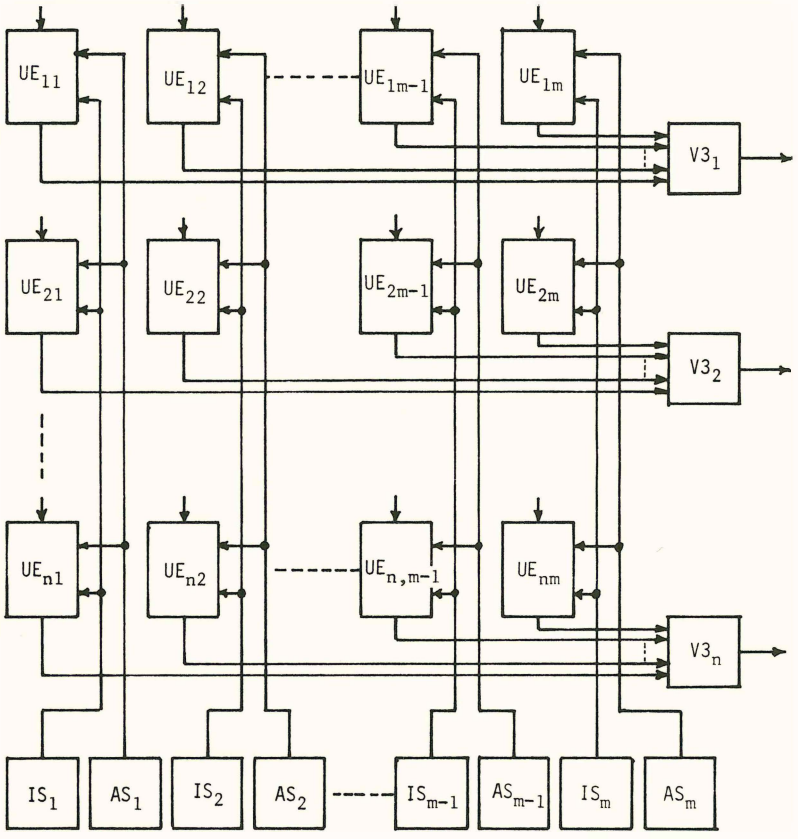
Bild 3 Beispiel zur Gruppierung der Unterbrechungseingänge, bei dem Unterbrechungsebenen und Gruppen mit gemeinsamer Aussen- und Innensperre zusammenfallen

b) Gemeinsame Lese- und Schreibbefehle

Bei einer solchen Gruppe können sämtliche Aussen-, Innenmasken- sowie Eingangsspeicherelemente gemeinsam gelesen und beschrieben werden. Die Zusammenfassung der Aussen- bzw. Innenmaskenelemente dieser Gruppe heisst Aussenmaske bzw. Innenmaske.

c) Gemeinsames Sperrelement der Eingänge

Diese Gruppe besitzt ein gemeinsames Sperrelement, das die Werte L oder O speichern kann. Dieses Element heisst Aussensperre, wenn es bei geeigneter Besetzung für alle Eingänge der Gruppe aussensperrende Wirkung hat. Es heisst Innensperre, wenn es innensperrende Wirkung besitzt. Aussen- und Innensperre konnen vom Rechner her gelesen, gesetzt und gelöscht werden.



Abkürzungen:

AS Aussensperre
IS Innensperre
UE Unterbrechungs-Eingang
V3 Verknüpfungsglied 3
(für Eingänge gleicher Unterbrechungspriorität)

Bild 4 Beispiel zur Gruppierung der unterbrechungseingänge bei dem Unterbrechungsebenen und Gruppen mit gemeinsamer Aussen- und Innensperre nicht zusammenfallen

d) Gemeinsame Anfangsadresse des Unterbrechungsanwortprogramms

Allen Unterbrechungseingängen dieser Gruppe ist die gleiche Anfangsadresse des Antwortprogramms im Rechner zugeordnet.

Die Gruppierung der Unterbrechungseingänge nach den in a) - d) genannten Eigenschaften kann zu verschiedenen Strukturen führen. Bild 3 zeigt ein Beispiel, bei dem die Gruppierung nach gleicher Unterbrechungspriorität mit der Gruppierung nach gemeinsamer Aussen- und Innensperre zusammenfällt; im Beispiel des Bildes 4 fallen diese Gruppierungen dagegen nicht zusammen (das in Bild 3 und 4 verwendete Schaltsymbol UE ist in Bild 2 erläutert).

5. Auswertung der Priorität der Unterbrechungssignale

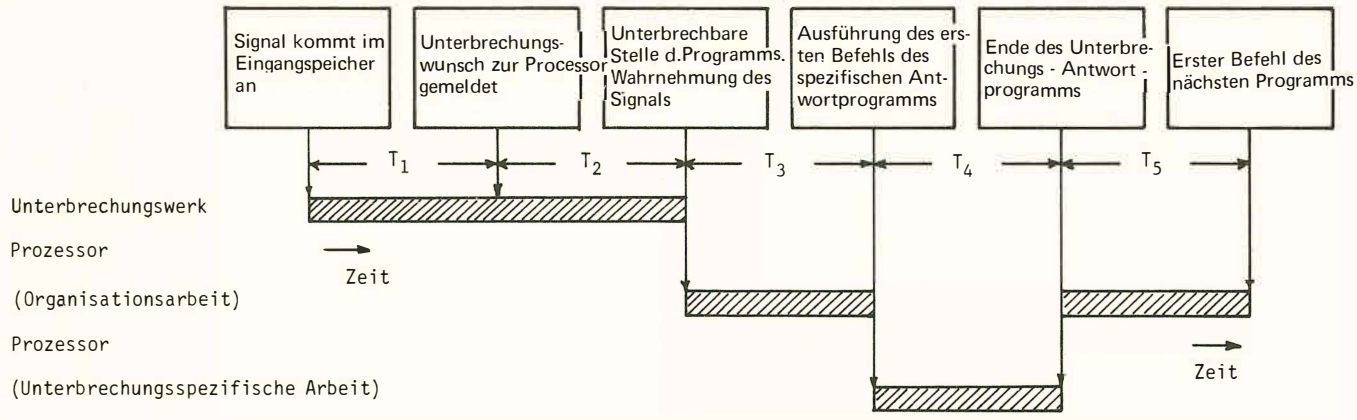
Es wird unterschieden zwischen Unterbrechungspriorität und Prozessor-priorität.

Die Unterbrechungspriorität ist diejenige Priorität, die einer Unterbrechungsebene statisch oder dynamisch zu einem bestimmten Zeitpunkt zugeordnet ist.

Die Prozessorpriorität wird einem Prozessor entsprechend der gerade von ihm bearbeiteten Aufgabe zugeordnet.

Die Prioritätsauswertung umfasst folgende Funktionen:

a) Zuordnung der Unterbrechungspriorität zu den aktivierten Unterbrechungsebenen.
b) Ermittlung der Ebene mit höchster Priorität.
c) Vergleich zwischen Prozessor-Priorität und höchster Unterbrechungs-Priorität (bei Mehrprozessor-Systemen Auswahl eines geeigneten Prozessors).
d) Entscheidung über eine Unterbrechung auf Grund folgender Kriterien:
   α) Vergleich nach c).
   β) Generelle Unterbrechungssperre gesetzt?
   γ) Vorliegen einer unterbrechbaren Stelle im Prozessor.
e) Innerhalb der Ebene mit höchster Unter-



Definitionen:

T_1 = Durchlasszeit
T_2 = Latenzzeit
T_3 = Erkennungszeit
T_4 = Ausführungszeit
(umfasst nur den Ablauf des unterbrechungsspezifischen Antwortprogramms. Identifizierung der Unterbrechungsursache und Bereitstellung der Betriebsmittel sind zu T_3 zu rechnen)
T_5 = Rückkehrzeit

$T_1 + T_2$ = Wartezeit
$T_1 + T_2 + T_3$ = Reaktionszeit
$T_3 + T_4 + T_5$ = Unterbrechungszeit
$T_1 + T_2 + T_3 + T_4$ = Antwartzeit
$T_3 + T_5$ = Organisationszeit

Bild 5 Definition charakteristischer Zeitintervalle

brechungs-Priorität Bestimmung der Unterbrechungsanforderung mit höchster Subpriorität.

Ausserdem muss die der Unterbrechungs-Anforderung zugeordnete Anfangsadresse festgelegt werden.

6. Definition charakteristischer Zeitintervalle

Bild 5 beschreibt ein Zeitdiagramm einer Unterbrechung. Im zeitlichen Ablauf werden sechs charakteristische Zeitpunkte fixiert und dadurch fünf Zeitintervalle T_1 bis T_5 definiert.

Die Zeitangaben für diese Intervalle können je nach angenommener Belastung des Rechners sehr unterschiedlich sein. Die günstigsten Zeitangaben erhält man, wenn man annimmt, dass der Ablauf der Unterbrechungsverarbeitung nicht durch andere Aktivitäten gestört wird. In allen anderen Fällen sind Angaben über die angenommene Belastung des Rechners mit anderen Aufgaben erforderlich. Zu Vergleichszwecken wäre eine Auswahl von Standardaufgaben durchaus sinnvoll.

Choosing a standard high-level language for real-time programming

M.G. SCHOMBERG
AERE, Harwell, England
and
P. WARD
Plessey Radar, The Plessey Co, England

Introduction

This paper is concerned with the problems of providing high-level language facilities for a real-time project. It covers particularly those features necessary for a real-time applications programmer and does not consider the rather more special problems for the systems software programmer.

A large number of real-time projects are programmed entirely in assembly language for a variety of reasons. These range from necessity, due to space and time requirements, to simple prejudice. This paper assumes that the use of a high-level language is justified for the project.

Section 2 of this paper considers the alternative approaches of using existing languages or writing special ones. Section 3 identifies the special features necessary for a real-time language and how they can be achieved and in Section 4 a method of selecting the most appropriate language for a hypothetical project is demonstrated.

General-purpose or special-purpose languages

There seem to be two main schools of thought concerning the selection of a suitable high-level language for real-time projects. The first of these, which comes from the more academic circles, implies the development of a special language for almost every individual problem. The alternative view, supported by this paper, favours the use of existing high-level languages whenever possible.

The 'academic' point of view is based on the use of the programming language not only as the means of communicating with the computer but also the framework for thinking about the solution of the problem. On this basis they argue that an apt language will lead to a good solution, while an inappropriate one may lead to a poor solution. In addition, the special language for the particular situation includes within itself standard solutions to some of the fundamental operations required to formulate the solutions. Thus the programming language points the way to the solutions by already having solved some problems and laying the foundations for the solution of the rest.

The 'practitioner' point of view rejects the conclusions of this argument, not by denying the facts but by pointing out their characteristic of only marginal improvements in comparison with more general-purpose languages, and considering other problems and constraints of greater weight. The time scale for development of a new language is long (from initial ideas, constructing the compiler, testing and assessing the value of the new features). A new language also implies re-training of programmers and the writing of user manuals. Consequently it is not usually practicable to undertake to do it within an already tight time scale for a development project. Furthermore, the solutions to the problems involved will be discovered by people working in the project with their individual prejudices, experience, capabilities and potentials exerting a much stronger influence than the programming language. A man using a new programming language is most likely to continue with design habits acquired during his previous experience with other languages: thus the potential benefits of a special-purpose language are weakened, and the effort which would have to be expended to develop one is not judged to be justified.

Real-time features

This section identifies the special features associated with real-time programming and then attempts to show how they can be realised in more general-purpose languages.

These are divided into three categories.

1. Time-dependent features: those features which control and monitor the dynamic system state. They include:

   I/O control
   resource allocation
   storage control
   interrupt handling
   re-entrancy
   parallel processing
   non-sequential operations

2. Time-independent features: those features which are mostly found in the solution of real-time problems but which are not strictly time-constrained. They include:

   byte handling
   list processing
   string handling
   bit addressing
   in-line insertion of code

3. General features: those features which are not only important in the real-time field but also in other fields of computing. They include:

   modularity
   macros
   data structures
   well established
   ease of use
   ease of program maintenance

   The last three, although not strictly language 'features', are included to bring out the importance that is attached to these aspects of a language.

In the above it is only the time-dependent features which require communication with the operating system. It is therefore assumed that the operating system provides the necessary 'hooks' to perform such operations as the scheduling and suspension of tasks and resource allocation as requested by the 'user' programs.

How these special features can be provided

Having established what special features are necessary for programming a typical real-time application it is possible to examine the means by which they can be provided.

Consider the problem faced by the programming manager about to undertake the implementation of a real-time project. He has a computer, a general-purpose operating system and one or more general-purpose high-level language compilers.

He must first identify precisely which real-time features are necessary for the project. Those which only the operating system can provide will entail either modification to the existing operating system to provide them or writing a new operating system. Those features which are independent of the operating system are discussed later.

A means must then be found of making the additional features of the operating system available to the programmer. There are three possibilities:

1. To provide the facility as an assembly code subroutine which can be called from the user program.
2. To provide a facility in the compiler whereby in-line assembly code instructions can be used. This facility can be given a high-level flavour by the use of macros.
3. To modify the compiler to provide a set of new language statements which ensure the necessary linking to either the operating system or to a library subroutine that performs the required function.

The first alternative of using assembly code subroutines is probably the easiest to implement but the subroutine calling sequence will result in some loss of object code efficiency. The other alternatives avoid this by generating in-line code. If the compiler permits the insertion of in-line code then this can be achieved by a macro pre-processor. The chief penalties here are a reduction in the general legibility of the source program and a loss of checking by the compiler. The desirability of enhancing an existing compiler to include new language statements can only be judged in each individual case.

Time-independent features

The above considerations lead to an unexpected conclusion. Those language features normally associated with real-time programming are not, in fact, the most essential features when assessing a language for a real-time project. Such features can usually be provided by other means without undue difficulty.

However, the language feature provided by a general-purpose language that is least likely to match the requirements of a real-time application is its data structures. The ideal data structure will probably be different from one real-time application to another and requiring any combination of list structures, multidimensional arrays, Coral TABLE structures or the COBOL hierarchical record structure, while, at the same time, being influenced by data retrieval requirements. PL/1 is the only language that approaches this degree of flexibility. However, in spite of this, real-time projects do get implemented and such problems with data structures do get overcome as they arise; apparently without too much difficulty. It is possible to describe data structures in a long-hand manner if suitable language facilities are not available.

To conclude this section, it can be seen that all the special real-time features, except perhaps data structures, can be provided without the need for special language development. The lack of suitable data structuring facilities makes the programmer's job more difficult but not usually

impossible. Some of the currently available general-purpose languages provide adequate facilities which enable enhancement to the level necessary for most real-time projects. It is therefore suggested that the use of such languages provides the most cost effective solution to this problem in many instances.

Graded assessment of high-level languages

This section examines six commonly used languages for their suitability for a real-time project. They have been selected on the basis that they include the most widely used languages and represent a wide spectrum of applications. Others, e.g. JOVIAL, should perhaps be included for completeness, although the purpose of the exercise is to illustrate a method of assessing languages and not to do an assessment.

Method of assessment

In order to carry out an assessment of programming languages for any project it is first necessary to list all language features and other factors which have any effect on the application. This list of assessment criteria is then grouped together according to their relative importance. A typical grouping might be as follows: essential features, highly desirable features, desirable features and possibly useful features. Each group is then assigned a grade range. For example, the most important group is given a grade of 100, the next group the range 50-99, and so on. Within this grade range each of the assessment criteria is given a particular grade to show its relative importance compared with other criteria in the same group. Each language is then taken in turn and for each of the criteria is given a mark up to the maximum allowed for that criterion. This mark must reflect the capability of the language to meet the criterion compared with the other languages being assessed. Once the assessment table is complete it is possible to carry out a simple elimination exercise.

Any language which does not meet all the essential criteria can immediately be eliminated. By totalling all the marks for each language the comparison between them can be made. Hopefully, the language showing the highest over-all total is the best for the job.

Although the initial marking of features for a particular project must necessarily be subjective and based largely on experience, once this has been done the actual assessment process is probably fairly reliable.

Other factors, however, influence the final choice of language. Among these, and perhaps even more important than the language itself, is the availability of a suitable compiler for the selected computer. To have a professionally implemented compiler which has stood the test of time contributes more towards the successful achievement of a project than any number of features specially designed for the project. This is particularly so when such features can be provided, perhaps less elegantly, by other means. Full error checking and unambiguous error reporting, comprehensive de-bugging facilities and a good user's manual are of prime importance in the use of a language in any project. These features can, of course, be included in the language assessment criteria.

Table 1 illustrates this technique for a hypothetical project. The main points that the table brings out are as follows

1. The relative merits of each feature to each other feature, e.g. it is more important that the language is easy to learn and use than it should have I/O statements.
2. How each language performs for each feature, e.g. FORTRAN and COBOL are easy to learn, whereas PL/1 is the most difficult.
3. Neither COBOL nor ALGOL can be considered for this project.
4. PL/1 is the most suitable language for the project.

This method is both simple to use and understand and can be designed to any level of detail required. It is fairly objective in its approach and enables the value and effect of new information on the choice of language to be readily assessed and leaves irrelevant arguments to be discarded immediately.

Conclusions

This paper has sought to show that in many real-time projects it is more appropriate to use existing languages than to develop special ones. It is suggested that most time-dependent features can be realised without too much difficulty and that it is more important to consider such aspects as a good implementation of an existing language which has the necessary data structures. The final section illustrates one way of selecting that most suitable language for a real-time project.

Discussion

C. I applaud your courage in presenting specific figures.

C. The results have little meaning, because the weights can be changed to manipulate the figures.

Q. What do you mean by dynamic storage in this context?

TABLE 1 Language assessment for a real-time project

	Max. points	Fortran	Cobol	PL/1	Algol 60	Algol 68	Coral 66
Essential features							
Modularity	100	100	0	100	0	100	100
Bit/byte addressing	100	100	0	100	0	100	100
Sub total	200	200	0	200	0	200	200
Highly desirable features							
Data structures	85	0	85	85	0	85	50
Macro facility	85	0	0	85	0	0	85
Easy to learn and use	60	60	60	30	50	40	40
Good documentation and maintenace	60	60	30	60	30	30	30
Parallel processing	60	0	0	60	0	60	0
In-line code insertion	60	0	0	0	0	0	60
Usage	50	50	50	30	30	0	10
Sub total	460	170	225	350	110	215	275
Desirable features							
Dynamic storage	40	0	0	40	40	0	40
I/O statements	40	40	40	40	0	40	0
String handling	40	0	40	40	0	40	0
List processing	30	0	0	30	0	30	0
Floating point	20	20	20	20	20	20	20
Sub total	170	60	100	170	60	130	60
Possibly useful features							
Free format	9	0	0	9	9	9	9
Machine independence	5	0	5	3	3	4	3
Recursion	1	0	0	1	1	1	1
Sub total	15	0	5	13	13	14	13
TOTAL	845	430	330	733	183	559	548
Implementation features							
Compiler available	90						
Compiler usage	90						

A. The program's facility of influencing storage allocation.

Q. How would you take into account the quality of the programmer?

A. We had this in mind in 'easy to learn and use'.

C. You should also take into account
a) the experience of the programmer
b) the size of the computer.

C. Learnability should be taken into account in language design.

Use of high- and low-level languages in a real-time system

J. STENSON
Plessey Radar, The Plessey Co, England

Introduction

This paper discusses briefly some of the arguments for and against the use of high-level languages in a real-time system. Then it goes on to describe the results of efforts made by the Plessey Company to use a high-level language in parts of a real-time operating system.

It is hoped that our experience will be interesting to everyone who is looking at the use of high-level languages in real-time systems.

1. For and against the use of a high-level language

Consideration of whether to use a high-level language rather than a low-level one is influenced by the following factors:

1. Writing in a high-level language is much quicker.
2. Coding errors are less frequent.
3. Testing is often much quicker at the off-line stage.
4. Documentation is sometimes simpler.
5. Handover of the program is easier – whether between programmers or to the customer.
6. Even an efficient high-level language usually needs more space and more run time than a lower-level one.
7. If obscure bugs are found on line, the programmer will probably have to look at the machine code instructions to trace them, and this is easier from a low-level language than a high-level one.

A balance has to be reached between speed of program production, achieved by using a high-level language, and efficiency of code, achieved by using a low-level one. This is a very difficult decision because programmer time is always expensive, and any means of reducing it is worthwhile, but time and core space in most real-time systems are precious.

The solution may be to make high-level languages more efficient, so the object code they produce is nearly as good as a programmer can write. Or it may be to improve low-level languages to incorporate some of the features of high-level ones. Either or both of these improvements may be made.

When the decision between high- and low-level languages had to be taken we had time-scales to meet and limited space and time in the system. The rest of the paper shows how we decided between high and low languages, and the results of our choice.

2. Brief system description

This system is a multicomputer Air Traffic Control project. The computers are of two main types, and not all have access to the same facilities – for example, some have access to magnetic tape decks, some do not. All the computers can communicate with a common store in which the data base is held.

Each computer receives one interrupt at regular intervals, and from this interrupt all the system timing is derived, although there are other significant times, e.g. radar scan time and data link time, which have to be observed.

The system can be considered as having two parts, a 'foundation', which would be nearly the same whatever the system were used for, and 'application software', which is concerned only with the air traffic control functions of the system.

The 'foundation' consists of an operating system providing scheduling, communications, peripheral handling, reconfiguration, fault detection and location. Parts of this have been implemented during the past year, and it is from this implementation that the following information has been drawn.

3. The operating system

The operating system contains programs which perform the following functions:

1. Schedule tasks in every computer in system.
2. Provide communication between the computers in system and the common store.
3. Handle peripheral equipment – teleprinters,

punches and readers, magnetic tapes. Also the 'control panel', which is hardware device controlling the configuration of the system.
4. Control and information service to the operator. This program is known as The Director.
5. Detect faults.
6. Load programs.
7. Aid on-line testing.

4. Available languages

In this system we use two types of computer, the XL4, a double address machine, and the XL6, a single address, less powerful computer. For the XL4 we have a high-level language, MINICORAL, a subset of CORAL, and XAL, an assembler. For the XL6 we have XAL only.

MINICORAL can be a very efficient high-level language. An experienced programmer writing with store economy in mind can achieve a 1.1:1 size ratio – we have tried experiments and it can be done. But, of course, a less experienced programmer can produce much worse results than that; 2:1 is about average. There are very good off-line testing aids associated with the language, and this is a great advantage.

XAL is a mnemonic assembler with very good macro facilities. It is easy to learn, but there are fewer off-line testing aids associated with it, and therefore programs can take longer to debug. Really experienced XAL programmers have very little difficulty debugging their programs.

5. Available people

The programming team contained people with a wide range of programming skill. They ranged from those with 5 years' experience to those with none at all. Our trainee programmers were used mostly on the MINICORAL programs.

6. Split between high and low-level languages

When we looked at the list of tasks to be performed (Section 3, 1-6) some points were obvious immediately:
1. Some programs would have to exist in every computer in the system (scheduler, communication, on-line aids, fault detection).
2. Some were restricted to only a few computers because of the arrangements of the hardware (magnetic tape handling, control panel handling).
3. From the nature of their design some programs needed to appear in one (or only a few) computer only (The Director, Reload).

A simple split would be to use XAL for programs held in every computer, MINICORAL for those held in only a few. To a certain extent this was done but there were a number of weighting factors to be applied. These were:
4. Any program in the XL6 must be in assembler.
5. Some programs which appear in every computer in system have very low priority, and if their run time is a little bigger than it need be it may not matter very much, since the program will only run when there is spare time anyway.
6. Some programs which appear in every computer are run on rare occasions only in the operational system. Therefore their space and time need not be optimised to the same extent as others.

Starting from the position that we wanted to use MINICORAL if possible to reduce program production times we arrived at this split between MINICORAL and XAL:

MINICORAL	XAL
	Scheduler
	Communications
Control Panel Handling	Teleprinter Line
	Printer
	Paper Tape Handling
Magnetic Tape	Magnetic Tape
Handling (XL4)	Handling (XL6)
Director (XL4)	Director (XL6)
Fault Detection	
On-Line Aids	On-Line Aids
Reload using XL4	Reload from XL6

7. Review of each program

Each of these programs is considered below. The arguments for using the particular language are given, and where MINICORAL was used the results are considered. Where XAL was used less information is included.

There are two areas, Magnetic Tape Handling and The Director, where comparison between a MINICORAL and a XAL version of the same program can be made. It is not a direct comparison, because the MINICORAL versions were for XL4 computers and the XAL for XL6. The XL4 has more powerful instructions than an XL6, so one expects to find fewer instructions used in this computer.

7.1 Scheduler, communication

These programs were written in XAL because they are used in every computer in system, and therefore space is important. They are used con-

stantly in every cycle of work, and therefore time is important. We considered that any increase of space or time could not be allowed, and they were written by skilled assembler code programmers.

7.2 Teleprinter, line printer and paper tape handling

These were written in XAL because MINICORAL does not offer particularly good facilities for this type of peripheral handling, they were short programs anyway, and some were for the XL6. We did not spend very long over this decision – peripheral handling of this type seems to demand an assembler code.

7.3 Load

These parts of the load programs housed in an XL6 were written in assembler code, but where they were housed in an XL4 they were written in MINICORAL. This is because they are used comparatively rarely in the operational system (probably less than once every six hours) although they are heavily used during program development. It seems more important in a case like this to ensure that the program can be maintained easily and handed on from one programmer to another than to achieve minimum space or time. Space had to be considered, because there is limited space in the System. The programs did not occupy an excessive amount of core (0.7 K in 4 computers of 64 K capacity).

If we had to make this decision again we would make the same one. Any program used at a very low rate is a reasonable vehicle for an experiment in using high-level languages.

7.4 Control panel handling

The program resides in one computer only in the system, and is called every cycle. No significant amount of processing takes place unless the operator uses the System Control Panel to reconfigure the hardware in the system. This should be a rare occurrence – major reconfigurations involving computer changes should not occur more than once in 24 hours, and minor ones should happen less than once per hour.

In these circumstances no serious penalty in overheads will be paid at run time if the program is slightly slower than it need be. It was, therefore, written using the MINICORAL compiler.

The program was written by experienced programmers. It was completed quickly and with very few unforeseen difficulties. It operates well within the time available to it, and whenever the program has been run it has given satisfactory results.

This was another area where we feel we made the right decision. The ease of production and documentation justified the use of the high-level language, and no significant time penalty has been paid.

7.5 Fault detection

This program is scheduled at the end of the lowest priority list of tasks. It is called when all other work has been done, and having been completed adds itself to the bottom of the list again. Thus if there is very little work being done in the computer this program is run frequently, but as the work load builds up so the calls become fewer.

As the program will only be used when the workload allows it, the time overheads incurred if it is written in MINICORAL are not serious. The program resides in every computer in system, and therefore space is a matter of concern.

Despite the consideration of space the program was written in MINICORAL to reduce timescales. The space occupied is more than we allowed for when the program was designed, and we are faced now with a need to look through the program to find ways to reduce the space used.

Nevertheless, the program was ready on time and it can be used in its present overlarge state for a few months before the 'applications' programs increase in number and its size becomes a problem.

This is where the judgement between the two alternatives is difficult – we did achieve a result on time, which is important, but the excessive size of the program is just as relevant.

Summing up, it is apparent that this MINICORAL program is less successful than the two previous ones.

7.6 On-line aids

On-Line Aids will be used most frequently during the system development stage, at which run time and space overheads are not particularly significant. (It is possible to alter real-time situations just by using on-line aids, but this situation is unlikely to be made worse by longer run times.) The aids will be used occasionally in the operational system, but not often enough to make run-time overheads significant. Like 'fault detection' (Section 7.5), they exist in every computer in system, and space is important.

For reasons of speed of production, allied to a conviction that MINICORAL should be a good language for tasks like this, we decided to do two-thirds of the total work in MINICORAL.

The project suffered throughout its life from changing manpower, and was frequently understrength. Despite these difficulties it was completed on time: but its size is excessive. We attribute this to its low staffing level leaving no

time for a careful examination of the coding of each module to ensure that minimum space was used. Like 'fault detection', we now have to carry out this examination and reduce the size of the program.

It is possible that the same troubles would have been met if we had elected to write the program in XAL; in fact, it might have been even worse.

It is difficult to tell with this project whether it would have been better to write in a low-level language. It seemed to be a function of the low manning levels rather than an inherent difficulty in using a high-level language that caused the growth in size.

7.7 Magnetic tape handling

Only 2 of the XL4 computers are expected to use magnetic tape decks at any one time and they will not be in constant use. This was an obvious place to use MINICORAL and it has been used very successfully. At present the XL4 version is estimated at 2 K (with 75% completed, so the estimate should be reliable).

The XL6 version, written in XAL because there is no other choice, is 2.2 K.

These figures show a very close correlation between the MINICORAL and XAL versions.

7.8 Director

The Director has two parts, one of which resides in every computer in system, the other in one XL4 and XL6 only. The part that occurs in every computer in system was written in XAL for economy. The control part, in one XL4 and XL6 computer, runs once every cycle. This means that size and run time are not critical, although they cannot be ignored. The program for the XL4 was written in MINICORAL and that for the XL6 in XAL.

The sizes of these programs are: XL4, 2.2 K; XL6, 3.6 K. This is a very satisfactory result – XL4 programs should be smaller because the instruction set is more powerful.

8. Conclusion

In this multi-computer system it was possible to use a high-level language in those areas which were not common to every computer in the system. When a high-level language was used in an area which was common to every computer the results were less successful, although it is likely that the fault did not lie with the compiler.

From the experience gained in this system, designers may well be less sceptical of the wisdom of using high-level languages in real-time system than they have been in the past.

Acknowledgements

This paper is based on the work done by Plessey Programmers in System Implementation with the help and collaboration of members of the Royal Radar Establishment at Malvern.

Discussion

Q. You remarked that the most successful programs are those that are run least often. Is this because these programs are tested less thoroughly?

A. Not quite, but perhaps because they are less prone to interactions from other debugging, and errors in them have less far ranging consequences.

C. Good programmers want to get all the power of the machine, so they program in assembler. But the very best programmers want to get this in higher level languages.

C. Even when using low level languages we do not allow our programmers to use 'tricky code'. With high-level or macro languages one should accept some overheads and inefficiencies which result from enforcing programming standards.

Workshop on basic design principles of operating systems

Chairman: **Dr I.C. PYLE**
AERE Harwell, England

The discussion was concerned with operating systems, their definition and programming. The usefulness of the idea of virtual machines was agreed, although the concept itself is not clearly specified. On the issue of programming language, the question was asked whether operating systems must be written in languages different from those used for general-purpose real-time or process control; and if so what are the essential differences?

Operating systems and virtual machines

L. Is an operating system the 'glue' which holds together the facilities that a user has at his disposal? Or is the operating system that which is performed by programs rather than carried out by the system operator? Different people seem to use the phrase for anything between these extremes.

B. Looked at from the outside, e.g. a process control application, we need to estimate the stability of a computer system with respect to changes in

a) The environment which may affect the operating system such as new kinds of devices.
b) The application programs to be executed, such as new control algorithms.

Regarding the hardware plus operating system as a virtual machine helps us to do this.

P. There are three different aspects of an operating system, which one should clearly distinguish in discussing the levels of a virtual machine:

a) The facilities through which the user can use the hardware of the computer system more easily.
b) The facilities through which the user can control interactions of his own tasks.
c) The facilities through which the operating system can control the user program to share resources between several user programs.

B. An ESONE Working Group is engaged in trying to define an interface between application programs and operating systems for use with CAMAC. This is guided by the concept of a virtual CAMAC controller and the virtual machine concept is found valuable in the work.

E. Is a real-time system only a small-scale batch system?

L. No, because the programs do not cooperate.

M. A language defines a machine. The definition of any programming language must imply a virtual machine, which can run programs written in the language.

Programming languages

M. The operating system is an extension of the hardware and has to be able to interact with it in very detailed ways which may not be appropriate for application programs. It should therefore be programmed in a different way.

J. The time factor is an important difference between the language chosen for operating systems and application programs. The operating system is called frequently in comparison with applications programs and therefore must be written in very efficient language.

X. This is not uniformly true for operating systems: parts are used with different frequencies.

L. Not only do we need to consider the basic operating system primitives, we need to realise that the important structure in a real-time system is a process rather than the program. We need programming languages in which the idea of a process is clearly expressed.

P. It is important to distinguish between a 'real-time language' which incorporates timing facilities, and a 'language for real time' which is suitable for use in real-time programming while not having any specifically real-time features. CORAL 66 is

one of the latter.

H. An application package should look like a virtual machine when it is finished, therefore you should have a common language with which to construct them. A language for operating systems should have more privileged features so that the application programmer has a subset of the same language.

COMPONENTS OF REAL-TIME SYSTEMS

Chairman:
Prof. Dr R. BAUMAN

Basic supervisor facilities for real-time

I.C. PYLE
AERE Harwell, England

Present address:
Department of Computation, University of York, England

An analysis of the supervisor facilities in a number of real-time operating systems leads to the specification of a number of basic facilities. These cover task handling, resource allocation, and inter-process communication. The parameters required for these facilities include identification of tasks and identification of peripheral devices. The handling of I/O may be treated as an autonomous process to carry out the transput, with inter-process communication transferring the data to the main process.

Introduction

Several papers at the First European Seminar on Real-Time Programming discussed the interface between the real-time programming language and the real-time operating system. In our investigation of CORAL 66 as a programming language for real-time work, on a PDP-11 with the Disk Operating System (DOS), we have been seeking to determine a suitable set of basic supervisor facilities which we could implement by elaboration of the DOS Monitor. Our hope is to establish a standard interface between a running CORAL 66 program and the supervisor (or run-time operating system) controlling it.

CORAL 66 as officially defined (Woodward, Weatherall and Gorman, 1970) does not require a run-time operating system. In practice, programs written in CORAL normally will use a run-time operating system (i.e. a supervisor program) so that the program as written does not have to be concerned with the details of peripheral handling, and to permit multiprogramming. This is an extension of the concept of a library, which is in official CORAL 66. The other point about the official non-insistence on a supervisor is that it enables the supervisor itself to be written as a CORAL 66 program. This has been done for the Myriad Computer, making the supervisor called MOLCAP (Jackson, 1970).

An attempt to standardise supervisor calls amounts to an extension of the CORAL 66 definition, by establishing certain facilities which would then be common to all CORALS at the appropriate level. In order to make the definition strong enough, we have to define the facilities to be provided by the supervisor with great care and introduce a suitable notation by which they may be specified in a CORAL program. The latter part, the syntax, is not difficult. The major part of the problem is in specifying the facilities.

Supervisor facilities

There are three principal categories of supervisor facility: Task (or Process) facilities; Resource Allocation facilities; Communication facilities. The first of these is actually a special resource allocation (since it is concerned with the allocation of the processor in the computer), but it is sufficiently distinctive to warrant a separate category. The third covers what is normally called I/O, and also inter-process communication. I/O devices may be simple slaves to the main processor, requiring simple communication facilities, or they may have autonomous capabilities in which case they are best regarded as separate processors running in parallel with the control or computing processor(s), and the communication between the processes propelled by these processors is inter-process communication.

The principle of selection of facilities for a prospective standard must be to keep the list short and simple. The supervisor occupies valuable store during run time, and must not be made unnecessarily elaborate. If the standard covers more than the bare minimum the supervisor program will provide the facilities whether or not the application program requires them. Any enhancements needed to elaborate the supervisor facilities can be included in library procedures to be loaded only when required, which call on the supervisor for the primitive facilities. Accordingly, we proceed by giving a list of facilities which appear to be primitive.

The entry conventions for supervisor calls will have to be settled for each particular implementation, but the standard must specify what parameters are to be transferred for each supervisor call. Some computers will have an established supervisor call instruction, others will use a trap or entry to a standard location, or force

some special exception to occur. The standard cannot expect to lay down which method shall be used, but can reasonably assume that there will be just one general method of supervisor call in each implementation, which is different from an ordinary procedure call (for example, to cause a change of processor state where there is a difference between user mode and supervisor mode).

Task facilities

The facility which almost all supervisors will provide is 'generate task'. This will activate an already loaded program to operate on given data in the store, at a specified execution priority; the supervisor will provide the identification for the task. Thus there are four parameters, three input and one output:

entry location	(a destination)
data origin	(a pointer)
priority	(an integer)
task identification	(output, probably a pointer).

The priority may be omitted: the default value will be the priority of the task issuing the supervisor call. There may be rules prohibiting a task from generating other tasks of higher priority than itself. Improved ideas on task scheduling may lead to a better method of regulating the allocation of processors to processes, but at present the use of a priority number is the most suitable.

The next and most fundamental supervisor facility is 'terminate task'. Every program needs to be able to specify its dynamic end. With computers which have a 'stop' instruction, the first rule for programmers is not to use it, especially in real-time work. There can be two variants of this supervisor call: terminate own task (without parameter) and terminate other task (with other task identification as parameter). This introduces the need for task identification and the problem of naming tasks. The standard cannot solve the problem, but must assume that a solution is chosen in a particular implementation, and any restrictions consequent on that solution limit the range of other tasks which can be controlled. The two variants can be viewed as the same supervisor call, with a default value for the parameter if it is omitted, being 'self'. For a discussion of the problems of task identification, see Taylor (1972).

As an alternative method of terminating a task, but denoting an abnormal or exceptional dynamic end, it is desirable to have a separate supervisor call: 'abnormal termination'. This applies only to the current task, and carries a parameter which can express the nature of the abnormality or exception.

A task which may be subject to abnormal termination from another task should be given the ability to specify its own local termination, to do any tidying up necessary. The local termination would be a procedure, which is activated by the supervisor immediately after a supervisor call to terminate. It would have one input parameter, being the termination code. There has to be a supervisor call to pass this local termination procedure to the supervisor for the task. If there has been no such procedure specified before termination, then the supervisor removes the task immediately after a supervisor call to terminate; otherwise it activates the local termination procedure (after removing the record of it from the task entry), and on return from it removes the task. Thus we have a supervisor call

set task termination procedure (p)

with one parameter (of type PROCEDURE (INTEGER VALUE).

A task in a real-time system may require a delay of a period of time. We therefore need

wait (number of time intervals)

where the magnitude of the time interval will be system-dependent.

Delayed activation

A powerful technique, especially useful in real-time for dealing with time-out situations, is a delayed activation which can be cancelled. With an activate which takes effect after a specified time delay, the time-out action can be easily set up, leaving the supervisor to look after the timing. The additional facility which is necessary is to be able to cancel the delayed activation, if the awaited response arrives in time. Since this would be the normal situation, the supervisor must keep its delayed activation requests in a way which permits efficient cancellation.

This can be treated as a combination of the supervisor calls to activate and wait (activate the time-out action immediately, but put an appropriate wait at the beginning of the task). The cancel is then simply a call to destroy the task, which will take place while it is in the wait state. There is a danger of trouble if there are other waits in the time-out action task, since the task might then get destroyed in a partially completed state. This can be solved by the discipline of insisting that such a time-out action contains no further waits, although it may in turn activate a further task (which would not get destroyed when the response comes).

Task synchronisation and interrupts

The well known primitives for handling semaphors permit a task to be held up at defined points until another task gives it the go-ahead:

Secure (semaphore in*)
Release (semaphore in*)

There are arguments in favour of a further supervisor call, which cannot be implemented by use of the above:

attempt to secure (semaphore in*, status out*)

This is equivalent to secure if the semaphore would permit the task to proceed, but otherwise gives the status of the semaphore without suspending the task. Releasing a semaphore gives a stimulus to the task which had previously made a secure for it.

Stimuli can arise from outside the computer (hardware interrupts) or from other running tasks. The effect of the stimulus is not directly to create a task, but to resume execution of a task which must previously have been set up and suspended awaiting the stimulus. In order to deal with critical sections, it is necessary to be able to inhibit the response to stimuli for certain periods of time.

Thus the primitives required are:

Set response to stimulus (stimulus identity in, semaphore out)
Inhibit response to stimulus (stimulus identity in)
Permit response to stimulus (stimulus identity in)

In the task to be performed, it is normally held up by the semaphore; it starts in response to the stimulus, and must return to the semaphore wait again if it is prepared to respond to further stimuli.

The principal problem about this technique is identifying the stimuli between separate processes. In a simple system, the natural solution is to adopt a system-wide assignment of stimulus type numbers corresponding to distinct physical interrupts at the hardware level.

Resource allocation

The allocation of resources other than processors and main store will assume that they are discrete, and have in general to be allocated in two stages: shared and exclusive. Every resource has initially to be reserved, and it will in principle be shared. This means that there need be no time delays on reservation. A claimed resource can then be secured for exclusive access, and must be released when exclusive access is no longer needed. The resource must be discarded when it is no longer needed. A subsequent reservation will not necessarily get the same device, assuming that there are several of the same type forming the resource. Depending on the nature of the resource, different things may be done when the resource is claimed and secured.

Thus we need two supervisor calls:

Reserve resource (resource type in, identity out)
Discard resource (identity in)

To handle the exclusive use of a resource, we need to use a semaphore which will control access to that resource:

Prepare for multiple use (identity in, semaphore out)

The exclusive uses are then controlled by the use of the semaphore.

Communication

We assume that the executing task may communicate with a number of data sets outside itself, which may be accessed sequentially. Streams of data will be either input or output, and may be buffered. A stream which is output from one task may be input to another task, or after it has been closed, to the same task. Buffering implies having separate autonomous tasks (usually very simple) complementary to the main task: if the main task puts information to an output stream for printing, then an auxiliary task has to read the output buffer and print what it finds there.

We have to consider input output for different categories of devices, depending on the nature of the information they transmit in an elementary operation. Basically, there are character devices (e.g. teletype), word devices (e.g. ADC samplers) and block devices (e.g. disk). For character devices, it is usual to deal with streams of characters, and it is more economical to specify the source or sink separately, rather than with each individual input or output character

Set source (device identity in)
Set sink (device identity in)
Input character (character out)
Output character (character in)

For the character devices, it is often desirable to be able to translate the character code, so that a uniform internal character code can be used, but this is more economically done by a library procedure on a complete line of characters together. Words are input and output from explicitly identified devices:

* The suffixes 'in' and 'out' distinguish input and output parameters of the supervisor call.

Input word (device identity in, word in)
Output word (device identity in, word out)

For block devices, a buffer area is needed, which has to be set up by the program in a special format, and then given as a parameter in the supervisor call. This can include the specification of which area on the device is to be used, and the direction of the transfer:

Put block (buffer in)

Storage allocation

No run-time supervisor facilities are proposed for allocation of main storage. This is a deliberate restriction on the flexibility of the operational system, which corresponds to the essential characteristics of the situation, at least at the present state of the art. The argument for excluding them rests on the requirement that the real-time programming language be predictable.

If there is run-time allocation of storage by the supervisor, then a general strategy has to be adopted to deal with block of various sizes and unco-ordinated requests and releases. Consequently store fragmentation is a real danger, and from time to time an unpredictable store jam will arise; in this event, either the system will be brought to a complete halt or there will be a significant time delay while there is some reassignment of storage.

The implications of this decision are that the allocation of storage is done at load time, with the user's program thereafter responsible for how the store is used. Thus, for example, the program can state that it needs a stack, and must specify to the loader an appropriate maximum size for this stack. Space will be allocated permanently for the maximum size stack, and within this the user program keeps its stacked variables and its own stack pointer. Separately, the program can state that it needs some buffers of a certain size, and must specify to the loader an appropriate maximum number of buffers which can even be concurrently in use. Space will be allocated permanently for the maximum number of buffers, and within this it is up to the user program to control its own use of the buffers, and take its own action if the situation arises when all buffers are in use and it needs another one.

In other words, the organisation of store provided is fixed at load time as far as the supervisor is concerned, and any dynamic usage is directly controlled by the user program. This is in accord with the views expressed at the First European Seminar on Real Time Programming (pages 31 and 32, speakers D and G).

Conclusion

This paper has begun the process of specifying the supervisor interface, by concentrating on the areas of tasks, communication and resources. The interface does not specify any facilities referring to protection between processes or store areas, because it assumes that this matter is handled entirely within the supervisor, and needs no further information or stimuli from the work program. It has not covered store allocation or file structures, so is far from complete for a full operating system. Likewise it does not include backing store usage or swapping facilities. Nevertheless, facilities such as these will be needed within some real-time operating systems. At the First European Seminar on Real Time Programming, it was thought premature to make such an attempt. It is still premature to expect a standard to be widely adopted, but it is not too early to start thinking about the problem.

1. JACKSON, K., 'Myriad Library Part 2.1', RRE (1970).
2. TAYLOR, J.R., 'Control of names in systems with dynamically created objects', AERE-R (1972).
3. WOODWARD, P.M., WEATHERALL, P.R., and GORMON, B., 'Official Definition of CORAL 66', HMSO (1970).

Discussion

Q. Why do you have no absolute delay?

A. The wait specified is absolute. There is no relative delay because it can be derived from an absolute delay.

C. In my applications, delay timing has not been critical. Swapping can cause timing problems, though.

Q. How do you deal with random access devices, e.g. disk or other backing store?

A. Not in the level of supervisor I have described, but at a higher level. (Similarly for CAMAC.)

Q. Does this also apply to shared devices such as shared typewriters?

A. Yes: the (reentrable) program to handle them would be written to use the supervisor facilities described.

C. I have found that this tends to cause rather high overheads. For this reason I would include the facilities in the basic supervisor, although conceptually they belong at a higher level.

Q. How do you have reentrable programs without dynamic storage?

A. Each task must take care of its own storage allocation.

Q. How do you wait for events?

A. By semaphores.

Software aspects of the UMIST hybrid

J.N. HAMBURY
Messerschmitt-Bolkow-Blohm GmbH, W. Germany

1. Introduction

The basis of the software developed for the hybrid computer was a field-proven, multi-access digital computer, the PDP-10. The software supplied by the computer manufacturer was extended to service multiple analogue computers, treated as advanced peripherals by the digital. The software was developed as a team effort by five different people, taking the approximate equivalent of $3\frac{1}{2}$-4 man-years. It was carried out during a period of about $1\frac{1}{2}$ years, but it is still being extended and improved.

The software implemented for the PDP 10/Cil75 hybrid computer can be divided into three general categories:

1. Extensions to the PDP-10 Time-sharing Monitor.
2. Hybrid packages to be used in the implementation of applications.
3. Diagnostic routines to check the performance of the system.

Item 2 includes three languages intended to assist hybrid problem set-up and testing. These were originally formulated during work on the previous Elliott 903/Cil75 hybrid [1], but were extended for the PDP-10 implementation.

The most important aspect of the hybrid software philosophy has always been to make problem development as easy as possible. The user must inevitably contend with a great deal of familiarisation and could not cope with the additional problem of critical real-time demands on a time-sharing system. Just as a multi-access terminal user appears to have the complete digital power available, each hybrid user must feel that he has independent access to the system and must be unaware of the complicated organisation involved in servicing the other hybrid users, displays and all the standard peripherals at the same time. It is very important that the user should avoid learning assembly language (as there are 366 instructions on the PDP-10). Thus a high-level language must be provided, without adding an unacceptable overhead to the solution time.

2. Hybrid software

The principle followed has been that a user programs the initial and terminal regions [1] (problem set-up and processing of results, respectively) in FORTRAN, supplemented with an additional library of Hybrid I/O Subroutines. These Subroutines operate by making I/O requests on a Hybrid Service Routine [2], which was added to the service routines (or drivers) for the standard peripherals in the monitor. The dynamic region (when the analogue computer is in compute mode and hybrid function generation may take place) is programmed partly with a set of macros and partly in FORTRAN. The former are required for I/O, timing and synchronisation between the analogue and digital computers, and the latter for calculations.

The I/O Subroutines (FORTRAN), Hybrid Service Routine and Hybrid Macros are shown as blocks in Fig. 1, which indicates the interconnections between the various packages involved in the hybrid software. The other packages fully implemented are the Hybrid Diagnostic and the Hybrid Interpreter, which is employed during problem testing.

A considerable amount of investigation has been devoted to Terminal Booking and Interface Allocation. Typewriter terminals and analogue computers are reserved on booking forms, where the type of use is indicated – hybrid, computer aided design (CAD) or program development. A limited number of configurations are available, because the research demands cause bottlenecks to be encountered (e.g. disk space) and it is impracticable to permit reservations to exceed the capacity of the system. Typewriter bookings are enforced by hardware connections adjusted by the operator, and disk space is restricted by software. An allocation of interface components, required for a particular problem, is reserved at the beginning of run-time by means of a conversational program (ALLOC), included with the Hybrid I/O Subroutines. Future development will enable this to be carried out earlier so that con-

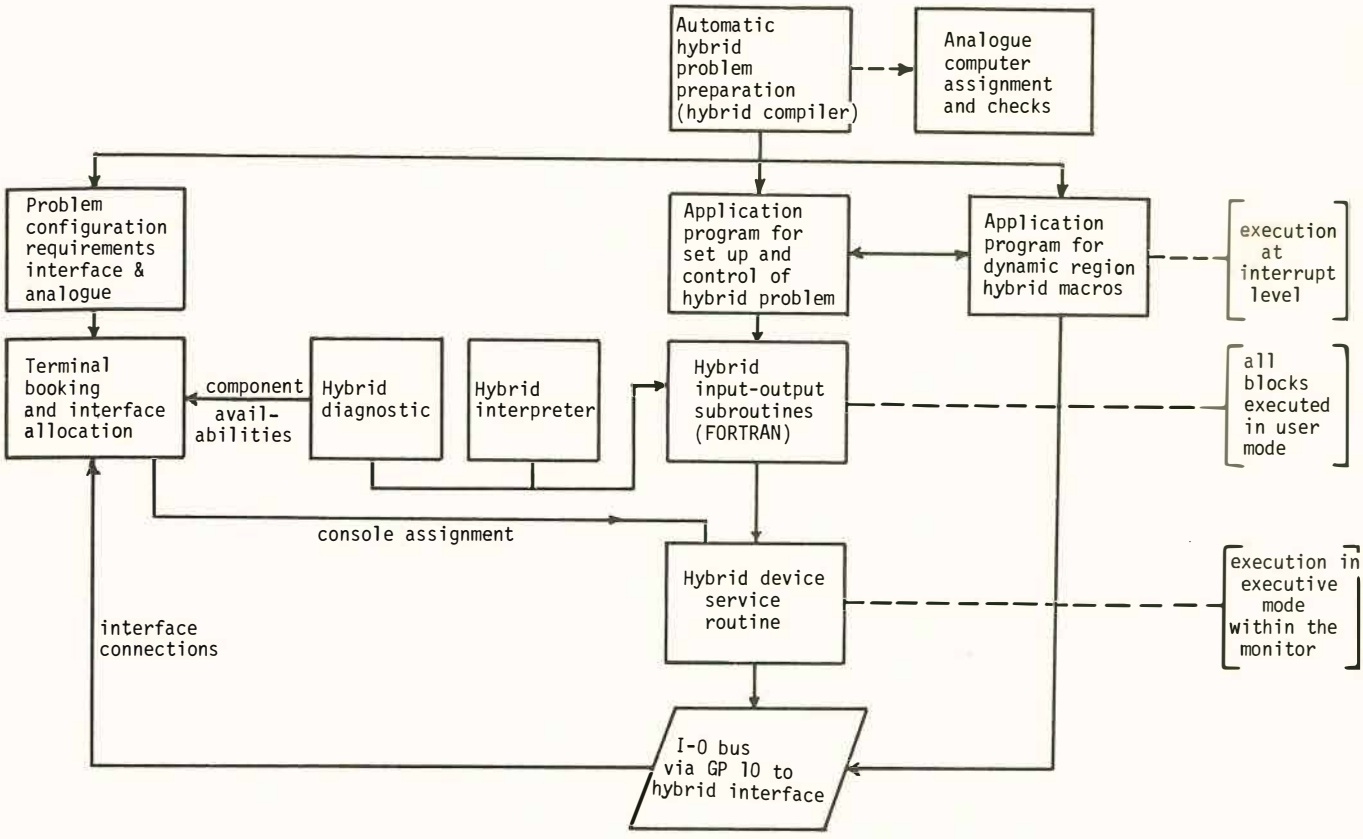


Fig. 1 Organisation of hybrid software

flicts do not arise.

Automatic Hybrid Problem Preparation (Fig. 1) has also been studied in collaboration with the British Aircraft Corporation. It was apparent that such a facility would save considerable preparation time, especially as the generation and execution of the hybrid object program (for set-up and checking) would be performed by the same digital computer, but too much support effort is required to enable the Centre to implement a hybrid compiler at present.

3. Priority structure

A flexible priority structure is provided by the PDP-10 digital computer. The levels are as follows:

1. 8 hardware interrupt levels.
2. Time-sharing monitor, including the hybrid service routine.
3. User jobs.

More than one device may be connected to each hardware interrupt level. It is possible to attach a special response routine, written by the user, to any priority interrupt level [3]. A request is made to the Monitor (by a user job) to connect a designated peripheral to its response routine, which is subsequently executed when an interrupt occurs. This technique is used to satisfy the critical response demands of the dynamic region.

The user jobs are not executed strictly in rotation, but are serviced in a variable manner according to their run-times and core occupancies. Since they are not critically time-dependent the initial and terminal regions are executed as user jobs and take their turns with CAD and compilation jobs.

4. Hybrid service routine

The function of this module of the Monitor is to perform non-critical, buffered transfers between the digital and analogue computers. It has been designed to conform, as far as possible, with the standard I/O procedures.

When an initial region job requests a number of transfers by calling the I/O subroutines, the latter transform the FORTRAN parameters into interface addresses and converter codes. The transfer is initialised, the addresses and codes are placed in buffers, which may be accessed by the Hybrid Service Routine (HYDSRN) and an input or an output request is made to the Monitor. This causes the execution of a section of code common to all peripherals, followed by a section of the HYDSRN, which sets up the transfer of the first entry in the first buffer by generating an interrupt. An interrupt is used to indicate the completion of

every individual transfer to the HYDSRN, which then performs the following transfer. This method was chosen, although the interrupt response time may exceed the hardware conversion time, for conformity with the standard procedures.

The HYDSRN implemented initially could only perform transfers to one analogue console at a time. Thus only one hybrid problem was running at any time, but it operated concurrently with all the time-shared terminals. This software enabled all the other hybrid packages to be developed and the first problems to be tackled. The initial HYDSRN was later extended to organise and queue concurrent requests for transfers to more than one analogue console. These cannot actually be performed simultaneously because only one hardware interface (GP-10) is used. An initial version of the multiple console service routine has been run. It is now being re-specified in the light of experience gained in preliminary trials and feedback from the use of the single console system.

The development of the two versions of the HYDSRN has been proceeding over a period of $1\frac{1}{4}$ years with only one systems programmer working on it at a time. This period has been so protracted because it included familiarisation with the design of the Monitor and joint hardware-software negotiations on the final design of the interface. Extending the Monitor presented a problem because of its complexity, although it is well documented, and could only be undertaken by experienced systems programmers (who were often required to assist with unrelated problems on the system).

The development of the HYDSRN was justified by the following considerations:

1. Conformity with the Monitor by the use of common I/O code and programming uniformity.
2. Servicing multiple non-critical hybrid jobs together.
3. Consistent control of the single interface and protection of transfers in EXEC mode.

It is also possible to perform transfers from a user job in USER I-O mode, a technique involved in critical operations during the dynamic region and in simple interface diagnostics. Both cases involve communication with only one analogue computer. It is conceivable that multiple console I/O could have been performed by a similar method, by placing queue handling in the re-entrant FORTRAN run-time library, but this would have sacrificed protection and conformity, without considerable simplification. An alternative possibility would have been to purchase a separate GP-10 interface for each analogue console, thereby avoiding the queuing problem and to program in USER I-O mode.

5. Hybrid packages

5.1 Hybrid input-output subroutines

These subroutines were used for programming the Hybrid Interpreter and Diagnostic, as well as the initial and terminal regions of problems. They are summarised in Table 1.

This form of definition has been found straightforward for familiarisation and application, and is consistent with the component identification on the analogue and logic patch panels. The

TABLE 1

Mode	Transfers		
	Logic (L)	Fast Analogue (A) (using ADC, DAC or DAM)	Slow Analogue (A) using DVM for input
RESET	RLTRUNK	RATRUNK	RAMP
HOLD	SLTRUNK	SATRUNK	RPOT
COMPUTE		SDAMPOT	RSUPPLY
POTSET			RTRUNK
RITER			SPOT
HITER			
TIMESCALE			
UNTIMESCALE			
ABBREVIATIONS:	R = read S = set		
PARAMETERS:	MODE Console number only		
TRANSFERS	Component numbers and values, number of transfers, console number		

principle of operation has been briefly described with the HYDSRN. A suite of dummy I/O subroutines has also been provided for problem testing.

The development of this package has been fairly straightforward as it was based on experience obtained on the previous hybrid. Hardware definitions were modified during implementation and all the programming was in assembly language. The complexity of the flowcharts was increased by the flexibility of the interface, since at every access to the transfer subroutines the component numbers in the parameters have to be checked for consistency with the components allocated.

5.2 Hybrid macros

An alternative method of communication between the digital and analogue computers was required for the dynamic region for the following reasons:

1. To obtain a fast response.
2. The I/O Subroutines provided generality, but incurred some overhead.
3. Programmed operators in the Monitor cannot be executed at the interrupt level.
4. A time-shared job (e.g. initial and terminal regions) may be descheduled.

The use of macros was also explored on the previous hybrid. It enabled an intermediate level language to be provided, with minimum overhead on execution time. This can readily be modified during development. For incorporation into a FORTRAN program, the macros are combined into subroutines. The macros implemented initially are shown in Table 2.

Other macros to be specified will deal with logic and mode control. An associated subroutine is required for execution before the critical part of the dynamic region to request critical operation from the HYDSRN, to lock the job in core and connect it to an interrupt level. Only one hybrid job may perform critical transfers at a time and all non-critical transfers must be suspended whilst it is in progress.

The macros have been applied satisfactorily, but it is proposed to simplify the source definitions, and possibly make them compatible with the I/O Subroutines. Although the dynamic region is programmed with macros the core occupancy is not excessive as the number of different transfers is small.

5.3 Hybrid interpreter

This package allows a user to issue one line commands to the system, without having to program CALL statements for the I/O Subroutines. Entry and execution of commands is performed without noticeable delay during time-sharing. A different command is provided for reading or setting each type of component. It is also possible to specify the problem variable names, and the corresponding component numbers and scale factors as parameters of the Interpreter. The problem may then be checked in terms of its own nomenclature and units.

The Interpreter was programmed in FORTRAN in about 3 man-months and has been most effective in problem testing.

TABLE 2

Macro	Function	Number of executable computer instructions
ADCIN (IFADD, DUPLEX)	Read pair of analogue values with interface address IFADD and store in location DUPLEX	6, but involves loop until conversion completed
DACOUT (IFADD, DUPLEX)	Set pair of analogue values in DUPLEX on DA converters at IFADD	2
TRANSI (DUPLEX, INTGLH, INTGRH)	Transform pair of analogue values in DUPLEX into unpacked integers and store (left-hand and right-hand values)	13
TRANSO (DUPLEX, INTGLH, INTGRH)	Packing for output (reverse of above)	15
CLRHYD	Clears hybrid interface conditions register	2
USUB (A, B)	Declares subroutine header, number A and external labels B	2
URET	Declares subroutine return	3

6. Hybrid diagnostic

This provides an interactive diagnostic package to check the performance of all analogue and interface components. Like the Interpreter, it is run during time-sharing. There is a total of 22 different tests, programmed as separate FORTRAN subroutines, divided between three pairs of patchboards. The tests associated with each patchboard may be run in sequence or demanded individually.

The diagnostic was also programmed in FORTRAN in a shorter time than the Interpreter, but much additional time was involved in specifying the diagnostic actions.

7. Conclusions

The production of the hybrid software for a multi-access system involved a considerable amount of investigation, including research and development, before designs could be frozen. Hence the implementation time was extended beyond that for an equivalent known system.

The use of a time-sharing system has been found very beneficial both for software development and for problem testing. After an initial short period when a number of monitor crashes were caused by hybrid development, computer-aided design users have scarcely been affected by the hybrid work. The latter has been able to proceed without severe pressure to minimise the amount of digital computer time used. Setting potentiometers under interrupt control is quite rapid and it has been possible to obtain a critical response without shutting down the time-sharing facility. During the development of the HYDSRN, however, the complete computer system has been required during all major testing work. It has therefore been necessary to devote the computer to system check out for two hours every day and occasionally for longer periods outside normal working hours. The time-sharing facility is withdrawn from the users at these times, when software and hardware specialists often work together.

The hybrid software is documented both for users and for system development. The packages have been applied to the hybrid simulation of the LD Steelmaking Process [4].

Acknowledgements

The software work described in the paper was designed and implemented by the following people:

Lynn Wiseman
K. McGill
W. Swindells
D. Utting

in addition to the author. Without a major contribution from everyone in this team (and smaller contributions from others), the system could not have been developed.

1. HAMBURY, J.N., 'Programming and testing hybrid simulations', Computer Bulletin, V15, p. 300 (August 1971).
2. HAMBURY, J.N., and BARNEY, G.C., 'The hybrid in control', AICA - IFIP Conference on Hybrid Computation, Munich (1970).
3. PDP-10 Reference Handbook, Section 3, 'Time-sharing monitors', Digital Equipment Corporation, pp. 153, 302 (1969).
4. MIDDLETON, J.R., BARNEY, G.C., and HAMBURY, J.N., 'A simulation of the LD steelmaking process using the UMIST facilities', UK Simulation Council Symposium (January 1972).

Discussion

Q. How much core store did you have, how big was the monitor and how much of that is locked?

A. We have a 48K store of 32-bit words. The monitor size is 20K, of which 3K is locked. These are the sizes we get without particularly trying to make it small.

Process scheduling by output considerations

G. McC. HAWORTH
University of Cambridge, England

Introduction

In multi-tasking systems when it is not possible to guarantee completion of all activities by specified times, the scheduling problem is not straightforward. Examples of this situation in real-time programming include the occurrence of alarm conditions and the buffering of output to peripherals in on-line facilities. The latter case is studied here with the hope of indicating one solution to the general problem.

Three parameters are associated with each job J_i ($i = 1,\ldots,n$). These are the processing time estimate p_i, the due-date d_i, and a positive weighting w_i. Three problem areas are distinguished:

1. The unweighted deterministic case (UDP) where p_i is the known processing time and $w_i = 1$ for each J_i.
2. The weighted deterministic case (WDP) where we have general weights w_i and known processing times p_i.
3. The stochastic case (SP), where the processing time of J_i is a random variable with an exponential distribution and mean p_i. General weights cause no difficulty in this case, but we are particularly interested in sub-optimal schedules as the optimal schedule cannot be derived entirely in algebraic terms.

Given a schedule S, J_i will be completed at time $c_i(S)$: the cost which we seek to minimise is

$$T = \text{Expectation}\left\{\sum_{i=1}^{n} w_i \times \max[0,\ c_i(S) - d_i]\right\}$$

and jobs are indexed so that

$$i < j \Rightarrow p_i w_i^{-1} < p_j w_j^{-1} \quad or$$

$$(p_i w_i^{-1} = p_j w_j^{-1},\ \ d_i \leqslant d_j)$$

The full derivation of the algorithms, which are original, can be found in [4].

The unweighted deterministic problem

The cost in the deterministic cases is minimised by sequencing the J_i without pre-emption or idle-time [1, p. 24]. When $w_i = 1$, the iterative algorithm on p. 76 computes the best sequence. This method is more convenient than the branch-and-bound approach of [2] but springs from the same source, namely

Th. UD1

$d_1 \leqslant \max(d_2, p_2) \Longleftrightarrow J_1$ precedes J_2 when $n = 2$ (Fig. 1)

Th. UD2

$i < j$, $d_i \leqslant \max(d_j, p_j) \Rightarrow J_i$ precedes J_j (for any 'n')

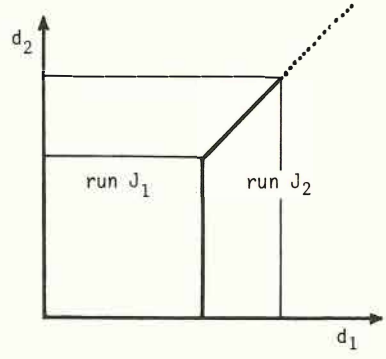
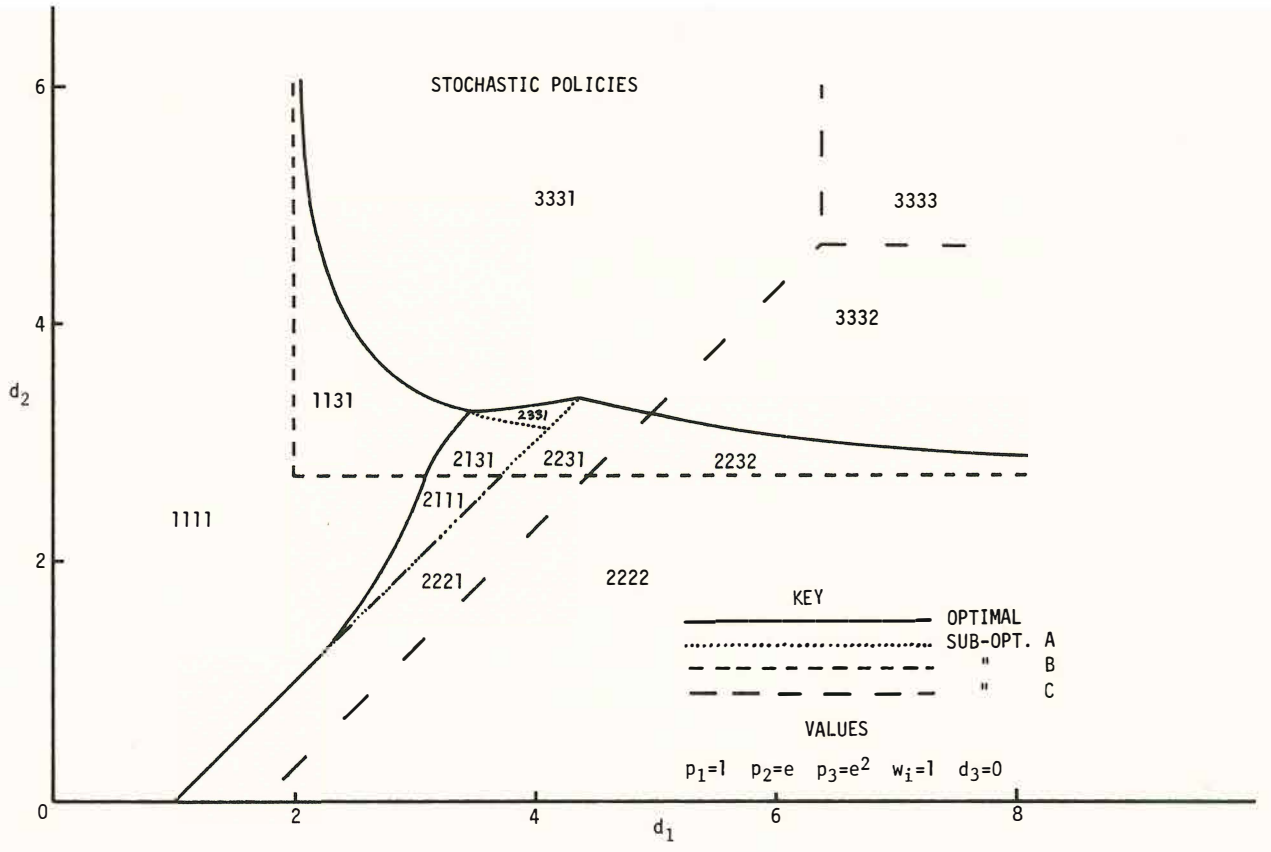


Fig. 1

An 'eligible' job is one which is not excluded from the next position by Theorem UD2. The algorithm forms the LEJ (longest eligible job) sequence by executing the longest eligible job at each stage. The LEJ sequence is certainly as good as the EDD (earliest due-date) and LST (least slack-time) sequences, and connot be improved by interchanging jobs adjacent in the sequence.

Algorithm for the UDP

```
global  proc, due, best sequence [1:n] ;
comment  'proc' and 'due' hold the processing time and due dates
         respectively of the jobs 1,....,n;

 {comment  this block computes the best sequence [1:n]  from
           proc, due [1:n]  for the unweighted deterministic problem;
  local  tard, done, seq [1:n] , best cost, deferred job,
         start of tail, in doubt;
  form LEJ sequence (1,n);  best sequence := seq;  best cost := cost;
  checktail;
  while  in doubt  do  {try to improve best sequence;  checktail}}

form LEJ sequence (start, limit)  ≜
   {comment  this fills places  start,...,n  with job indices ≤ limit;
    local  i, 1, pr, place;   pr := 0;
    if start ≠ 1 then for i := 1 to start-1 do pr := pr + proc[seq[i]];
    for place := start to n do
       {comment  this puts the longest eligible job in place;
        i := 1; while done[i] > 0 do i := i+1;  1 := i+1;
        while 1 ≤ limit and proc[1] < due[i] - pr do
           {if done[1] ≤ 0  and  due[1] < due[i] then i := 1; 1 := 1+1};
        seq[place]:= i; pr := pr + proc[i];
        tard[i] := max (0, pr-due i );
        if done[i] = -1  then  limit := n;  done[i] := 1}}


cost  ≜  Σ (i=1 to n)  tard[i]

check tail  ≜
   {comment  this looks for the shortest tail sequence that might
             not be in the optimal order;
    local  place, prefer;
    in doubt := false;  place := n;  prefer := n+1;
    while  not in doubt  and  place > 0  do
       {done[seq[place]] := 0;
        if  seq[place] > prefer  then
           {in doubt := true;  deferred job := seq[place];
            start of tail := place}
        else  if  tard[seq[place]] > proc[seq[place]]  then
                  {done[seq[place]] := -1;
                   if  seq[place] < prefer then prefer := seq[place]}
        place := place - 1}}


try to improve best sequence  ≜
   {comment  this defers the first job in the doubtful tail sequence;
    form LEJ sequence (start of tail, deferred job -1);
    if cost < best cost  then {best cost := cost; best sequence := seq}}
```



If $W = \{i \mid c_i(\text{LEJ}) - p_i > d_i, \exists j > i$ s.t. J_j precedes J_i in LEJ $\}$

then the minimal cost is not less than

$$T(\text{LEJ}) - \sum_{i \in W} [c_i(\text{LEJ}) - p_i - d_i].$$

Hence, $W = \phi$ is a sufficient but not necessary condition for the LEJ sequence to be optimal, and it is usually satisfied.

If $c_i(\text{LEJ}) - p_i \leq d_i$ for all i, then LEJ coincides with EDD and is optimal. Thus the EDD rule, which is known to be best if each J_i can be completed by d_i, is seen to have far wider optimal properties.

The algorithm has been run on collections of 100 job-sets for various 'n', with p_i, d_i random in $[0, 1]$ and $[0, \frac{1}{2}n]$ respectively. For $n = 4$, $W = \phi$ 95 times (out of 100) and LEJ was sub-opt. 3 times.

For $n = 8$, $W = \phi$ 77 times and LEJ was sub-optimal 7 times.

For $n = 16$, $W = \phi$ 28 times, LEJ was sub-optimal 26 times, and on average, the algorithm performed only two iterations after the initial LEJ.

The weighted deterministic problem

The addition of general weights w_i complicates the problem considerably, and no method matching the efficiency of the UDP algorithm is possible. One method [5] has been proposed, but it is unwieldy for our purposes and has been shown to be sub-optimal by a three-job case study [1, pp. 46-8].

Corresponding to Th. UD1, we have, when $d_1, d_2 \leq p_1 + p_2$

Th. WD1 $\quad d_1 \leq \max[p_2 + (1 - w_1^{-1}w_2)p_1,$

$$w_1^{-1}w_2 d_2 + w_1^{-1}(w_1 - w_2)(p_1 + p_2)]$$

$\Leftrightarrow$ J_1 precedes J_2 when $n = 2$ (see Figs. 2, 3)

but this does not lead to the analogous form of Th. UD2.

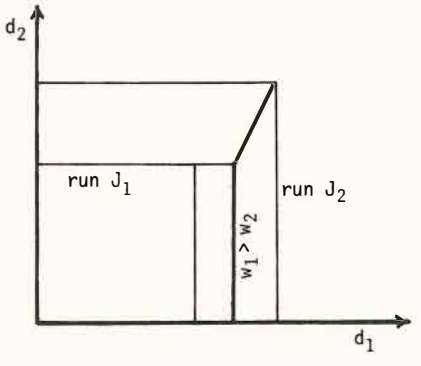


Fig. 2

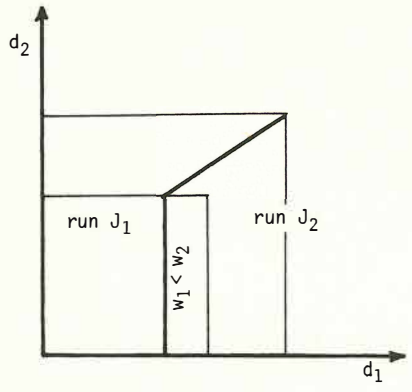


Fig. 3

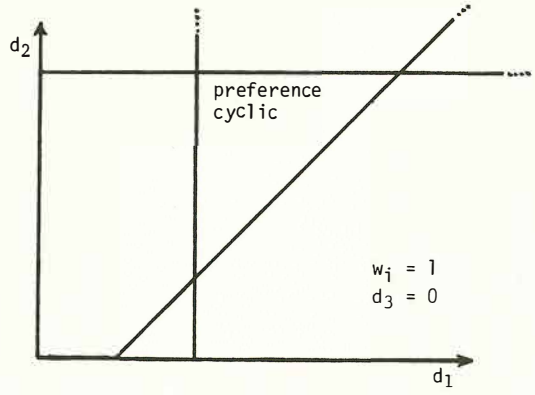


Fig. 4

The current goal is to find an algorithm which will reduce to the UDP algorithm of p.76 when we set the general weights to 1. In this direction we have:

Th. WD2 $\quad i<j<k$: $d_i \leqslant p_j + (1 - w_i^{-1} w_j) p_i$

$$\Rightarrow \quad d_i \leqslant p_k + (1 - w_i^{-1} w_k) p_i$$

analogous to $\quad d_i < p_j \Rightarrow d_i < p_k$

It is a remarkable fact that general weights are more easily included in a stochastic study than in the deterministic study, and we can for the moment only ape the LEJ sequence, leaving the question of suitable optimality criteria and an iterative procedure open.

```
global  p, d, w, seq[1:n];
local   done[1:n], i, ℓ, place, pr ;
pr := 0;  for i := 1 to n do done[i] := false;
for place := 1 to n do
   {i := 1;  while done[i]  do i := i+1;  ℓ := i+1;
    while ℓ ≤ n  and w_i p_ℓ + (w_i - w_ℓ) p_i < w_i (d_i - pr)  do
       {if not done[ℓ] and  w_ℓ (d_ℓ - pr) + (w_i - w_ℓ)(p_i + p_ℓ) < w_i (d_i - pr)
            then i := ℓ;
        ℓ := ℓ + 1  }
    seq[place] := i; pr := pr + proc[i];  done[i] := true  }}
```

The dynamic programming principle of optimality gives the equation

$$-\dot{C}(S,t) = e_S(t) + \min_{i \in S} p_i^{-1} [C(S_i,t) - C(S,t)]$$

where $S \subset \{1,\ldots,n\}$; $S_i = S - \{i\}$;
$Z = \{i \mid i \in S,\ d_i \leqslant t\}$; $e_S(t) = \sum_{i \in Z} w_i$ and $C(S,t)$ is the expected additional future cost at time t, given that we follow the optimal policy and have to schedule the jobs J_i, $i \in S$.

It is useful to know that the form of the cost function is

$$C = l(t) + \sum_{i=1}^{n} a_i(t) . \exp(t/p_i)$$

where the coefficients $l(t)$, $a_i(t)$ are constant in half-open intervals $[c,d)$. There is only a finite number of these intervals.

The stochastic problem

We add conditions of uncertainty to the problem in the most convenient way by assuming that the processing-time of J_i is a random variable exponentially distributed with known mean p_i. The probability of J_i terminating does not then depend on the time for which it has already run, and so this simplification leads to a Markov system. More general probability distributions and cost functions are treated in [3] for the case $n = 2$.

The result for $n = 2$ is

Th.S1

$$d_1 \leqslant d_2 + p_1 \log_e(w_1 p_2)/(w_2 p_1) \iff \text{run } J_1 \quad (J_1 < J_2)$$

Note that with three jobs, the situation $J_1 < J_2$, $J_2 < J_3$ and $J_3 < J_1$ may arise! (See Fig. 4.)

The best-decision areas for $w_i = 1$, $d_3 = 0$ and $p_i = \exp(i-1)$ are depicted in the graph on p.77. It is conjectured that the decision at time $t \leqslant d_3$ is the decision at $t = d_3$ for $n = 3$, in which case the graph is sufficient representation of the solution.

Since numerical techniques are involved in finding the decision areas, we turn to more convenient sub-optimal policies for implementation.

Sub-optimal stochastic policies

The graph on p.77 compares the optimal decision rule with three sub-optimal rules. The quartet '$ijkl$' in each region means "choice 'i' with optimal rule, choice 'j' with rule A, 'k' with B and 'l' with C".

A. If we insist that when we switch from J_j to J_i using this rule, J_j is the best choice in the absence of J_i, and vice versa, then the boundaries can be calculated algebraically. Where they differ from those of the optimal policy and those of policy B below, they are marked by dotted lines. Figure 5 indicates how the order of preference of the jobs at any one time changes in a simple case.

B. With $i < j$, define $J_i > J_j$ as

$$d_i - d_j > p_i \cdot \log_e[(w_i p_j)/(w_j p_i)]$$

$$Z = \{j \mid J_i > J_j \ \forall\ i<j\} \ \exists\ k \in Z \quad \text{s.t.} \quad k \geqslant j \quad \forall j \in Z$$

The decision of rule B is to run J_k, and this is analogous to the LEJ sequence in the deterministic case.

C. $p_i + d_i$ acts as an index when all $w_i = 1$. With rule C, we run J_k where $p_k + d_k \leqslant p_i + d_i \ \forall\ i$.

D. Also when the $w_i = 1$, we might apply the deterministic rule. This is not shown on the graph as it simply divides the area depicted between J_1 and J_2.

Rule B is the best for the purposes of scheduling in real-time systems as it is applicable with general weights, quick to calculate and in close agreement with the optimal decision rule. In many applications, including the one studied in the next section, the other modelling assumptions being made will make the difference between rule B and the optimal rule insignificant.

One application: an on-line facility

The primary purpose of processor scheduling in an on-line system is to maintain the illusion for each user that the computer is dedicated to him. Most of the scheduling algorithms proposed and implemented are based on resource considerations only such as time quanta, time already spent on a process, or – if the process is not represented in core – storage requirements.

The illusion mentioned above will be achieved if all output channels are kept busy, and the proposal of this section is that we can come closer to that goal by monitoring the output performance of each process.

The on-line system modelled here is one in which 'n' users U_i at separate consoles have each submitted a task J_i. The number 'n' varies with time, but we make no attempt to anticipate the arrival of new tasks: later arrivals pre-empt the currently running process if necessary. An amount $d_i(t)$ of output has been created by J_i but not received by U_i, and this backlog, measured by the time it will take to clear, gives us a due-date for J_i. A further time $d_i(t)$ may elapse before J_i produces more output to maintain the continuity of the data stream. p_i is an estimate of the time required by J_i to produce a unit of output, and a variety of simple heuristics are available to decide this parameter. The weight w_i might represent an amalgamation of factors such as the status of U_i, the time already received by J_i and the output already received by U_i.

The stochastic model with sub-optimal policy 'B' seems appropriate given the approximations above. Since $\underline{p}$, $\underline{w}$, $\underline{d}$ and 'n' are all functions of time, the algorithm must be quantised in some way to maintain CPU utilisation and control housekeeping overheads.

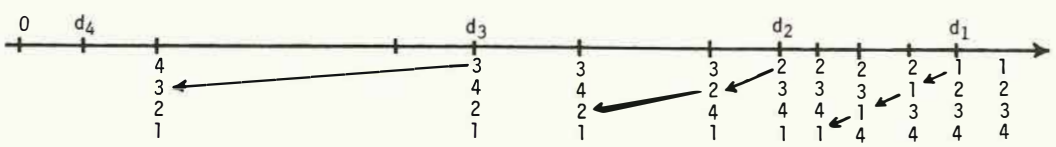


Fig. 5 Illustrating how the preference order can change with sub-optimal strategy A in the stochastic case. The calculations are done in backwards time.

Acknowledgement

I should like to thank Professor J.F. Coales and Dr J.D. Roberts for supporting this contribution to the Seminar, and P. Nash for the stimulus of a continuing dialogue on the ramifications of the stochastic problem.

1. CONWAY, R.W., MAXWELL, W.L., and MILLER, L.W., 'Theory of scheduling', Addison-Wesley Publishing Corporation (1967).
2. EMMONS, H., 'One machine sequencing to minimise certain functions of job tardiness', Operations Research, V17, pp. 701-715 (1969).
3. GITTINS, J.C., 'Optimal resource allocation in chemical research', Advances in Applied Probability, V1, pp. 238-270 (1969).
4. HAWORTH, G.McC., 'Tardiness scheduling and computer systems', Cambridge University Engineering Department Technical Report CUED/B-Control/TR22 (1972).
5. SCHILD, A., and FREDMAN, I.J., 'Scheduling tasks with deadlines and linear loss functions', Management Science, V7, pp. 280-285 (1961).

Discussion

Q. Is the algorithm for the Weighted Deterministic Problem linear?

A. No, the execution time is proportional to $N \log N$. (Similarly for the Unweighted Deterministic Problem.)

Q. What happens when a new job is added?

A. Its priority has to be decided, and then the algorithm used to determine the schedule for the new set of jobs.

Q. How does your method compare with a pure priority scheme?

A. It works as though each job has a gradually increasing priority.

CALAS70 – a real-time operating system based on pseudoprocessors

G. HEPKE
Kernforschungszentrum Karlsruhe, W. Germany

1. Introduction

The changing prices of minicomputers mark one direction in which real-time systems will develop: cheap computers will be used in a decentralised manner to solve individual problems. This solution covers quite a number of applications for real-time systems; other applications, however, call for new central systems, to be developed satisfactorily in the future. The concept and design of central systems, however, may profit from the decentralised solution. The processes* to be controlled and computed by the system presented here are long-term scientific experiments. The creative man carrying out the experiments requires a high degree of convenience, and that means a time-sharing system.

2. Principles of system design

The system design is based mainly on:

1. Specific requirements of experimental operation.
2. Advantages of decentralised systems.
3. Precise knowledge of the task profile.
4. Facilitation of implementation, testing, and assembling of system elements.

The specific requirements of experimental operation include, for example, rapid data acquisition and transfer to tape and/or disk, long-term data management, and programs for interactive work with graphic displays.

The advantages of decentralised systems are simplicity, easy handling and programming, reliability of system software, and independence.

A well known task profile allows time and storage saving restrictions by dedicated programs. The system architecture reflects the natural static and dynamic modularity of the tasks of the system and thus facilitates the management of the system.

*i.e. technical processes in this context.

3. Pseudo-processors as system modules

The elements of the system are pseudo-processors which can be considered to be independent dedicated minicomputers. The operating system can be taken as a network of pseudo-processors which are coupled via common core storage; cooperation takes place through special commands.

Each pseudo-processor is represented by a set of registers, a task queue and a program with variables for processing these tasks. As a rule, each task is characterised by a small number of parameters for setting the respective pseudo-processor, and that means programming the program.

By using only one processor and a restricted core memory, the independent pseudo-processors become dependent on each other with respect to time.

The multiplexing of the processor as well as the adaptive processing of the queues are subjects of Mr Herbstreith's paper [1] (see p. 84 of this book).

4. Peripheral devices and pseudo-processors

All activities of the system are integrated into the concept of the pseudo-processors. System and experiment periphery are embedded in pseudo-processors in order to be capable of general cooperation. Thus, the task for a peripheral device is no longer different from tasks for other pseudo-processors; the peripheral device is directly accessible to the user who need not tackle its specifications as check and control bits. Nevertheless, he is directly connected to the hardware of the device, and is able to state as task parameter the interrupt subroutine programmed on its own.

5. States of pseudo-processors and system

In order to increase the transparency and to decrease reaction time and failure rate of the

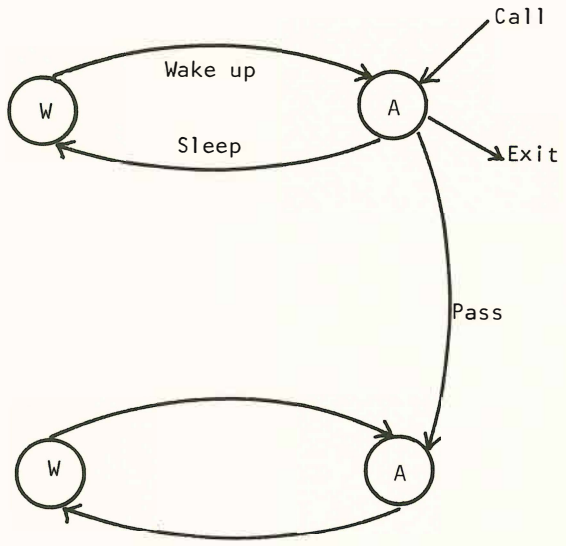


Fig. 1 States of the pseudo-processors

system, the number of relevant states and changes of states must be kept as small as possible. The same is true for commands acting on these states.

The relevant states of the pseudo-processors (see Fig. 1) are:

1. Waiting, for activation through a task or reactivation of a task already started.
2. Active, i.e., able to work or working.

The relevant changes of state for the integral system are:

1. Upon definitive or provisional termination of a task the next active task must be searched for processing.
2. Upon activation and reactivation, respectively, of a more urgent task than the task being processed, the more urgent task is started at once.

6. Communication, control commands

The pseudo-processors are independent units executing processes of specific types defined by tasks. Since, in general, a job includes the sequence of different processes, commands must be available for cooperation between processes and for sequence control.

By CALL a process gives a task to another pseudo-processor. Significant parameters of this command are the name of the pseudo-processor addressed and a task identification which distinguishes up to 23 tasks given by only one single process.

By SLEEP a process changes over to a preliminary state of rest ('waiting') in order to synchronise with other processes initiated by it. Parameters of this command are the task identifications of the corresponding tasks.

By WAKEUP the possibly 'sleeping' initiator of the waking-up process is woken up (set 'active').

By EXIT a process comes to an end and thus the task clears the pseudo-processor. EXIT may imply an automatic SLEEP or an automatic WAKEUP. This mechanism, which is installed as a safety precaution, may block a whole line of pseudo-processors. This blocking can be avoided by use of the PASS command.

By PASS a process transfers a task and is brought to an end unconditionally. This is possible since the 'obligation to wake up' is transferred to the new task.

7. Prevention of deadlocks

The task profile of the system allows fulfilment of all jobs by cooperation of few processes. This is important when deadlocks are to be excluded *a priori*. All system resources available for the users are pseudo-processors, which, in general, cannot occupy each other, but cooperate with equal rights in a dynamic mode through task and receipt. Nevertheless, all conditions leading to a deadlock can be satisfied [2]. However, one of them can be excluded *a priori* in this system by the 'circular wait condition'. The only prerequisite consists in checking the individual short task chains for loops.

8. Concluding remarks

The fundamental elements of CALAS70 have been working for more than one year. The system was improved steadily by extension and addition of pseudo-processors. Major difficulties have not occurred.

Experience gained under the test conditions used so far – hardware simulation of experiments – has been very satisfactory.

Further improvement now calls for measured results furnished by the real system or by simulation models, so that an even better static and, above all, dynamic adaptibility of the system can be achieved.

The small number of control commands and system states guarantee fast changes of state, good software reliability and system transparency. The given design has been described with reference to a system developed by the Institut für Datenverarbeitung in der Technik (IDT) of the Kernforschungszentrum Karlsruhe.

The concept of pseudo-processors has stood the test.

1. HERBSTREITH, H., 'A dynamic adaptive scheduling scheme for a real-time operating system', Proceedings of the 2nd European Seminar on Real-Time

Programming – Computing with Real-Time Systems, Vol. 2 (this issue), p. 84 , Erlangen (March 1972).
2. COFFMAN, E. G., et al., 'System deadlocks'. Computing Surveys, V3(2), (June 1971).
3. HERBSTREITH, H., and HEPKE, G., 'Ablaufsteuerung für ein System mit einfacher Hardware Struktur', KFK-Bericht 1530, Gesellschaft für Kernforschung Karlsruhe (1971).

Discussion

Q. When a job is waiting it can be awakened by a REPLY from a subtask or by a CALL from the Scheduler. Does it follow from this that there are two waiting states?

A. No, there is only one waiting state.

Q. Are pseudo-processors real machines?

A. No, they are virtual machines.

Q. What 'safety precautions' are there?

A. A pseudo-processor must wait until all its subtasks are finished. Each created task does its own book-keeping.

Q. Which computer is this implemented on?

A. Telefunken TR86 with 64K words of store, in which this software takes 10K.

A dynamic adaptive scheduling scheme for a real-time operating system

H. HERBSTREITH
Kernforschungszentrum Karlsruhe, W. Germany

1. Introduction

This paper is intended to extend and complement the paper by G. Hepke [3]. It describes some design criteria and the realisation of adaptive scheduling strategies in the multiprogramming real-time system CALAS70. At an early stage in the design of the system [2] we noticed that common strategies could not be realised in the same way as in an interactive time-sharing system [1]. The reason was that the term response time in process control systems generally has another meaning from that in other systems. This forces the designer to over-design the system capacity, to avoid process failure as a result of bad observance of the required response times. On the other hand, one has to compromise for an optimal utilisation of the unused system capacities with regard to changing load distribution as encountered in laboratory automation work. The effect of the so-called background tasks on real-time requirements was eliminated by the strategies to be described below. As far as scheduling is concerned, a further requirement is the realisation of response times of the order 300-500 μsec.

2. System concept

The internal system structure is based on Dijkstra's concept of "Hierarchical Ordering of Sequential Processes" [4]. It consists of a fixed number of pseudo-processors, created during system generation. These processors execute the sequential processes. It is possible to differentiate between four classes of pseudo-processes, according to their responsibilities for the following tasks:

1. System services (e.g. I/O functions, main storage management, timer, statistics, etc.).
2. Real-time tasks.
3. Man-machine communication.
4. Non-time-critical background work.

Each pseudo-processor is identified by its ID-number. The real processor is allocated according to a second number, the priority number. The distribution of the different tasks among the pseudo-processors of classes (1), (3) and (4) follows a fixed scheme, depending on the configuration. As will be pointed out below, certain system functions require two pseudo-processors with unequal priorities. Pseudo-processors of class (2) are allocated dynamically by the scheduler. The non-time-critical background tasks are performed by only one special pseudo-processor, which executes the requesting processes on a round-robin basis with fixed time slices.

3. Scheduling strategies

For the description of the implemented scheduling concept, it seems convenient to use the distinction between two main levels of scheduling, the short-term and the medium-term scheduling, made by Saltzer [5]. Only short-term strategies are considered here.

Short-term scheduling

At this lower level of scheduling (also called hardware management) the tasks to perform are:

a. Allocation of a pseudo-processor to each sequential process.
b. Management of changes of state of the pseudo-processors.
c. Making available a set of primitive basis functions for communication between processes.

The allocation of the real processor, according to the ready list of pseudo-processors, is managed by the dispatcher. The ready list has a two-dimensional organisation and makes the selection of highest priority processes as fast as possible. One dimension, the vertical, represents the pseudo-processors. In the horizontal dimension the ready processes are queued with respect to

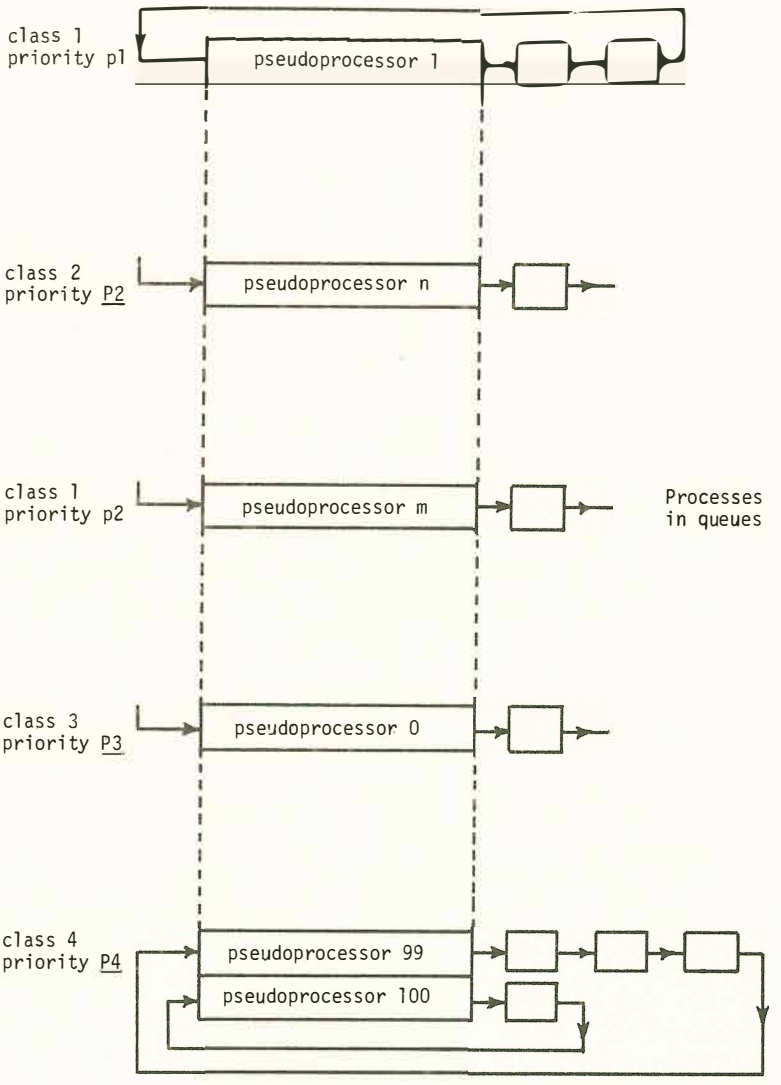


Fig. 1 Ready list of pseudo-processors

their priorities. These two degrees of freedom for scheduling allow a reasonable distribution of system capacities among the different tasks (Fig. 1).

Process queuing (Fig.2)

On creation of a process, the priority of that pseudo-processor which creates the process is assigned to it. The process will then be inserted in the corresponding queue of its pseudo-processor. The insertion is performed, according to the priority, and the look-up is started at the tail of the queue, with the restriction that a possibly already-active process at the head of this queue will not be moved by the insertion. By this method the processes with the highest priorities are always

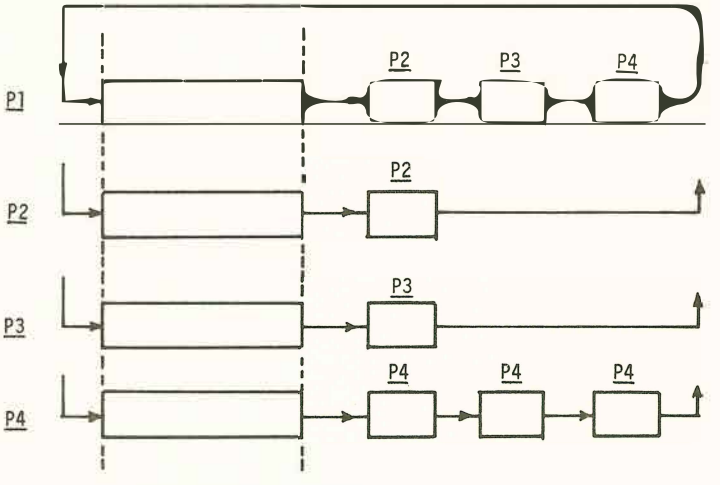


Fig. 2 Assignment of priorities to system processes

at the head of the queue and are therefore executed first. Thus, processes generated from classes (3) and (4) (Section 2) do not disturb the time sequence of processes generated from the real-time class (2).

To accomplish fast response times of the system, the scheduling functions have to be executed as fast as possible, since the real processor cannot be interrupted during the execution. As an example, we note that the necessary processor time for the insertion of a new member into a queue increases with the length of the queue. Assuming that, on an average, ten members will have to be checked, this time should not exceed 300-500 μsec. This assumption is sufficient, and so the length of all queues connected to class (1) will be reduced in the following way:

a. If the length of the queue exceeds 15, all processes generated by classes (3) and (4) are rejected.
b. If the length still exceeds a value of 20, the lowest priorities of the real-time class are rejected too.

Assignment of pseudo-processors to system service processes

As already mentioned, there are two pseudo-processors available for each of some special dedicated system service functions of class (1). The corresponding processes in the queues are assigned dynamically to either one or the other pseudo-processors, depending on the priority of these processes. If $\underline{P}_k$ (with $k = 2, 3, 4$) are the priorities of classes k respectively, and p_1, p_2 the priorities of the special pseudo-processors of class (1), then (Fig. 3)

$$p_1 > \underline{P}_2 > p_2 > \underline{P}_3 > \underline{P}_4$$

The pseudo-processor with priority p_2 is always executing processes as long as there are no processes generated from class (2). On creation of processes of class (2), the pseudo-processor with priority p_1, will be switched on until all these real-time tasks have been executed.

Three essential actions are required by the processor switching:

a. The last active processor is removed from ready list and becomes invisible for the dispatcher.
b. The status of the now blocked processor is copied into the register field of the other processor.
c. The new processor to activate is entered into the ready list and the corresponding process queue is assigned to it.

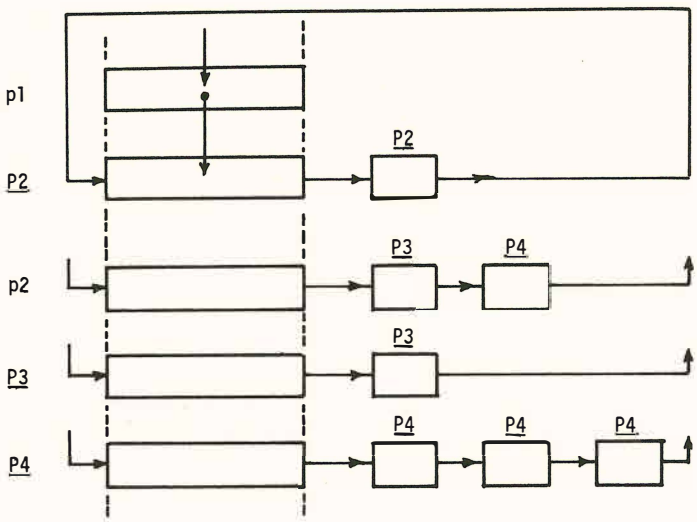


Fig. 3 Allocation of pseudo-processors p2 to system processes

4. Conclusion

By the use of these three scheduling strategies we have been able to develop standard solutions for the design of a real-time scheduler. We have shown the possibility of isolating real-time work from background tasks with a parallel decrease in response time. It is our belief that the methods discussed here improve the design of central multi-access systems in process control and laboratory automation. Our concept allows for the introduction of additional background work into the system, without deterioration of real-time requirements.

1. CHAMBERLIN, D., 'Comparative modelling of time-slicing and deadline scheduling for interactive systems', RC 3378 (15410), Research Report of the IBM Thomas Watson Research Center, Yorktown Heights, New York 10598 (25 May 1971).
2. GAGEL, G., HEPKE, G., HERBSTREITH, H., and NEHMER, J., 'CALAS68 – ein computergestütztes Vielfachzugriffssystem zur Laborautomatisierung', Ext. Bericht Nr. 19/69-1 der Gesellschaft für Kernforschung Karlsruhe (November 1970).
3. HEPKE, G., 'CALAS70 – a real-time operating system based on pseudo-processors', Proceedings of the 2nd European Seminar on Real-Time Programming – Computing with Real-Time Systems, Vol. 2 (this issue), p. 81 , Erlangen (March 1972).
4. DIJKSTRA, E.W., 'Hierarchical ordering of sequential processes', Intern. Seminar on Operating System Techn., Belfast (August 1971).
5. SALTZER, J.H., 'Traffic control in a multiplexed compiler system', MAC-TR-30, MIT, Cambridge (July 1966).

Workshop on components of real-time systems

Chairman: **Prof. Dr R. BAUMAN**
Munich, W. Germany

Discussion was concerned with the use of priorities for controlling the task scheduling and working space allocation (how dynamic it has to be). It was generally agreed that a real-time system differed from a general time-sharing system in the attitude of the users:

J. Ours is not a time-sharing system in which users may be trying to break the system. We can assume they are cooperating on the solution of a problem.

Priority structures

L. There is a golden rule: the priority of a process varies inversely as the critical response time for that process. Demands for very short response time may call for activation of lower priority processes. As an example, a block of disk transfers generates demands on at least 3 levels of priority:

1. The demand for individual word transfer.
2. The interrupt to identify the end of the block.
3. The scheduling of the process which will use the information.

The level of priority decreases from 1 to 3.

G. Hardware interrupts are handled at the priority needed to deal with the necessary immediate action only. A software interrupt is then scheduled at a lower priority to deal with the less urgent part of the action. In general the higher priorities are used for the minimum of time to handle the most urgent part of a response, the rest of the processing being scheduled at a lower priority.

J. Is it not true that the author of a task is the best one to assign its priority, since he knows how important his job is. (This is not a time-sharing system for user-programmers.) We insist that programs suspend themselves regularly stating their own priority, e.g. low, high, resume after 5 secs.

Working space allocation

S. In relation to Dr Pyle's paper I understand that what he is proposing with regard to store allocation is as follows: store is allocated in a static manner at load time for each process but can be allocated dynamically within a process.

Q. What is 'load time' in a real-time system?

A. When you first set the program up to run.

M. The choice of storage allocation mechanism depends very much on the type of application. The term 'real-time' is used to cover a multitude of sins from a simple data gathering system which has fixed storage requirements but relatively fast response times, to an Airline reservation system which needs sophisticated algorithms for storage but not particularly demanding response times.

Y. Cases can be made for dynamic storage allocation, for example a free store area containing a pool of blocks of fixed size (the size agreed between the designers of the processes using this pool). However, even with this in a system, the whole store is partitioned in a fixed way at load time between those areas used for program storage, those used for genuine static storage and those used for stacks or buffer pools. The management of the areas within which storage is used dynamically is done by a higher level program at a level of sophistication above that of the primitives given.

P. Dynamic allocation of working space (administered by the operating system) can give efficient use of store and easier implementation of reentrant tasks. At RRE we implemented message passing in an operating system. The mechanism provides message communication between parallel processes, which simplifies the problems of controlling program interactions. We found that the timing overheads in acquiring free areas and sending them to other processes were rather high.

We are now in the process of implementing a hardware version of this system. The hardware, which will be a computer peripheral, will enable us to acquire free area and send messages to other processes in time of order 100 nsecs.

Y. We may contrast the situation when the time a piece of program will take to execute can be determined, perhaps as a conditional expression involving conditions not removable in advance (e.g. a program containing a test on character read in and appropriate action, or a program containing an 'attempt to secure' primitive), with the situation when that time cannot be determined even to that extent, and is essentially unbounded (e.g. a program containing a 'secure' primitive, or a program in an environment using dynamic storage allocation where there is the possibility that store required may not be immediately available).

I regard the former situation, of conditionally determinable and bounded time, as acceptable for real-time systems, and the latter, of unpredictable unbounded time, as not acceptable.

SPECIAL SYSTEMS AND SYSTEM FEATURES

Chairman:
Dr H-J. TREBST

Simulation du déroulement logique de programmes de commande de spectromètres à neutrons

Ph. BLANCHARD, Ph. LEDEBT and M. TAESCHNER
Institut Laue-Langevin, Grenoble, France

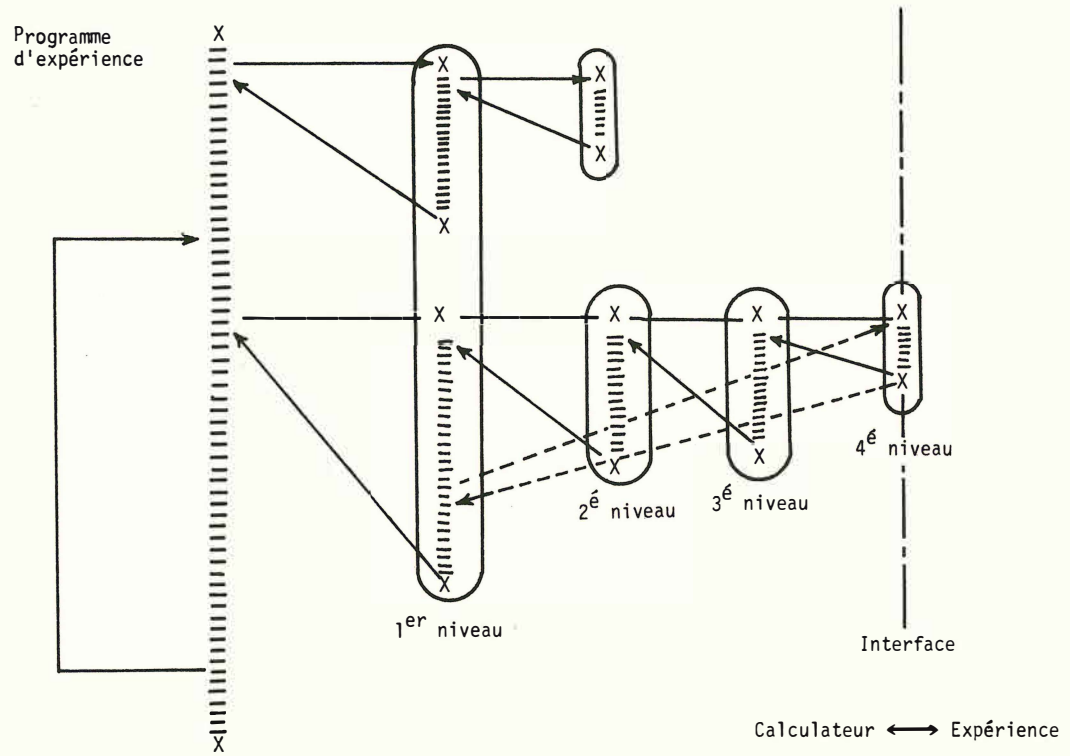
1 – Generalités – Introduction

Ce système de simulation est supporté par un terminal 2741, connecté sur le matériel IBM 360/67 de l'Institut de Mathématiques Appliquées de l'Université de Grenoble. Ce calculateur fonctionne en time-sharing, avec le système CP/CMS, basé sur l'utilisation des machines virtuelles en mode conversationnel. Chaque utilisateur a donc à sa disposition, un calculateur virtuel, type 360/40 avec imprimante virtuelle lecteur cartes/perfo cartes virtuel, etc.

Nous allons regarder maintenant le système réel de gestion des expériences.

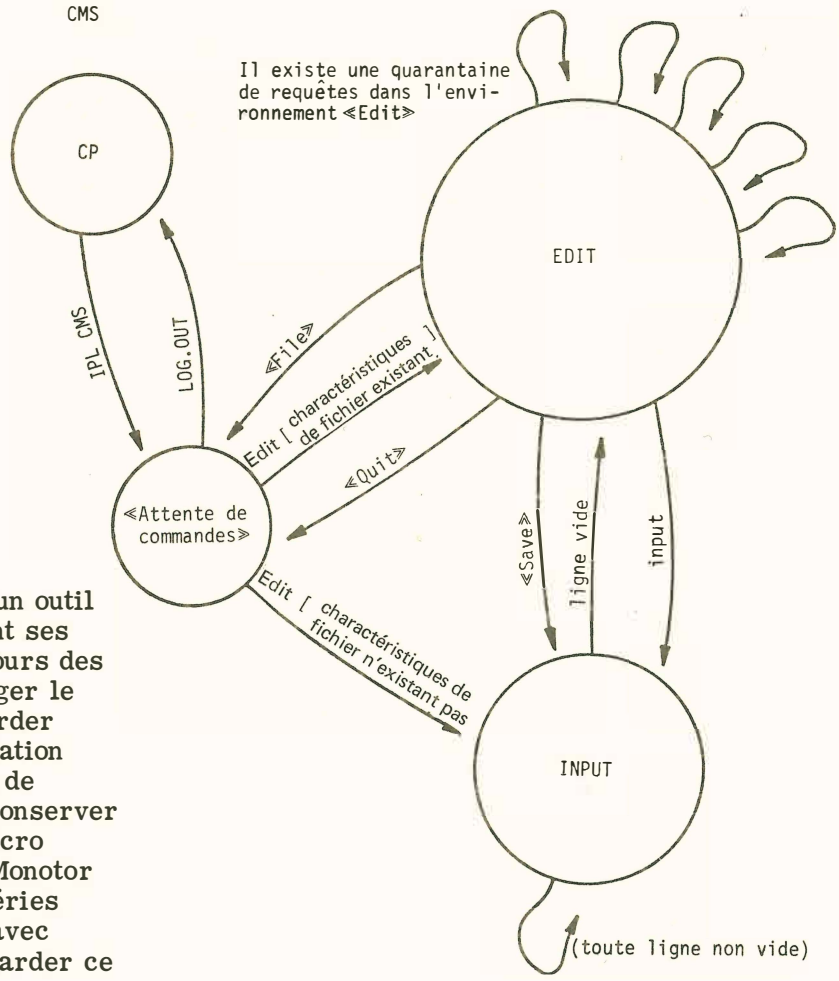
Description du Système Carine

Chaque calculateur de processus gère six spectromètres par l'intermédiaire de programmes FORTRAN temps réel. Une bibliothèque de sous-programmes assembleurs, gère les entrées/sorties, et chaque sous-programme peut être appelé par le FORTRAN. Chaque utilisateur dispose auprès de son expérience, d'un télétype pour démarrer, contrôler, arrêter ses programmes et suivre ainsi l'évolution de sa manipulation. Les transferts d'information calculateur-expérience et vice versa, sont assurés par l'intermédiaire d'un standard électronique, le CAMAC.



Diapositive 1 Diagramme général

Notre but était de fournir à l'utilisateur un outil lui permettant de créer et mettre au point ses programmes d'expériences, sans le secours des calculateurs de processus, afin de soulager le background de ces systèmes. Afin de garder toute son efficacité, le système de simulation garde la même structure de dialogue, et de possibilités que le système réel. Pour conserver cette structure, nous avons utilisé le macro langage de commande CMS (Cambridge Monotor System), qui permet de remplacer les séries d'ordre par une seule commande EXEC avec paramètres. Nous allons maintenant regarder ce qu'est le système CP/CMS.



Diapositive 2 Système CP/CMS

11 – Système CP/CMS

CP: Le système générateur de machines virtuelles CP 67 a été conçu et développé par le Centre Scientifique IBM de cambridge en collaboration avec le Lincoln Laboratory du M.I.T. La version initiale a été mise en service en Avril 1968. CP 67 permet de faire fonctionner sur un 360/67 un nombre quelconque de machines virtuelles qui sont des 360 standards. La mémoire centrale de cette machine virtuelle peut être plus grande que celle du 360/67 utilisé. L'activation d'une machine virtuelle se fait à partir d'une console terminale connectée au 360/67 grâce à son nom et son mot de passe. Une fois la machine activée, un système de programmation est chargé en mémoire virtuelle par utilisation du bouton (simulé) 'chargement'. A partir de ce moment, l'utilisateur dialogue directement avec le système qu'il vient de charger. Dans le cas qui nous concerne, nous chargeons un système CMS, mais d'autres systèmes sont disponibles. (APL - OS - etc.).

CMS: Cambridge Moniteur System). Le système CMS donne à l'utilisateur la possibilité d'employer, en mode conversationnel un ordinateur IMB 360, donc une machine virtuelle générée par CP/67. L'avantage de cette solution est de mettre à disposition d'un utilisateur les vastes possibilités d'un gros calculateur: grande mémoire, unités d'entrée sortie rapide, puissance de calcul, etc.

Immédiatement après initialisation, la machine CMS atteint un état stable particulier, appelé 'attente de commandes'. Les commandes disponibles appartiennent à deux familles:

1. Celles dont l'exécution provoque un changement d'état.
2. Celles qui après exécution impliquent un retour immédiat à l'état stable de base.

Différents états stables permettent d'entrer des fichiers, de corriger des fichiers, d'exécuter des programmes EDIT - INPUT - ATTENTE DE COMMANDES.

Chaque commande rencontrée est considérée comme phase élémentaire, d'un langage particulier et est exécutée immédiatement. L'idée de pouvoir conserver (donc de réutiliser) un groupe de commandes en lui donnant un nom collectif vient naturellement à l'esprit. On a donc un fichier de commandes et il suffit alors de frapper EXEC ‹ nom de fichier › pour que le systeme aille automatiquement lire ce fichier, et en interpréter

le contenu. C'est cette possibilité que nous avons utilisée pour la simulation, afin de conserver le même dialogue que sur les calculateurs de processus. Le langage utilisé est un macro langage de commandes CMS. EXEC est donc une commande de CMS permettant l'exécution de la séquence de commandes contenue dans le fichier portant le nom de cette macro-commande.

111 – SIMUL

Le système comprend Trois parties essentielles:

1. Identification de l'utilisateur.
2. Analyses des commandes.
3. Exécution des commandes.

L'identification utilisateur comprend l'appel du système SIMUL sous CMS, puis la reconnaissance de l'utilisateur (par son nom d'expérience). Ensuite ou exécute la concaténation des bibliothèques nécessaires, bibliothèques système et utilisateurs.

L'analyse des commandes, dirige vers les options Background-Foreground, (comme sur les calculateurs de processus) puis analyses les requêtes de l'utilisateur, et prépare le travail (créations de fichiers intermédiaires, appel de programmes, etc...), avant de donner la place à l'exécution.

L'exécution correspond au travail proprement dit, et peut aussi se décomposer en trois parties:

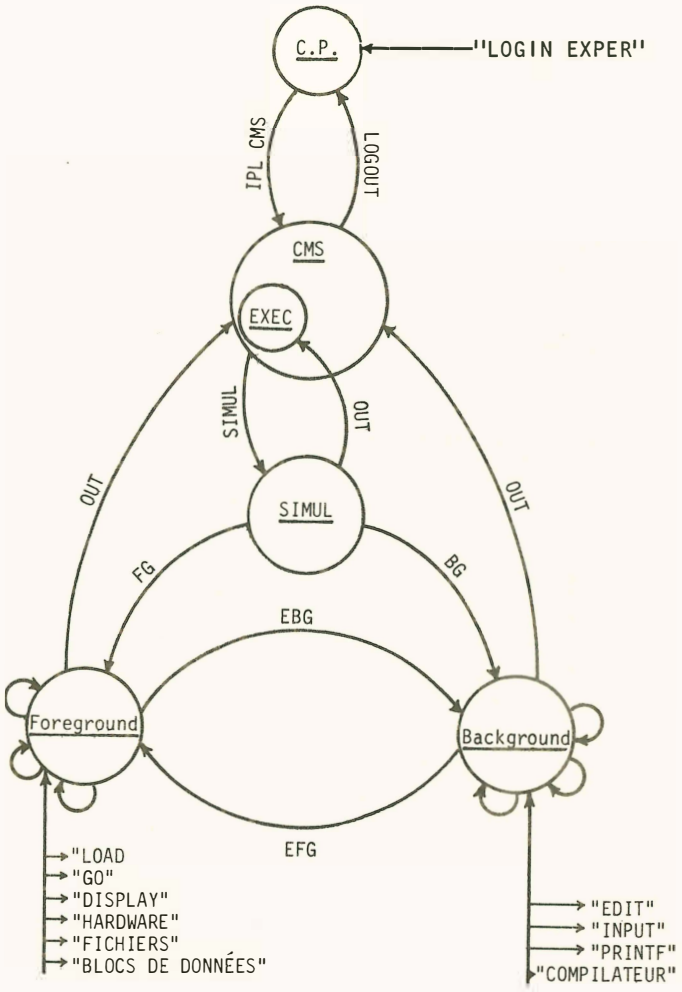
1. Exécutions d'ordres systèmes (CMS).
2. Exécutions de programmes FORTRAN système.
3. Exécutions de simulation expérience.

La partie BACKGROUND permet la création et la correction de programmes FORTRAN utilisateur, mais ne permet pas l'exécution, les modules temps réels étant seulement disponibles en exécution dans la partie FOREGROUND.

Dans l'environnement Background, les opérations suivantes sont disponibles:

1. Création d'un programme ou sous-programme FORTRAN..... (EPC)
2. Correction d'un programme ou sous-programme FORTRAN..... (CPG)
3. Liste d'un programme FORTRAN... (SPC)
4. Liste d'un sous-programme FORTRAN..... (SSC)
5. Compilation et génération d'un module..... (COP)
6. Compilation sous-programme ..... (COS)
7. Lecture programme sur ruban ..... (EPR)
8. Lecture sous programme sur ruban. (SSR)
9. Transfert dans l'environnement Foreground..... (EFG)

Dans l'environnement Foreground, l'utilisateur à accès à certaines commandes. Certaines autres commandes dépendent de la commande précédente exemple: on ne peut arrêter un programme FORTRAN s'il n'a pas été démarré.



Diapositive 3 Organigrame des environnements du système SIMUL

A tout moment, l'utilisateur peut, par la frappe de la commande 'HELP', demander les explications nécessaires à chaque commande.

De même à tout moment l'utilisateur peut sortir de SIMUL par la commande OUT. Il est nécessaire d'utiliser OUT, si l'on veut obtenir une bonne gestion des fichiers.

Les bibliothèques:

Le système SIMUL gère un certain nombre de bibliothèque:

1. Bibliothèques utilisateurs.
2. Bibliothèques systèmes.

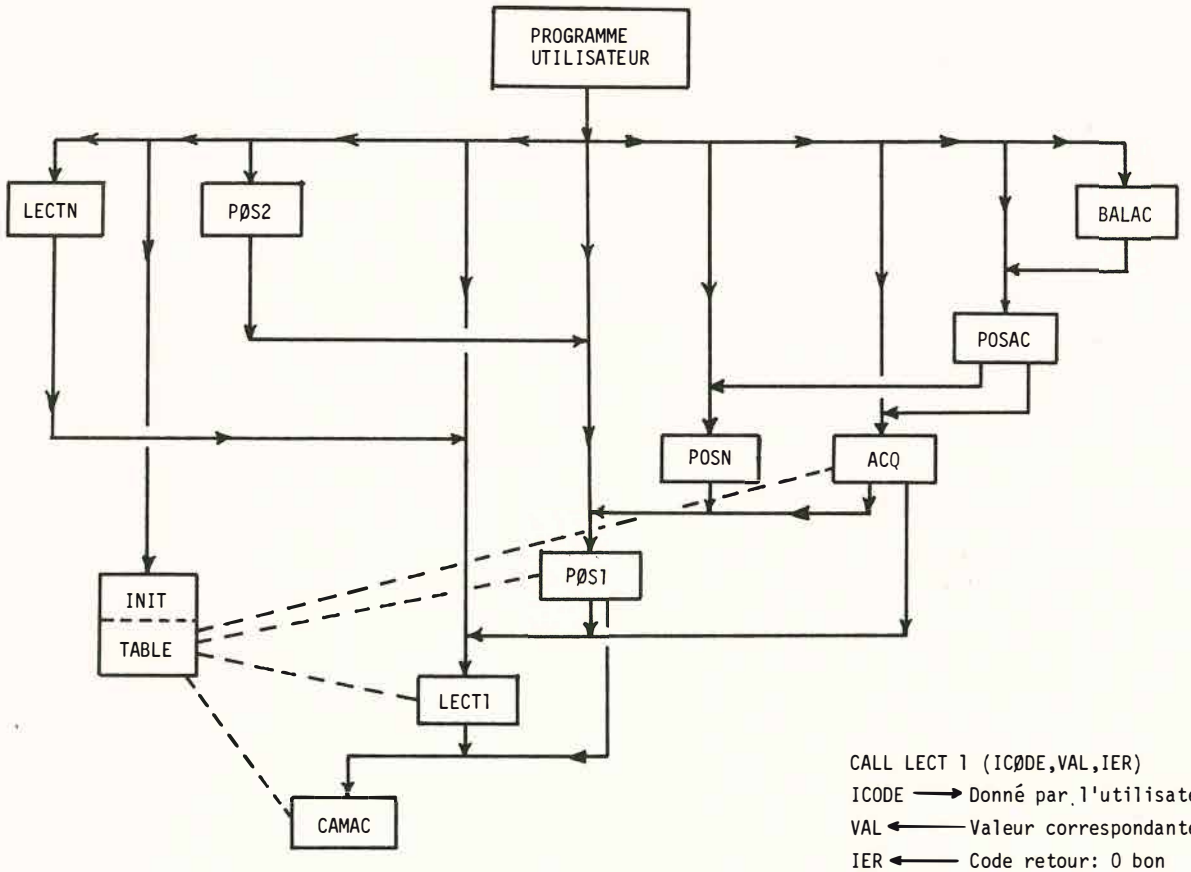
Dans les bibliothèques utilisateurs on comprend:

1. La bibliothèque de sous-programmes.
2. La bibliothèque de programmes.

directement liées aux problèmes utilisateurs. Ces bibliothèques sont extensibles, et compressibles.

Les bibliothèques systèmes comprennent:

1. La bibliothèque standard FORTRAN.
2. La bibliothèque Temps Réel.
3. Les programmes systèmes.



Diapositive 4 Modules

CALL LECT 1 (ICØDE,VAL,IER)
ICODE ⟶ Donné par l'utilisateur
VAL ⟵ Valeur correspondante
IER ⟵ Code retour: 0 bon
1 erreur LECT 1
-1 erreur CAMAC

Organigramme

La bibliotheque temps réel, est celle qui représente en fait, le coeur de la simulation. C'est à travers ces sous-programmes que l'utilisateur peut avoir accès à son électronique, donc à son expérience.

Le module de base est le module CAMAC, c'est lui qui simule l'interface de transfert, tous les autres modules feront appel à lui, lorsqu'ils voudront communiquer avec l'expérience.

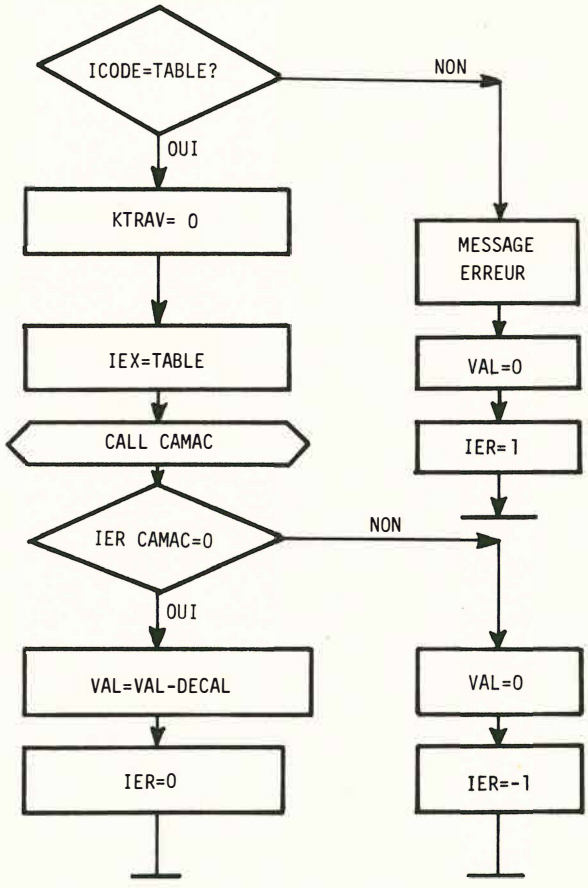
Le standard CAMAC consiste en une gestion du transfert d'information bidirectionnelle. Chaque module hardware CAMAC est repéré par une adresse (exemple COUNTER adresse BAC 1 - MODULE 15 SOUS-ADRESSE 2), pour éviter d'indiquer à chaque fois ce genre d'adresse, chaque variable physique CAMAC est répére dans le FORTRAN par un ICODE (entier), utilisé par le module CAMAC pour rechercher la réelle adresse.

Nous avons défini 3 sortes d'action au nivau du module CAMAC:

1. Lecture d'information.
2. Ecriture d'information.
3. Fonctions spéciales telles que RAZ échelles; initialisations, etc.

Exemple d'appel du module CAMAC:

```
CALL   CAMAC   (ICODE, KTRAV, IEX, VAL, IER).
```



Diapositive 5 Exemple de module LECT 1

ICODE est lié directement à la simulation, c'est lui qui fournira ou recevra l'information VAL. Dans le fonctionnement réel, ce sera le hardware lui-même qui fournira la valeur.

KTRAV indique la fonction: lecture, écriture spéciale.
IEX est l'adresse du module hardware concerné.
VAL information à transmettre.
IER code retour d'erreur 0 ----->bon
1 ----->mauvais

Les modules de plus haut niveau se composent de:

LECT 1 - LECTN

POS1 - POS2 - POSN qui font appel à LECT1 - LECTN

1V – Conclusion

Ce système de simulation fonctionne depuis le mois de Juin 1971 à l'institut Max von Laue-Paul Langevin de Grenoble, en donnant entière satisfaction.

Nous avons pu constater, que la facilité de mise en bibliothèque de sous-programmes particuliers, permettait une évolution constante du système en fonction des réalisation hardware.

De plus, ce système libère les calculateurs de processus de tâches importantes telles que édition, correction de programmes FORTRAN, en les transportant sur IBM 360/67 de fonctionnement plus souple, et aux plus larges possibilités.

An implementation of a virtual CAMAC processor

A. LANGSFORD
AERE Harwell, England

1. Introduction

The impetus to design and build a 'virtual CAMAC processor' came from experience in using CAMAC in a multi-user, multi-programming environment, gathering data for nuclear physics experiments [1]. Two factors were major influences in the design:

1. Programs tend to have a short life-time in this environment. As the needs of each experiment changed, it was found necessary to recode programs. This meant that not only high-level (FORTRAN) source code changed but also the individual CAMAC operations. Changing the high-level code is relatively easy, but manipulating bit patterns representing CAMAC commands was, by contrast, difficult and more prone to programmer error.
2. To speed data gathering, autonomous transfer of data between CAMAC modules and computer memory was implemented, the program controlling these transfers being hardware encoded within CAMAC.

Recognising that CAMAC had considerable processing power independent of the Central Processor Unit, it was a reasonable step to suppose that the CAMAC controller and the main processor could form a dual processor system, each accessing a common memory for their instructions (see Fig. 1).

Rather than build such a processor in hardware at the outset, it was decided to build a 'virtual processor' [2]. A simple hardware interface between computer and CAMAC dataway was supplemented by a software driver which interprets sequences of instructions placed in memory as CAMAC sub-routines. This approach has the advantage that, if the instruction set and addressing technique chosen for the virtual processor is found unsuitable, it can be readily altered.

2. Processor structure

The processor is designed to obtain its instructions from a read-only program segment. Data is held in a read-write permitted data segment. The

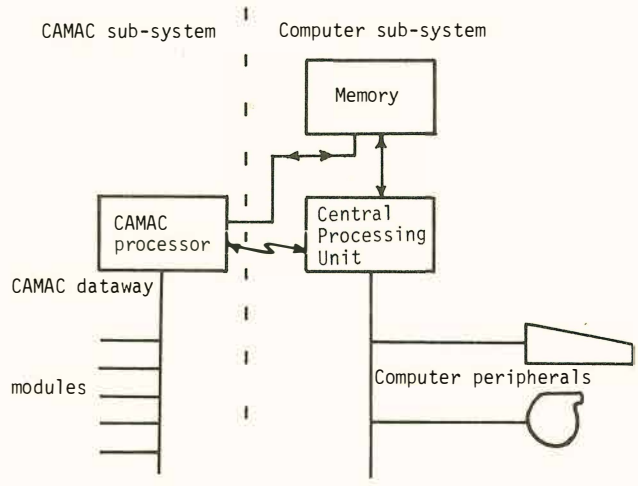


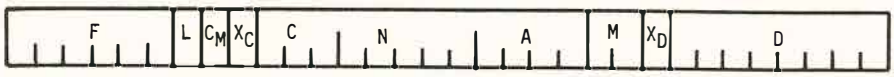
Fig. 1 Schematic diagram of CAMAC and CPU as a dual processor system

base addresses of these two segments are passed to the CAMAC processor whenever a sequence of CAMAC operations is called, the addresses being held in a Program Segment Register and Data Segment Register, respectively. In the present implementation each segment has a directly addressable length of 256 words.

The processor has, in addition, three active registers:

P	the program counter	(8-bit)
Q	the CAMAC condition register	(1-bit)
X	an index register	(8-bits)

A two-address structure has been chosen, the form of the instruction word being shown in Fig. 2. Because the CAMAC processor has been implemented on a 16-bit word length central processor, it was found convenient for the CAMAC instruction to occupy two central processor words, i.e. 32-bits. The simplest type of instruction provides a direct transfer of data between a CAMAC module (addressed by crate, C, Module, N, and Sub-address, A) and a location in the data segment, given by the displacement address D. All 32 CAMAC functions are implemented in this way, though half of them do not require a data address-



F	function code	N	module
L	length bit	A	sub-address
C_M	CAMAC address mode	M	displacement address mode
X_C	CAMAC index bit	X_D	displacement index bit
C	crate	D	displacement field

Fig. 2 The CAMAC processor instruction word

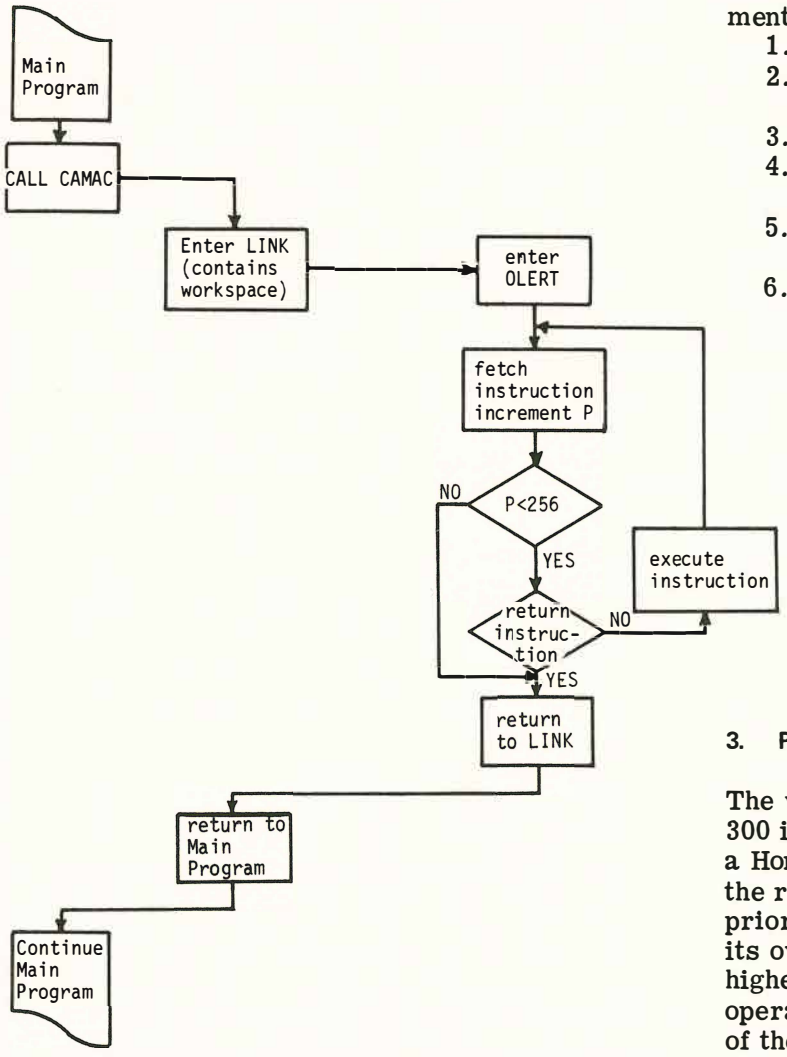


Fig. 3 Flow chart showing access to CAMAC interpreter

part being data-less transfers. However, CAMAC data-less functions 8, test-look-at-me, and 27, test-status, which affect the condition register, Q, have an address part. The contents of the specified address are set logical 'true' or 'false' depending upon whether Q is 'set' or 'cleared'. There are in addition 'non-CAMAC' operations which permit operations on the X and P registers. The latter provide branch unconditional and branch conditional on the value of Q. There is a return instruction which provides an exit from the CAMAC segment. Any instruction which cannot be decoded is treated as a return.

A number of address modes have been implemented. These provide for:

1. Direct CAMAC addressing.
2. Indirect CAMAC addressing, the direct address being found in the data segment.
3. Direct data segment addressing.
4. Indirect data segment addressing with pre-indexing.
5. Indirect data segment addressing with post-indexing.
6. Direct program segment addressing.

3. Processor implementation

The virtual processor has been written in about 300 instructions of re-entrant code to operate on a Honeywell DDP-516 computer working under the real-time executive OLERT. It runs at the priority level of the calling program which holds its own set of CAMAC registers. In this way higher priority processes may interrupt the operation of the CAMAC processor, the context of the interrupt process being automatically preserved. It is necessary to inhibit processor interrupts only for short periods. These are essentially during the sequence of operations to transfer data over the CAMAC data-way, when the operations –

set up command word: transmit data: test Q

– must be treated as an indivisible instruction.

The user program accesses the virtual processor through the FORTRAN call statement:

CALL CAMAC (data segment, program segment name)

or its assembly language equivalent.

The data segment is an integer array declared in a DIMENSION statement. The program segment can either be an internally named array or an external subroutine name. The sequence of operations carried out in accessing, running and return from the virtual processor is shown in Fig. 3. It will be seen that entry to the executive is through a link which holds workspace and the registers P, X and Q peculiar to the calling program.

4. Possible processor extensions

The author is aware that there are alternative approaches to the design of a CAMAC processor, real or virtual, and that many of the features of the present design result from the environment in which it has been implemented. With increasing experience and the freedom to manipulate the design of the processor, it is hoped that improvements will follow and omissions be remedied. Already certain desirable operations can be seen. It would be useful to have a conditional branch instruction, where the condition was set by the pattern of data transferred between store and CAMAC data-way. A far more significant omission, and one which the author intends to remedy soon, is the lack of real-time features. Because CAMAC provides for autonomous data transfer and interrupts, the virtual processor needs additional instructions so that parallel processing can be accomplished. To do this satisfactorily requires information from the user program to identify a point in his program whose execution can be scheduled when both the current CPU process and the parallel CAMAC process have terminated. While the author knows, in principle, how to achieve this, for demonstration purposes the present implementation has to be made compatible with the structure of OLERT, the environment in which the code will be executed.

It is the author's belief that, by making it more easy to program CAMAC operations, especially in a real-time environment, a better understanding of the structure of CAMAC programs will emerge. This will, in turn, be reflected in the improved definition of CAMAC statements in high-level languages. Moreover, defining a range of processors, real or virtual, on which CAMAC statements are to be executed provides ready implementations for the high-level language statements.

1. LANGSFORD, A., JARVIS, O.N., and WHITEHEAD, C., 'DAMUSC – a direct access, multi-user, synchrocyclotron computer', AERE R. 6832.
2. LANGSFORD, A., 'An implementation of a virtual CAMAC processor and its assembly language', AERE R. 6914.

Discussion

Q. What is the speed of the hardware in your system?

A. The computer itself is a microsecond machine, but the interpretation is appreciably slower than this: not fast enough for high speed CAMAC operations.

Q. What is the size of your interpreter?

A. About 300 instructions.

An implementation of the assembly language and program assembler for a virtual CAMAC processor

V.J. HOWARD
A E R E Harwell, England

1. Introduction

Langsford [1, 2] has described a possible implementation of a virtual CAMAC processor. This processor possesses its own repertoire of instructions and a set of virtual registers which enable it to execute sequences of instructions placed in the core store of the host computer. The instructions are stored in a read only block termed the I/O or program segment and the data for the program is transferred to and from a read/write area of store called the data segment. The virtual CAMAC processor shares store access with the CPU (central processing unit) of the host computer and will execute CAMAC instructions concurrently with CPU activity.

To assist the user to program the virtual CAMAC processor and to plant I/O segments within the computer store a symbolic assembly language has been developed and a two-pass program assembler has been implemented. In form, the assembly language reflects the facilities offered by the CAMAC processor. It supports symbolic addressing of individual CAMAC modules, data segment locations and locations within the I/O segment and it also provides facilities for extending the set of instructions recognised by the program assembler should this be necessary as the virtual processor evolves.

For ease of implementation the program assembler was written in PL/1 for use on an IBM 370/165 computer. It will assemble programs coded in CAMAC assembly language and produces a block of object code in a form suitable for loading into the store of the host computer using the standard loader program for CPU instructions and data. This technique has certain advantages in that any I/O segment for the CAMAC processor is relocatable within the store of the host computer and can be shared by a number of programs. It also allows the I/O segment to be called at run time and permits it to be treated as a separate program module which can be link-loaded with modules coded either in the assembly language of the host computer or in any high-level language implemented on that machine. The program assembler is modular in structure and is so organised that the CAMAC processor instruction word format and the output format of the virtual object code can be readily altered to suit the computer system in which the virtual CAMAC processor resides.

2. The CAMAC assembly language

The assembly language for the virtual CAMAC processor is of conventional form for a two-address machine. The language provides a symbol for every legal function that the virtual processor can carry out and will detect and flag invalid instructions at load-time. It also supports symbolic descriptions of addressing modes and of locations within the I/O and data segments and it defines the format of statements for input to the CAMAC program assembler.

2.1 Statement layout

An assembly language statement is divided into five fields, laid out as shown below:

```
Location:  function, CAMAC address,
                        store address, comments;
```

For all instructions the 'location' and 'comments' fields are optional. For certain instructions only one of the two address fields need be specified.

2.2 The function field

The function field specifies either the function or operation to be carried out by the virtual CAMAC processor or one of a set of pseudo-operations

TABLE 1 Non-CAMAC operations

Operations on the X register	
LOAD X	Load X-register with contents of location in store address field.
STORE X	Store X-register in location specified in the store address field.
SWAP X	Interchange contents of the X-register and the location specified in the store address field.
INCR X	Add 1 to the X-register. Compare X-register with contents of the location in the store address field. If they are equal skip the next instruction.
Operations on the P register	
JUMP	Jump to I/O segment location specified in the store address field.
QSET	Jump to the I/O segment location specified in the store address field if the Q-register is set to 1.
QNOT	Jump to the I/O segment location specified in the store address field if the Q-register is set to 0.
Addressless instructions	
HALT	Terminate execution of the I/O segment.

used to control the program assembler and to associate symbols with data and I/O segment locations. The operations available to the processor and their symbolic names are summarised in Tables 1 and 2. The functions in Table 1 are non-CAMAC functions. They require no CAMAC address field and operate on the P, X and Q registers of the virtual CAMAC processor. The HALT instruction uses no address fields at all. When encountered at run time it terminates execution of the CAMAC program.

Of the 32 valid CAMAC functions listed in Table 2 only the 16 standard functions have been assigned symbolic names. However, to facilitate the use of all 32 CAMAC function codes and to provide for the addition of new instructions individual operation codes can be specified numerically in the function field as (f1, f2). Field f1 is a 6-bit function code which defines either a CAMAC or non-CAMAC operation depending upon whether the most significant bit is 0 or 1. The remaining five bits specify either the CAMAC function code or the non-CAMAC operation. Field f2 is a single bit field which defines the length of the data transfer in a CAMAC read or write operation. The prefix 'D' on READ-group or WRITE-group instruction mnemonics fulfils a similar function. The idea of multiple length transfers is introduced to overcome the problem of carrying out transfers when the computer word length is less than 24 bits, the length of the CAMAC data highway. For a machine length, i.e. single length transfer (f2 = 0), the virtual CAMAC processor matches the low order bits of the CAMAC highway to the natural word length of the computer. For CAMAC length, i.e. multiple length transfers (f2 = 1), the low order of the data highway are treated as for single length transfers and the high order bits are stacked in successive computer words. The DEFINE DOUBLE pseudo-operation provides facilities for reserving storage for multiple length transfers.

2.3 The store-address field

2.3.1 Addressing within the I/O segment

Locations within the I/O segment are referenced by jump instructions and when the address field specifies a constant which is defined explicitly or symbolically within the I/O segment. Literals are defined explicitly by an 'equals' symbol followed by an optionally signed constant,

```
eg  LOADX ,  = 500 ;
```

They can also be defined symbolically by the DEFINE LITERAL pseudo-operation,

```
eg  LITNAME: DEFINE LITERAL , 500 ;
```

Any reference to LITNAM would produce a reference to a location in the I/O segment initialised

TABLE 2 CAMAC operations and their symbolic names

Function number	Name	Type	Comments
	*		
0	READ		Read Group 1 Register
1	READ 2		Read Group 2 Register
2	READCL		Read and Clear Group 1 Register
3	READCMP	Read group	Read Complement of Group 1 Register
4			Non-Standard
5			Reserved
6			Non-Standard
7			Reserved
8	TESTLAM		Test look at me ≠
9	CLEAR		Clear Group 1 Register
10	CLEARLAM	Dataless transfers	Clear look at me
11	CLEAR 2		Clear Group 2 Register
12			Non-Standard
13			Reserved
14			Non-Standard
15			Reserved
	*		
16	WRITE		Overwrite Group 1 register
17	WRITE 2		Overwrite Group 2 register
18	SWRITE	Write group	Selectively Overwrite Group 1 register
19	SWRITE 2		Selectively Overwrite Group 2 register
20			Non-Standard
21			Reserved
22			Non-Standard
23			Reserved
24	DISABLE		Disable
25	INCREMENT		Increment
26	ENABLE		Enable
27	TESTSTATUS	Dataless transfers	Test Status ≠
28			Non-Standard
29			Reserved
30			Non-Standard
31			Reserved

≠ Dataless transfers, except TESTCAM and TESTSTATUS, use only the CAMAC address field.
For TESTLAM and TESTSTATUS instructions the value of the Q-register is loaded into the location specified in the store address field.

* READ and WRITE group instructions require both CAMAC and store address fields. Such instructions, prefixed by D, imply double length transfers.

to 500.

Addresses referenced within the I/O segment can be modified by placing (1) at the end of the store address field,

```
eg  JUMP , IOSYMBOL (1) ;
```

Thus, in the above example, control would jump to the location within the I/O segment whose address is that of IOSYMBOL plus the contents of the X-register.

Indirect addressing is not permitted in the I/O segment within the present implementation.

2.3.2 Addressing within the data segment

This represents the most common form of addressing and a number of options are provided. The basic form is a direct reference to some location within the data segment. This mode can be specified either by inserting the address of the

location in numeric form within the store address field or by using a symbolic address defined by a DEFINE DATA or DEFINE DOUBLE pseudo-operation at some other point within the program,

```
eg   STOREX, DATA SYMBOL ;
     - - - - - - - - - - -

DATA SYMBOL:  DEFINE DATA, ( );
```

As with I/O segment addresses, direct references to data segment addresses can be modified by placing (1) at the end of the address field,

```
eg   LOADX , PQR(1) ;
```

simple expressions of the form

Symbolic name ± integer

are also permitted.

To enable the user to reference any word within the computer store by its absolute address the virtual CAMAC processor provides for indirect addressing through the data segment. Indirect addressing is declared by enclosing the symbolic address of the data segment location which contains the absolute address in parentheses,

```
eg   STOREX, (RST) ;
```

In the above example if RST contained 350 then the contents of the CAMAC processor X-register would be deposited in absolute store location 350.

Indirect addresses can be modified either before indirection (pre-indexing) or after indirection (post-indexing). Pre-indexing is defined symbolically by a store address field of the form:

```
(SYMBOL(1))
```

In the above example, if the X-register contained 5 and SYMBOL was location 20 in the data segment then the processor would access location 25 of the data segment for the indirect address.

Post-indexing is defined by a store address field of the form:

```
(SYMBOL) (1)
```

In this instance the contents of the X-register would be added to the contents of location SYMBOL to give the indirect address.

Only one level of indirect addressing is allowed and in the present implementation address modification before and after indirection is illegal.

2.4 The CAMAC address field

A CAMAC address can be defined explicitly by placing the CNA address of the module as a three integers, enclosed in parentheses, in the CAMAC address field or by referencing a symbolic address defined at some other stage of the program by a DEFINE CAMAC pseudo-operation,

```
eg   CLEAR, (c, n, a) ;
or   CLEAR, COUNTER ;
     - - - - - - - -

  COUNTER: DEFINE CAMAC  (c, n, a);
```

The examples above carry out the same function. The quantities c, n and a are integers giving the crate, station and module sub-address respectively.

Direct CAMAC addresses can be modified by placing (1) after the CAMAC address. Here modification is performed on the N address alone and is taken modulo 32.

To permit CAMAC addresses to be defined at run time it is also possible to specify that a CAMAC address may be found within the data segment. Here, the CAMAC address field will contain either a symbolic reference to the data segment location holding the CAMAC address or an indirect reference to some other location in the core store in which the CAMAC address can be found. The symbolic representation of direct and indirect data segment addressing is the same for both CAMAC and store address fields.

2.5 The location field

The location field fulfils three functions. First, it permits symbolic names or labels to be assigned to locations within the I/O segment. Secondly, it is used with pseudo-operations to define data segment and CAMAC addresses symbolically and, thirdly, it is used to define the name of the I/O segment.

2.6 Pseudo-operations

In addition to the executable instructions listed in Tables 1 and 2 the assembly language also includes a number of pseudo-operations. These are used to permit the programmer to assign symbolic names to data and program locations and to control the program assembler. The full range of pseudo-operations is listed in Table 3. Reference 2 describes their detailed format and use.

TABLE 3 CAMAC assembler pseudo-operations (Ref.1)

DEFINE operations

```
funcname      :  DEFINE FUNCTION, (f1, f2);
CAMAC name    :  DEFINE CAMAC   , (C, N, A);
data name     :  DEFINE DATA    , (n1, n2);
data name     :  DEFINE DOUBLE  , (n1, n2);
literal name  :  DEFINE LITERAL , L ;
```

Assembly Controlling operations

```
I/O segment name  :  BEGIN CAMAC ;
                     END ;
```

3. A typical I/O program

A typical program, written in CAMAC assembly language, is shown in Fig. 1. It is taken from reference 2. The I/O segment PROGO1 is entered on CAMAC interrupt from one of the two units, FLAG1, which controls inputs and FLAG2 which controls outputs to an X-Y plotter. These modules are tested for L signals and on finding the unit generating the interrupt a jump is made to the appropriate section of code. On input, data from five consecutive modules is read into successive store locations. On output, a control word and data word are written to sub-addresses 1 and 0 of the plotter control module. The example

```
PROGO1  :  BEGIN CAMAC;
     TESTLAM, FLAG 1, DUMMY,              SEE IF FLAG 1 IS INTERRUPTING
     QSET, INPUT,                         JUMP TO INPUT IF YES;
     TESTLAM, FLAG 2, DUMMY,              SEE IF FLAG 2 IS INTERRUPTING;
     QSET, OUTPUT,                        JUMP TO OUTPUT IF YES;
     HALT,                                STOP NEITHER IS INTERRUPTING;

INPUT:  LOADX, = 0,                       CLEAR THE INDEX REGISTER;
     READCL, EVENT(1), BUFFER(1)          READ DATA AND CLEAR EVENT REGISTER;
     INCRX, = 5,                          TALLY X AND CHECK WHETHER 5 EVENT READ;
     JUMP, INPUT,                         MORE TO READ IN;
     CLEARLAM, FLAG 1,                    REMOVE Q FROM FLAG 1;
     HALT,                                STOP AFTER INPUT PROGRAM

               NOW START OUTPUT PROGRAM;

OUTPUT:  WRITE, PLOTTER 1, CONTROL,       SEND CONTROL WORD TO PLOTTER;
     WRITE, PLOTTER, MOVEMENT,            SEND DATA TO MOVE PLOTTER;
     CLEARLAM, FLAG 2,                    REMOVE Q FROM FLAG 2;
     HALT,                                STOP AFTER OUTPUT PROGRAM;

               DATA AREA AND CAMAC MODULES DEFINED;

     BUFFER:    DEFINEDATA, (0, 5),       RESERVE FIRST 5 WORDS OF DATA SEGMENT;
     CONTROL:   DEFINEDATA, (5, 1),       PLOTTER CONTROL WORD;
     MOVEMENT:  DEFINEDATA, (6, 1),       PLOTTER DATA WORD;
     EVENT:     DEFINE CAMAC, (3, 10, 0), EVENT MODULES 10 TO 14 INCLUSIVE;
     PLOTTER:   DEFINE CAMAC, (3, 1, 0),  MODULE 1 SUB-ADDRESS 0;
     PLOTTER1:  DEFINE CAMAC, (3, 1, 1),  SUB-ADDRESS 1 FOR CONTROL OF PLOTTER;
     FLAG 1:    DEFINE CAMAC, (3, 8, 0),  INPUT CONTROL UNIT;
     FLAG 2:    DEFINE CAMAC, (3, 9, 0),  OUTPUT CONTROL UNIT;
          END,                            END OF I/O SEGMENT;
```

Fig. 1 A typical CAMAC assembly language program

shows one technique of searching for an interrupt and a technique for loop control using the X-register of the virtual CAMAC processor.

4. The program assembler

The program assembler for the virtual CAMAC processor is implemented in PL/1 for use on the IBM 370/165 computer at Harwell. It will assemble programs coded in the assembly language described above and produces a block of virtual object code which can be transformed by a 'plug-in' module to a form suitable for loading into the store of the computer used by the virtual processor. At present the 'plug-in' module is configured for a Honeywell DDP-516 and will produce an output tape which is compatible with the standard linking loaders of that machine. However, by replacing this module with another of similar function, the CAMAC object code could be readily loaded into any other machine.

Programs for assembly are punched on cards in what is essentially free format. Individual language statements are delimited by semi-colons and statements may extend over many cards. Symbols can be any number of alphameric characters in length with the proviso that only the first twelve are significant, one of which must be a letter. There is one exception to this rule. To ensure compatibility with FORTRAN the first character of the I/O segment name must be alphabetic and only the first 6 characters are considered significant. All spaces within the source text are ignored.

The assembly is carried out in two passes. In pass 1 a symbol table is built up and the various pseudo-operations are processed. Pass 2 performs the instruction assembly. Output from the assembler consists of a listing of the source text as it was read from cards and an edited listing showing the text as it was interpreted by the assembler. Errors detected within an individual statement are listed immediately below the statement in question and the assembler lists the symbols defined in the program in a series of maps at the end of the program text.

In designing the virtual CAMAC processor it was realised that there were alternative approaches to the problem and that the present implementation would inevitably be modified or augmented in the light of experience gained with its use. In particular, it was reasonable to suppose that the instruction set would be extended and that the processor might be implemented on another computer or with a different instruction format on the DDP-516. The program assembler was structured with these possibilities in mind. It is table-driven and additional instructions and pseudo-operations can be readily introduced simply by extending the table of instructions stored within the assembler and by adding appropriate procedures to process the pseudo-operations. The format of the instructions can likewise be very readily modified providing a very powerful tool for investigating different implementations of the virtual processor.

The assembler will be made available for a PDP-11 installation.

1. LANGSFORD, A., 'An implementation of a virtual CAMAC processor and its assembly language', AERE R. 6914.
2. LANGSFORD, A., 'An implementation of a virtual CAMAC processor', presented at the II European Real-time Seminar.

Discussion

Q. How are the links established between the user's program and the virtual processor?

A. Simply by the subroutine call:

```
CALL   CAMAC   (DATAS, PROGO1)
```

where DATAS is the data segment and PROGO1 is the I/O segment, declared:

```
DIMENSION   DATAS   (256)
EXTERNAL    PROGO1
```

Q. It looks difficult: the user must combine everything in CAMAC by himself. Wouldn't it be better to extend the language?

A. It is not really very difficult. The user can assign symbolic names to the modules. Perhaps there will be extensions, but you have to implement it starting with the facilities of FORTRAN, e.g. common data areas.

Q. How do you get the CAMAC addresses into FORTRAN?

A. Here is an example to demonstrate how you use it. In the CAMAC part we would have

```
SYMBOL : DEFINE DATA (6, 1);
```

and in the FORTRAN part

```
DIMENSION DATAS (256)
EXTERNAL PROG
EQUIVALENCE (SY, DATAS (7))

CALL CAMAC (DATAS, PROG)
```

so the SY in the FORTRAN part is the same as SYMBOL in the CAMAC part. (DATAS in the FORTRAN part is enumerated starting from 1; the data segment in the CAMAC part is enumerated starting from 0.)

C. This is a common problem – it arises whenever an assembly language program uses variables in the same address space as a compiler language program.

Q. How does this fit with the work of the ESONE software group?

A. It is near but not coincident with that.

Q. Can you say something about the external processor and virtual object code for the PDP-11?

A. Not at this time. We are still working on it.

C. In CERN we have a different approach to the problem, and generate code in-line. This eliminates the need for the user to remember all the CALLs, etc.

Q. How do you debug this?

A. By trial and error. We do not have any special debugging tools, and recognize the deficiency.

System software real-time testing aids

W.E. QUILLIN
Plessey Radar, The Plessey Co, England

1. General methods of software system build-up

1.1 Design

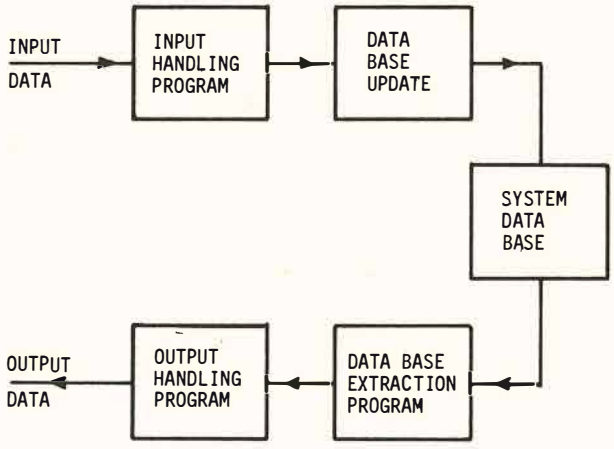
Before the use of testing aids can be considered it is helpful to consider the way a system is built. The system design is a top-down process, which takes the system from an overall specification to the design of individual program modules, which will be coded, tested and integrated to build the complete system. The design process also leads to the choice of suitable hardware.

1.2 Off-line testing

System implementation is a bottom-up process, starting with the testing and integration of the lowest levels of the modules. Initial testing will be in an off-line environment, in either a free-standing computer of the type for which the object code is intended, or in a computer of a different type, generally larger, which has a suitable emulator program. It is important that this off-line testing stage should be as thorough as possible, and remove all the coding bugs and design errors it can possible catch. A single free-standing computer is usually much easier to obtain and control for testing purposes than a complete on-line system. Some routines can be very extensively tested in this manner, others are much more difficult to off-line test, especially if they control complex interface equipment with real-time constraints. Not only single modules, but suitable collections of related modules, will be tested off-line. Input and output modules of these collections will be replaced by dummies, and assistance in the testing is provided by a set of off-line testing aids. The majority of faults which escape detection at this off-line testing stage will be interface errors and errors due to the constraints of real-time working.

1.3 On-line testing

For each package of modules there comes a time when either it is not possible to carry out any further off-line tests, or to do so would involve the production of an unacceptable amount of test program or test data. At this point on-line testing is started, and it is at this time that system testing aids are required. Like the off-line tests, the on-line testing starts with as simple a system as possible. The code is tested in its final object computer, but not all the other system computers and the full input-output facilities will be provided initially, see Fig. 1. Part of the job of the on-line test aids is to cover up the lack of these facilities and make it appear to the code under test that it is in its full system environment. The testing aids are primarily intended to help test the software, and leave it to a pre-test checkout or hardware detection routines included in the package under test to find any system hardware failures. The problem with a real-time system is that it is not designed to be capable of stopping and then being restarted. Hence the use of freeze-points in the code to trap errors is generally inappropriate. The testing must take place with the system in operation and it is highly desirable that minor changes can be made, whilst the system continues to run.



TEST 1 INPUT HANDLING PROGRAM
TEST 2 OUTPUT HANDLING PROGRAM
TEST 3 INPUT HANDLING PLUS DATA BASE UPDATE
TEST 4 OUTPUT HANDLING PLUS DATA BASE EXTRACT
TEST 5 COMPLETE SYSTEM

Fig. 1

2. Requirements for system testing aids

Testing aids are required to fulfil the following functions in a software system under development.

1. To inject data into the system under test. The simplest form of data injection required is the addition of items to stacks, queues and lists in the system, to check that the processing of this data is initiated and performed correctly. Such requirements generally arise because the system under test is incomplete. The need also arises sometimes during testing as it becomes necessary to exercise a suspect program on a more rapid basis than would be the case in normal use. Another type of data injection is the on-line amendment of contents of common core store and computer core store. This gives test personnel a method of controlling the running system, temporarily masking faults to enable other faults to be found, and, if they are not careful, completeley wrecking the software under test. It is a facility which must be used with caution, particularly when changing computer core store contents if the computers have a relocatable loader. The system testing software should enable tasks of the system on-line operating system to be initiated or suspended during the test by suitable data injection.

2. To extract data from the system for subsequent analysis. Once again the simplest case is extracting data from a stack, queue or list. There are two cases here, one where the queue or list forms the end of the process under test, i.e. the program which would usually empty it is not yet included. In this case the testing aid must remove entries at a suitable rate to prevent unwanted list or queue full indications. The second case is more complicated, where the list or queue is an intermediate point in the software under test. In this case the test program must look at and record entries but not remove them. The sampling rate should be rapid, so entries are not missed, but in practice if there is no designed phase relationship between the test programs, the filling program, and emptying program this is generally impossible as the occasion will arise where a list is read immediately after an entry is added and the monitoring software has no time to catch the entry. The debugging aids must also be able to observe more general system data, such as models showing current system configuration, current state of world models in which the system has an interest, operator and data link injected data, fault reports, etc.

3. To vary the real-time working of the system. Requirements arise during testing when it is required that the system should run faster or slower than real-time. Difficult faults may be found easier in a slow running system – the ultimate in slow running a system is to be able to 'One-Shot' it, and the system testing aids should make this possible, in conjunction with the on-line operating system. Other system faults may require the system to run faster than real-time, this being especially true of faults which require assistance of engineers using instruments, e.g. oscilloscopes. Testing, both hardware and software, is designed to ensure that such faults are rare, but they cannot be completely eliminated. A facility should also be provided to enable part of the program under test to be continuously repeated, as an aid to finding intermittent and other seldom occurring faults.

4. The system testing forms part of the total documentation. This is required to give a record of the testing methods used and to assist in the tracing of faults which occur during use and during subsequent software or hardware modifications. The system testing software can assist this documentation task by giving a print-out in a suitable form for incorporation in the documentation.

5. Frequently during testing the designed method of system start-up and shut-down is inappropriate due to the need to run with parts of the system incomplete and the desire to reduce testing time when possible. The system test software should enable various start-up and shut-down procedures to be followed as appropriate to the test being conducted.

3. Examples of testing software successfully used on on-line systems

The following examples consider two on-line test programs written and used by the Plessey Co. for testing on-line software. They are currently being modified and integrated, but for the purposes of this paper they are considered as individual programs.

3.1 Electronic data display monitor

This program was written to run on a system which has a number of operator positions which include alpha-numeric keyboard inputs and electronic data displays (or VDUs), onto which the system can write alpha-numeric characters, as output. The same design could be applied to other I/O media, e.g. keyboard and teleprinter or lineprinter. This would help the output to be of assistance for documentation reasons, but it can lead to the problem of the system testers disappearing under a mountain of paper whilst they search for a sheet which is relevant to a problem under test. The program was in fact so

arranged that inputs can be accepted from paper tape and the computer control console as the positions' keyboards. In practice these first two options have rarely been used.

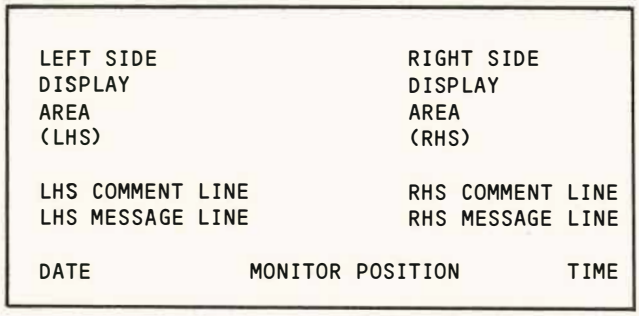


Fig. 2 Basic display for EDD monitor

3.2 The basic display is shown in Fig. 2. It requires a position with at least a 10 line display facility. The program will operate if using a position with only 8 lines (in one case some positions had only 8) but the Date, Monitor Position and Time Displays are then not available. The position in use is set up at the start of a trial and can be changed if required during the testing period. Each side of the display is treated independently and on each side there can be displayed up to six words of eight or twelve digits. Data is loaded into the display formats from the top and if a smaller number of words than six is required the lower data lines will remain blank.

3.3 There are four types of data messages:

i Display up to six words from a common core store.
ii Display up to six words from a computer store.
iii Inject up to 6 words into a common core store.
iv Inject up to 6 words into a computer store.

Data is set up on the monitor in octal. In the case of injections to core store, the operator must confirm the data and then inject, giving ample chance to remove an error. Repeated injections of the data set up are possible.

3.4 There are also four possible auxiliary messages:

i Add 'nn' to a given address (nn is a two-digit octal number).
ii Subtract 'nn' from a given address.
iii Continually update displayed data.
iv Freeze display.

The first two apply to any of the Data messages, the second two only to Display Data messages.

3.5 There are also four clear and erase messages:

i Clear a whole display.
ii Clear right side of display.
iii Clear left side of display.
iv Erase last character injected.

3.6 There are comment messages which are given to the operator upon a fault condition.

i Wrong Key – Try Again.

This is displayed upon operation of an illegal alpha-numeric key. It is cleared on a valid injection.

ii No Data message displayed.

This is displayed if an operator attempts an Auxiliary Data message when no Data message has been injected.

iii Common core address invalid.

If the full address field is not used for common core store addresses and an out of limit request is made, this message will be displayed.

iv Faulty Transfer.

After an inject to store message the program sends the data, then reads it back as a check. This message is displayed if the check fails.

v Inject Inhibited.

It is sometimes necessary to inhibit the injection of data, for example during system trials. This inhibit is controlled from a computer console key. The above message is displayed if the operator tries to inject whilst this key is operated.

vi Computer Not Available.

Whilst transferring injections to or from another computer the program expects a confirm reply. If this does not arrive in a predetermined time, the above message is displayed.

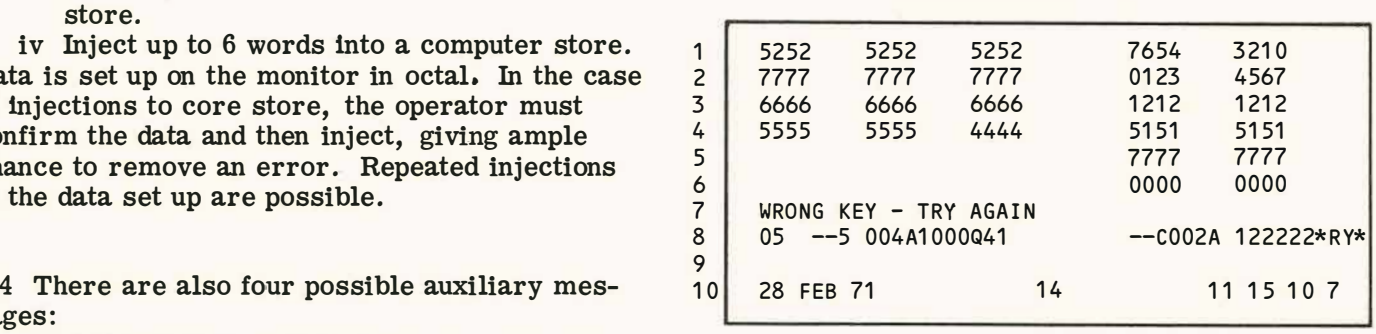


Fig. 3 Sample format of EDD monitor

3.7 Figure 3 shows a sample format. The left side display "--S 004 A 1000 Q4 I" means inject the four words of data into store 004, starting at

address 1000. The 05 is the number of times this data has been injected. The operator has operated an invalid key. The date is 28th February, 1971. The right side display message "--C 002 A 122222*" means display 6 words of data from computer 002 starting at address 122222. The Auxiliary Data Message RY* means continually update the display. The time is shown as 11.15 10.7, i.e. 10.7 secs after quarter past eleven.

3.8 Experience with this program has shown that it makes it possible for an experienced system test team to locate errors by working in an interactive fashion with the running system. The use of a display enables many data items in computers and common core stores to be scanned rapidly, to find for example where a chain of events breaks. Data changes of $\frac{1}{4}$ sec can be detected on the display enabling entries in queues and lists to be seen between addition and removal.

4. SCOT program

The second system test aid program to be described is called SCOT (System Computer Online Test) and covers testing functions described in paragraph 2 which are not covered by the EDD Monitor program, such as adding data to queues, giving output suitable for documentation and system start-up. Whereas the EDD monitor program is designed for searching for faults and obtaining small amounts of data (by writing down display contents or photography) the SCOT program is used to obtain large volume outputs.

4.1 The following description of input messages shows the program's capabilities.

i Specify common stores in use for the test. The address numbers of the stores to be used are input.
ii Place data in common stores. From one word up to a complete store may be set up.
iii Extract all entries from a queue and reset control word. The entries extracted are printed out.
iv As above, for a list.
v Compare Data in a specified area of a common store with its previous contents. Any differences are printed out.
vi Add an entry to a queue.
vii Add an entry to a list.

Write directives (ii), (vi) and (vii) must be preceded by a time-tag which can range from $\frac{1}{4}$ sec upwards in steps of $\frac{1}{4}$ sec. Extract directives may be time-tagged. If they are not, they will be executed every $\frac{1}{4}$ sec. A choice of time sources is provided – the program will use a system clock if available, if not a programmed timer can be used – which is less accurate.

There are a number of program usable control keys on the computer consoles. Keys one to twenty can be used to switch individual directives on or off if they specify the key number.

As well as pre-loading the common core stores the program can be used to start system computers in any order by sending specially coded messages to them, whilst they are in a start-up state. Queues, Lists and Data areas can be referenced by a CORAL name instead of the store number and address. This is arranged by a declaration of the form

⟨name⟩ * store number store address

where * is either Q, L or D for queue, list or data area.

```
S   2,  3,  15
K1   T   1   47.5
WDS   3   A   1250
C 1234   5670   1234
C 5252    5252    5252
T3   30.0
EQS  4   A 1000
```

Fig. 4 SCOT input message example

4.2 Figure 4 shows an example of a SCOT input message. First the stores in use for the test are declared, in this case 2, 3, and 15.

The next message is to write to store 3, address 1250 the two words given in octal. This is after 1 min 47.5 secs, and as KI is specified it will only be obeyed if computer console key 1 is operated.

The next message, which is independent of the console keys as none is specified, should be actioned at $3\frac{1}{2}$ min after the program starts. It is to extract data from a queue in store 4, address 1000. In fact, it will result in an error when it is read into the computer, as only stores 2, 3 and 15 have been declared for the test.

4.3 Figure 5 shows the form of a SCOT output. First the start time is output if the program has access to the system clock. Then the system stores in use are shown, not exactly as depicted here as the present version outputs the store availability table built up as a result of the store declaration input. However, this example has been simplified to make it more clear, and avoid a need for system table layout knowledge.

The next output message echoes the input which was to write to store 3, address 1250 the two octal constants shown, as a check that this action was completed.

```
START TIME    12   45   52   25
SAL 02
SAL 03
SAL 15
Str 3    Adr 1250
C 1234   5670   1234
C 5252   5252   5252
E min 3 sec 30.0      Str 3 Adr 1000
C 0003   0003   0401    cnt 3 tai 1 hed 3
C 0357   4732   0101
C 0
C 3716   5125   3076
C 1327   0312   7766
```

Fig. 5 SCOT output message example

The next output message is the result of reading a queue in store 3, address 1000. Note that it is a queue of five words (four plus a control word) the maximum length being shown in the control word. The control word also shows where the head is (word 3), the tail (word 1) and the count of entries (3 in this case). These are output as decimal integers as shown.

4.4 Error reports from SCOT fall into three classes.

(a) Input faults.
- i Fault in common core store declaration or no stores declared.
- ii Fault in octal constant.
- iii Fault in number given for common store, out of range or not declared.
- iv Address fault.
- v Fault in decimal number.
- vi Fault in time given.
- vii Fault in name declaration.
- viii Name not declared.

(b) Store faults.
- i Initial check unsuccessful. The stores declared are checked at the start of the test, and should this check fail this message is output, along with the data sent, data received and the store number and address.
- ii Transfer of data unsuccessful. Data written to a store is read back and checked. If the check fails, this message is output along with the four items as in (i).
- iii Routining fault. Every $\frac{1}{4}$ sec a routining pattern in a reserved word in every store is checked. A fault causes this output along with store number and data returned.

(c) Access faults.
- i Queue full.
- ii List full.
- iii Queue locked out too long.
- iv List locked out too long.
- v Computer failed to start.

The time is given along with the above five fault reports.

4.5 Experience has shown SCOT to be a valuable system testing aid. It is particularly good at eliminating interface errors before system integration tests. To achieve this, the programmer who is writing say a program to extract data out of a queue and process it will get a programmer whose code would normally help fill the queue to provide a test tape, thus ensuring no misunderstanding.

The hardware tests of common core stores were included in SCOT as a 'long stop' to help show any faults which may arise during a test. These checks have shown faults during test runs, but these have generally been due to manual equipment allocation errors, not hardware, when a store has been out of system when required or removed in error during a test. A system is more prone to such errors during build-up time as the tests do not often require the total system and other activities are allowed to use the spare equipment.

5. Hardware assistance with system testing

The constant problem during testing is one of data collection – finding out what state the system was in just before it faulted. Up to now this assistance has usually been arranged by special software, and this paper has considered in detail two programs which have been used to achieve this. It is also possible to arrange for the system hardware to collect data to assist in on-line testing. This can be by hardware built-in to the the system or special-purpose test equipment which can be attached. These alternatives are now considered.

5.1 Facilities built-in to the computer hardware

In the case of real-time computer systems there are a number of ways the hardware design can help the system testers. Historical registers and test driver facilities are examples of these.

- i Historical registers.
  The computer contains a group of, say, sixteen registers addressed sequentially so they form a circular queue. When an instruction involving a single store operand is obeyed, three values are placed in these historical registers:

Instruction Address Register, Absolute Value.
Instruction.
Store Operand Address.

The third item is omitted if the instruction does not involve an operand address. Hence the last six or seven machine instructions obeyed can always be found in the historical registers. Their contents can be frozen by either a fault interrupt or by program control, and their contents then extracted.

ii Test driver facilities.

Special interface facilities can be provided so that the computer may be test-driven by another computer (of the same or a different type) which can access its registers and core store. This enables a trace of the computer's activities to be obtained during real-time working, when a software trace facility is inappropriate due to the slowing down of run time (typically by a factor of 80).

5.2 System attachments

System attachments can be obtained for data recording. These are generally similar to the historical registers described above, but they are portable and will usually interface with any convenient points of the system, e.g. store data lines, store address lines, interconnection buses, peripheral devices. Data is recorded in special registers or on magnetic tape for subsequent analysis off-line. They are particularly useful towards the end of system testing to measure use factors of connection channels and look for system bottlenecks. Some are equipped with an address comparator which can be used to show, for example, the amount of time spent in the operating system routines. Once experience is gained patching such monitors into a system they can quickly provide detailed information on suspected fault areas without requiring special on-line software.

6. Conclusion

Experience has shown that time and money spent on providing system testing aids has been essential to the testing of on-line system software, from the start of first on-line tests up to final system trials. Even after a system becomes operational some faults will continue to occur and the test aids have a continued value.

As the cost of hardware decreases and computer designers pay more attention to software needs, more assistance is likely to be given by special circuitry to the on-line testing of computer systems.

Discussion

Q. Do your debugging aids run as part of the user's program in real time?

A. Yes. It depends on having enough capacity in the computer, which we have.

Q. Is it possible to make the program run more slowly than normal?

A. Yes, the rate can be controlled.

Testing and diagnostic aids for real-time programming

E.C. SEDMAN
Computer Analysts and Programmers Ltd, Reading, England

1. Introduction

Most papers on computing topics tend to be concerned with the design of programs or systems, but experience indicates that this is only a small part of the total effort involved. Aron [1] states that 30% of the elapsed time of a large project is spent on design, with 40% on implementation and 30% on testing. Figure 1 is a simplified form of Aron's diagram, which shows how the number of people engaged in a project changes during the life of the project. In terms of resources, the ratios are probably more like 20%, 40% and 40%, so it is worthwhile to consider the aids available to improve performance in these latter stages.

Real-time systems introduce a number of special problems, and this paper discusses them under two main headings. Section 2 deals with testing, i.e. getting rid of bugs before the system goes live. Section 3 deals with diagnostic aids, i.e. finding the bugs which were not got rid of before the system went live.

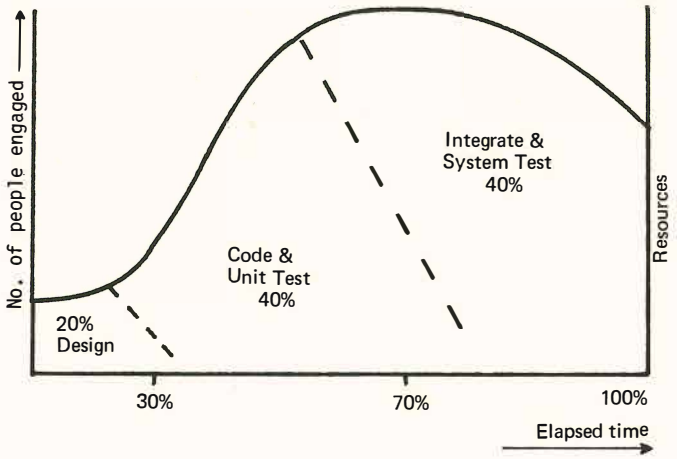


Fig. 1 Time dependence of resource utilisation

2.1 Test environments

It is common practice to create a test system to facilitate the unit testing of individual components and also the testing of functional packages. Such a tool has been described by McKenzie and Weston [2]. While not peculiar to real-time programming, it is particularly valuable for this because bugs which are not found at the unit testing stage can be very difficult to find at a later stage.

A particular point is that the test system should not zero out blocks of core before they are given to the program under test, if this will not happen in the live environment. It is even better to overwrite the block with some arbitrary pattern when it is returned to the system. This will help to detect errors resulting from references to data that is no longer available. One should generally try to prevent future trouble that could arise from implicit assumptions about control program interfaces.

The use of programming standards and conventions, for example in the use of registers, is similarly not special to real-time. Their use is essential however because of the large number of interactions that can take place.

2.2 Special hardware

In process control and similar applications, one is frequently dependent on special hardware such as analogue to digital converters, non-standard interfaces, etc. A good general rule is not to trust it – the combination of untried software and untried hardware can be a nightmare.

When starting system testing under such circumstances it is often useful to drive the programs using simulated inputs. The routine which services the special device is replaced by a dummy which supplies predetermined values. A good example of this used the standard service routine to issue a 'read' and service the interrupt, but it then ignored the data so obtained and took values read in from the disk. In this way valid timings were obtained, but one could also reproduce a test run without having to worry about whether the hardware was behaving consistently. Of particular value was the ability to introduce incorrect values in a controlled way, in order to check that the programs behaved correctly in error situations.

On the output side the same thing applies: the fact that equipment is not behaving in the way that it should, does not necessarily mean that the wrong values are being output by the program. It is therefore worthwhile to have a means of recording the values being output to a device.

One of the most sophisticated simulations of this type has been used in the Concorde fatigue test, where independent computers are used for control and monitoring. To test the system out, the computers were run 'back-to-back', with the output from the control computer fed directly to the monitoring computer.

As well as providing simulators to drive the software, it is also useful to have simulators to drive the hardware. This mean essentially simple programs to exercise the hardware and record the results. Such programs often have an extended life for diagnostic purposes long after the initial testing stage, but they are often best devised by a hardware expert who understands the way the hardware works. A further point to be borne in mind is that the diagnostic programs should be able to be time-shared with other work, or one can find that the whole computer is being used to check out one minor device.

2.3 Terminal driven systems

In the case of systems with a large number of on-line terminals, one has a different hardware problem. Either the hardware is not available when the system is being developed, or, if it is, one does not have the staff to drive all the terminals at once. Even if one had, there would again be the problem of lack of reproducibility, so again one has to provide a means of simulating inputs.

On one such system, it is possible to run card based tests of the version under development, concurrently with terminal usage of the live system. For this particular case, only single-thread testing is currently available, but multi-thread testing is desirable. The problem here is that concurrent tasks will not always terminate in the same order on successive runs, due, for example, to differences in disk arm movement. It is especially important to check that the correct action is taken when a task fails to obtain a resource. This is a special case of the general testing of limit conditions and should include looking for 'deadly embrace' situations. Volume testing of any exclusive read facility is desirable, while dealing with the maximum number of simultaneous transactions will help to show up any non-reentrant coding.

2.4 Error recovery

An important area that is often insufficiently tested is that of error recovery and restarting. The problems of providing a 'warm restart' can easily be underestimated, and an error arising during this procedure can be a major difficulty. Reconfiguration after an error must be checked out, and a requirement to run in degraded mode implies that such running must be tested.

It is often necessary to provide a file recovery program. This enables records that have been lost or corrupted to be repaired. When such a program operates on records that are chained together, it is particularly important to test that no harm comes from an attempt by another program to access a record which is currently undergoing repair.

Finally, it is necessary to test the recovery from hardware errors as well as software errors. For example, the action taken in the event of a disk read failure is often not tested until such a failure occurs. To produce a reliable system, it is necessary to check out all such actions (perhaps by simulation) before the error arises in practice.

3.1 Tracing

One of the simplest, but often the most useful debugging aid is a list in order of, say, the last 20 routines called before the crash. If one has a standard method of entering routines, there is very little overhead involved in storing in a circular buffer a code number for each routine as it is entered. It is usually worthwhile to keep such a list for each task.

A more elaborate facility is a dynamic trace that can be activated or deactivated selectively from a terminal. Its output can be written to magnetic tape for subsequent off-line analysis. Again the facility has to be built into standard entry and exit procedures. It involves some overhead, but its great advantage arises if it can be initiated selectively when there is trouble, without the need for any recompilation.

Sometimes the only way to find a bug is to introduce snapshots in the routine where the trouble is thought to be. This involves recompilation of course, but can be made easier if a standard routine exists which can be simply called.

3.2 Consistency checks

With a multi terminal system, one can differentiate sharply between errors which affect one user, and errors which affect the whole system. For errors in the first category it is usually possible to 'kill' the user concerned, but to keep the rest of the system working. Errors in the second category usually bring the whole system down. It is therefore good practice to carry out consistency checks on all data handed to the control programs, and if possible to carry out the checks at as early a

stage as possible in the application programs. Failure of such a check results in a selective dump of information about the particular user and the termination of his task.

A typical check can make use of a record identifier. When reading in a record from disk an identifying character is checked to ensure that the record has not been corrupted. Equally, before writing out a record, the character is checked as a protection against the record having been overwritten in core. Similarly, it is possible to have a block identifier. When a program is given a core block by the control program, it can check that its address is in an acceptable range, and also check a character in the block which indicates its size. In this way one can avoid problems arising from blocks being put onto the wrong lists when they are released.

Again, when updating a record the program doing so must have exclusive access to the record, i.e. it must 'hold' the record. It is possible to keep a check on exclusive reads, and ensure that this is the case before updating.

In designing a real-time system, redundant information of this type should be inserted deliberately to provide additional checking.

3.3 Monitoring

One of the problems with the consistency checks mentioned in the previous section is that the damage may have been done some considerable time before it was detected, with the result that it may be very difficult to find the cause of the trouble. To overcome this, a monitor can be introduced, which carries out checks at regular intervals, to try to detect corruption as soon as it occurs. The best chance of pinpointing the fault will be if the check is carried out on every subroutine entry, but it is adequate to check each time a task is scheduled because another has terminated or gone into a wait. Under these circumstances the task causing the trouble can be determined uniquely, and hence from the trace one can obtain the sequence of routine calls responsible.

The information to be checked can be anything which is known to become corrupted. Typically this might be a chain of core blocks each of which contains pointers to the next one and the last one. It is possible to chase down such chains, carrying out a consistency check, and when an address is incorrect it is known that overwriting has occurred. This is another example where use is made of redundant information designed into a system. In one case, the use of this technique rapidly identified a major bug which had been causing trouble but had remained unsolved for several months. It also detected four other bugs whose presence had been suspected but not identified for an even longer period.

3.4 Footprints

When a common data base can be updated by many different application programs there are always garbage disposal problems, but more serious are the problems of records which have been disposed of when they are still required. To check on this it is possible, each time a record is released, to log information about the program doing the releasing. In this way one can obtain evidence about programs which release records that are still required.

4 Conclusion

The above has been in the nature of a review of testing and diagnostic aids. It is hoped that it will be of some use as a check list, and also perhaps encourage the dissemination of other ideas that could be generally used.

1. ARON, J.D., 'Estimating resources for large programming systems', NATO Science Committee, Software Engineering Techniques – report on Conference, Rome (October 1969).
2. McKENZIE, K.I., and WESTON, P.J., 'An extended toolkit for PDP-10', DECUS Europe 6th Seminar, Munich (September 1970).

Discussion

Q. Which computers did you use for the Concorde fatigue test, and which languages did you use to program them?

A. There are three computers. Control of the loads is by a PDP-8. Control of pressures, temperatures, etc. is by another PDP-8 with special hardware and software supplied by Kent Instruments. The monitoring computer is a 48K word DEC system 10 (PDP-10).

The most difficult programming is on the monitoring computer. We did detailed design of the programs in PL/1, and they were translated by hand to Macro-10 by a more junior programmer, who also tested them. This worked fairly well because two people worked on every program, and there was some element of competition because the junior tried hard to find logical errors in the design given to him. It was not completely successful because corrections tended to be made to the Macro-10, without the PL/1 being updated.

Real-time programming using a real-time language of intermediate level

B. EICHENAUER
Elektronik System GmbH, München, W. Germany

1. Introduction

The higher programming languages such as ALGOL 60, FORTRAN and PL/1 for digital computer systems used in business, technology and science have enabled users with little knowledge of the systems themselves to use them cost-effectively in practice. On the other hand, users who wish to solve real-time problems by digital computers must have comprehensive experience in handling their particular types of machines, and have to spend much time in formulation and debugging of programs.

The last few years have seen several attempts to simplify the use of digital computers in the field of real-time applications. Today, there are essentially three views on how the difficulties can be overcome.

The first way, pursued since the beginning of the sixties, involves the development of special-purpose packages [1, 2, 3] and problem-oriented languages [4, 5, 6, 7] for special fields in real-time applications.

Experience in the use of special-purpose packages and of problem-oriented languages shows that differences between problems in one application field are usually so substantial that only a small part of the field for which a package or a language was made can be covered. With the development of technology, packages and problem-oriented languages have to be modified, resulting in a great number of packages, problem-oriented languages and dialects of these languages. In the field of automatic check-out, for instance, there are at least 30 languages and dialects, but to my knowledge not one of these languages can completely formulate the simple function test problem treated below on language level.

The second way to overcome the difficulties involves the development of intermediate-level languages like CORAL66 [8], allowing only for the algorithmic program parts which can be translated in efficient code. In these languages most of the features relevant to real-time programming, i.e. tasking, timing and process-I/O, are left quite open. Real-time programming is carried out by calling the functions of the target-computer operating system.

This means of simplifying the problem may allow faster formulation of programs, as compared with assembler programming, but the main disadvantages of current methods are still present. For instance, every user must know the actual surrounding manufacturer software nearly as well as when using assembler languages. The documentation and the portability of programs is restricted by the dependence of the target computer.

The third method, used, for example, in INDAC8 [9], PAS1 [10] and PEARL [11], consists in adding to algorithmic language features of intermediate level a sufficient set of basic real-time and I/O features of comparable language level. As the algorithmic features of intermediate-level languages such as ALGOL or FORTRAN can be used to formulate any numerical algorithms, the basic real-time features should allow separation of application problems in loosely connected processes [12], and formulation of any correspondence of a process with the surrounding world.

A detailed explanation of the above set of language elements cannot be given here. Instead, we describe the solution of a simplified automation problem in the process and experiment automation real-time language PEARL, which we believe to be the most advanced language for real-time programming at present.

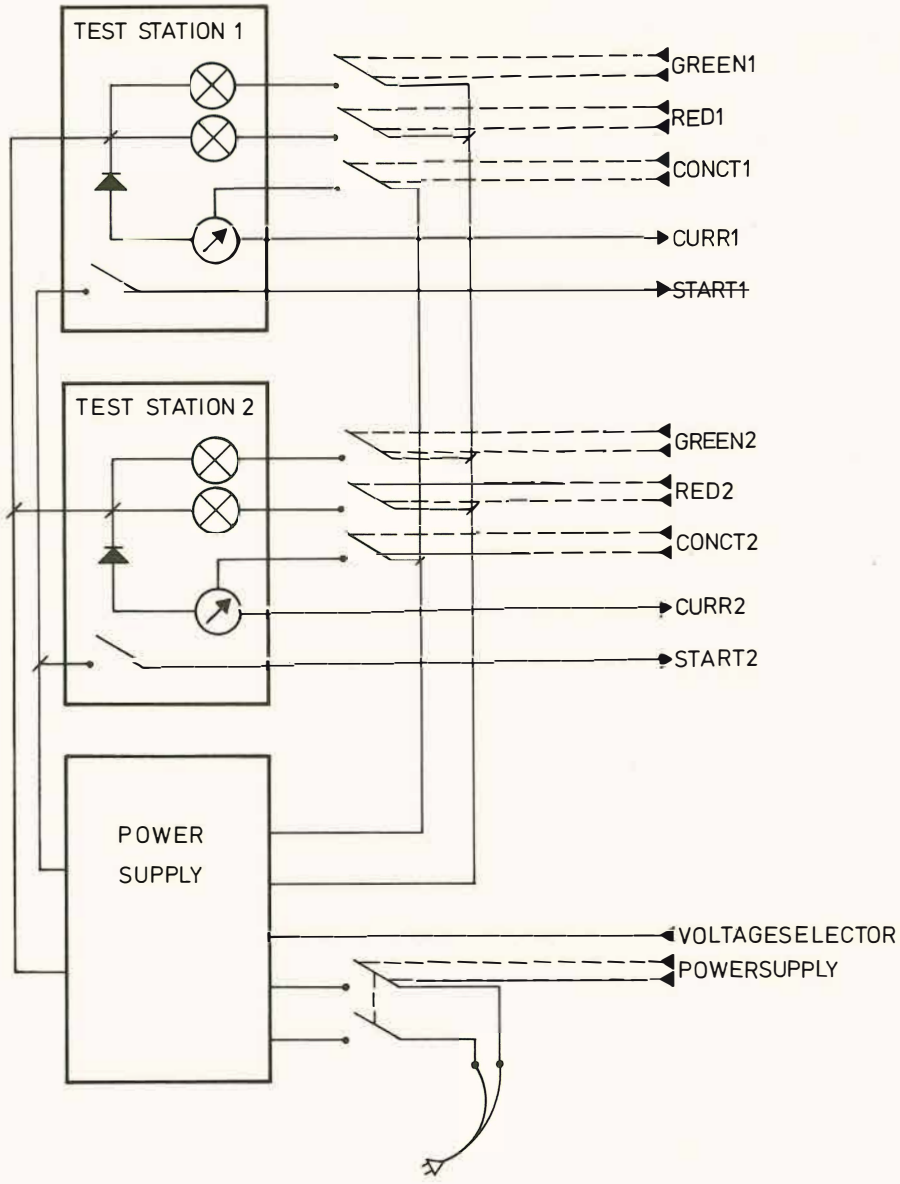


Fig. 1 Test installation

2. Programming example

The example is taken from the PEARL report [11].

2.1 Problem

Figure 1 shows a test installation consisting of two identical test stations and of a programmable power supply. The function test for four different types of diode has to be carried out at the test stations.

After the start of the automation program by the operator the computer switches on the power supplies of the two test stations (output to POWERSUPPLY) and adjusts the reverse voltage depending on the type of diode to be tested (output to VOLTAGESELECTOR).

The first step of the function test is to connect the diode to one of the test stations. Then the start button of the chosen test station is pressed to start the test program of the station (input from START1 resp. START2).

The test program applies the reverse voltage to the diode (output to CONCT1 resp. CONCT2), measures the reverse current (input from CURR1 resp. CURR2) and removes the reverse voltage (output to CONCT1 resp. CONCT2).

If the measured reverse current exceeds a type-dependent maximum, a red lamp on the instrument panel has to be switched on for 2 sec (output to RED1 resp. RED2). Otherwise a green lamp is switched on (output to GREEN1 resp. GREEN2).

The number of diodes not operating and the total number of diodes are counted for every test station and must be recorded at the end of the test period, or the type of diode must be changed.

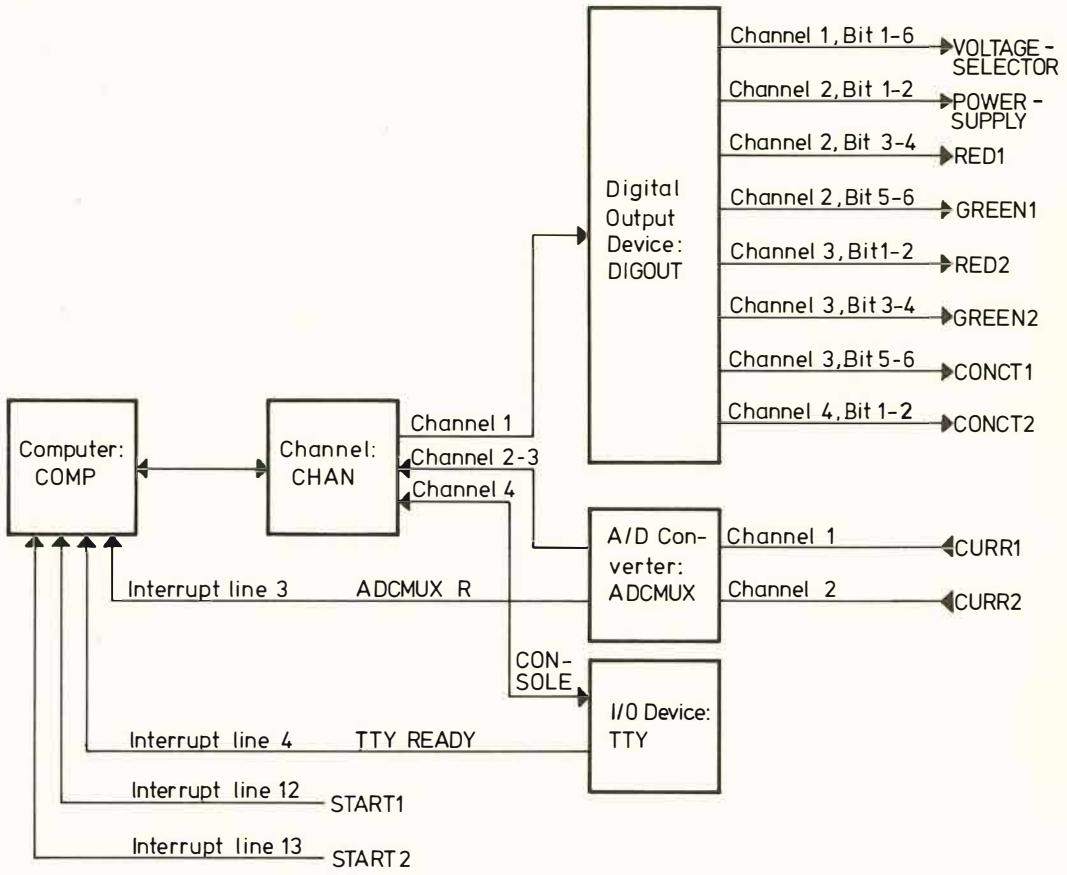


Fig. 2 Configuration of process periphery

2.2 Configuration of the computing system

Figure 2 shows the configuration of a hypothetical computing system servicing the test installation.

2.3 Automation program

The PEARL automation program consists of two program sections – the so-called SYSTEM-section and the PROBLEM-section. In the system section the configuration of the computing system shown in Fig. 2 is described. To allow for configuration-independent programming of automation algorithms in the problem division, all process end-points have to be identified by names.

The automation algorithm for the function test problem has been divided into three parallel resp. quasi-parallel executable program parts named TESTPROB, TEST1 and TEST2.

The main-task TESTPROB is activated by the operator by input to the standard-I/O-device CONSOLE (e.g. with the control statement: ACTIVATE TESTPROB PRIO 3).

To allow for parallel working of the test stations the service programs for the stations are isolated by introducing the tasks TEST1 and TEST2. TEST1 resp. TEST2 will be activated after interrupt START1 resp. START2 has occurred.

```
MODULE LIBRARY DIODETEST;
      /* PEARL EXAMPLE PROGRAM: FUNCTION TEST PROBLEM */
SYSTEM;
/*  CONNECTION OF STANDARD DEVICES:  */
   COMP              <->   CHAN;
   CHAN  *  1         ->   DIGOUT;
         *  2 * 1,12 <-    ADCMUX;
         *  4        <->   CONSOLE:  TTY;
```

```
/*  PROCESS END POINTS:  */
   VOLTAGESELECTOR:      <-     DIGOUT * 1 * 1,6;
   POWERSUPPLY:          <-            * 2 * 1,2;
   RED1:                 <-            * 2 * 3,2;
   GREEN1:               <-            * 2 * 5,2;
   RED2:                 <-            * 3 * 1,2;
   GREEN2:               <-            * 3 * 3,2;
   CONCT1:               <-            * 3 * 5,2;
   CONCT2:               <-            * 4 * 1,2;
   CURR1:                 ->    ADCMUX * 1;
   CURR2:                 ->           * 2;
/*  CONNECTIONS OF INTERRUPTS:  */
          CONTACT(3)     <-     ADCMUX * R;
          CONTACT(4)     <-     TTY * READY;
  START1: CONTACT(12)    <-     ;
  START2: CONTACT(13)    <-     ;
/*  END OF SYSTEM DIVISION  */
PROBLEM;
DCL (START1,START2) VAL INTERRUPT;
TASK TESTPROB GLOBAL HANDLE:
  BEGIN
  DCL           /*  STANDARD OUTPUT DEVICE:  */
      STANDARDWRITE VAL DVC = CONSOLE,
                /*  VARIABLES TO COUNT THE NUMBER OF TESTOBJECTS:  */
      (TOTAL1,TOTAL2,NOGO1,NOGO2) INT := 0,
                /*  VARIABLE TO STORE THE UPPER LIMIT OF REVERSE CURRENT:  */
      TYPECURR INT,
                /*  STRINGS TO ADJUST THE POWERSUPPLY TO REVERSE VOLTAGE:  */
      ADJUST(1:4) VAL BIN(6) = ('1000'B,'100'B,'10'B,'1'B),
                /*  UPPER LIMITS OF REVERSE CURRENT:  */
      MAXCURR(1:4) VAL INT = (50,35,28,20),
                /*  COMMON CHARACTER STRINGS:  */
      TEXT1 VAL CHAR(13) = 'TEST STATION ',
      TEXT2 VAL CHAR(28) = ': NUMBER OF TESTED DIODES = ',
      TEXT3 VAL CHAR(27) = 'NUMBER OF INFERIOR GOODS = ',
                /*  COMMON FORMATS:  */
      COM1 VAL FORMAT = F'S(13),S(1),S(28),,L',
      COM2 VAL FORMAT = F'(16)C,S(27)';
              /*  TEST PROCEDURE:  */
  DCL  TEST PROC REENTRANT =
    ((UAMP,CONNECTOR,REDLAMP,GREENLAMP) VAL DVC,
           (TOTALNUMBER,NOGONUMBER)        INT )
    BEGIN  DCL CURRENT INT;
           TOTALNUMBER := TOTALNUMBER + 1;
              /*  MEASURE REVERSE CURRENT:  */
        MOVE '1'B TO CONNECTOR;
        MOVE UAMP TO CURRENT;
        MOVE '10'B TO CONNECTOR;
```

```
            /*  CONTROL OF REVERSE CURRENT:  */
        IF CURRENT > TYPECURR OR CURRENT < 2 THEN
            /* NOGO BRANCH: */  MOVE '1'B TO REDLAMP;
                                DELAY 2 SEC;
                                MOVE '1Ø'B TO REDLAMP;
                                NOGONUMBER:=NOGONUMBER + 1;
                                          ELSE
            /* GO BRANCH: */    MOVE '1'B TO GREENLAMP;
                                DELAY 2 SEC;
                                MOVE '1Ø'B TO GREENLAMP;
                                                FI;
          END  /* OF TEST PROCEDURE  */ ;
            /* DECLARATION OF TASKS TO CONTROL THE TEST STATIONS:  */
TASK TEST1: CALL TEST(CURR1,CONCT1,RED1,GREEN1,TOTAL1,NOGO1);
TASK TEST2: CALL TEST(CURR2,CONCT2,RED2,GREEN2,TOTAL2,NOGO2);
            /* GET THE TYPENUMBER OF DIODE UNDER TEST:  */
TYPEDEF: WRITE (DATE,TIME,'TYPENUMBER := ')
                        FORMAT((2)(,(2)C),S(14));
        READ TYPECURR FROM CONSOLE FORMAT(,(2)L);
        IF TYPECURR < 1 OR TYPECURR > 4 THEN
           WRITE 'INPUT ERROR: UNDEFINED TYPE'
                        FORMAT((1Ø)C,S(34),(2)L);
           GOTO TYPEDEF;                          FI;
            /* SWITCH ON AND ADJUST POWERSUPPLY: */
MOVE '1'B TO POWERSUPPLY;
DELAY 2 MIN;
MOVE ADJUST(TYPECURR) TO VOLTAGESELECTOR;
            /* STORE LIMIT OF REVERSE CURRENT AND INDICATE TEST BEGIN:  */
TYPECURR := MAXCURR(TYPECURR);
WRITE (TIME, 'TEST BEGIN') FORMAT (,S(1Ø),(2)L);
            /* CONNECT START INTERRUPTS TO CONTROL TASKS: */
ON START1 ACTIVATE TEST1;
ON START2 ACTIVATE TEST2;
            /* SUSPEND MAIN TASK TESTPROB UNTIL OPERATORS' COMMAND: CONTINUE TESTPROB;  */
SUSPEND EXEPT TEST1,TEST2;
            /* RECORD OF TEST RESULTS:  */
WRITE (TIME,'  TEST RESULTS:',TEXT1,'1',TEXT2,TOTAL1,TEXT3,NOGO1,
                            TEXT1,'2',TEXT2,TOTAL2,TEXT3,NOGO2)
           FORMAT(,S(15),(2)((2)L,COM1,COM2),P);
END  /* OF MAIN TASK TESTPROB  */ ;
MODEND  /* OF MODULE DIODETEST  */ ;
```


1. '1800 process supervisory program (PROSPRO/1800)', IBM no. H 20-0473-1 (1968).
2. BATES, D.G., 'PROSPRO/1800', IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. IECI-15, No. 2, pp. 70-75 (December 1968).
3. 'BICEPS summary manual/BICEPS supervisory control', GE Proc. Comp. Dept., A GET-3539 (1969).
4. 'ATLAS abbreviated test language for avionics systems' ARINC Specification 416-1, Aeronautical Radio Inc. (June 1969).

5. METSKER, G. S., 'Checkout test language: an interpretive language designed for aerospace check-out tasks', Fall Joint Comp. Conf., pp. 1329-1336 (1968).
6. 'Bendix OPTOL programming system', The Bendix Corporation, Teterboro, New Jersey (March 1968).
7. 'PLACE The compiler for the programming language for automatic checkout equipment', Battelle Memorial Institute, Columbus Laboratories, Technical Report AFAPL-TR-68-27 (May 1968).
8. 'Official definition of CORAL66', Inter-Establishment Committee for Computer Applications (February 1970).
CALLAWAY, A. A., 'A guide to CORAL programming', Royal Aircraft Establishment, Technical Report 70102 (June 1970).
9. 'INDAC8', Digital Equipment Corporation, Maynard, Mass.
10. 'PAS1 Prozess-Automationssprache 1', BBC Mannheim.
11. BRANDES, J., et al., 'PEARL: A proposal for a process and experiment automation real-time language', to be published by Projekt PDV, GFK Karlsruhe.
12. DIJKSTRA, E.W., 'Co-operating sequential processes', Programming Languages, F. Genuys (Ed.), Academic Press, London (1968).

Workshop on special systems and system features

Chairman: Dr H-J. TREBST
Erlangen, W. Germany

The discussion centred on problems of testing software, and the kinds of testing aids which can be offered, such as simulation of the environment in process control (either in the same or a different computer), checks of source language semantics and conventional debugging.

E. In checking against garbling of data, we have kept a check sum on a block of data which is tested at regular intervals. This works well for data which is either not changed much or usually changed in complete blocks, but is not so good when one or two words of a block may be changed more frequently than the rest of the block.

Y. The method described by Professor Wettstein is a high-level analogue of the conditional assembly methods used for system generation by at least IBM and DEC. It is very hard to debug these, because each possible configuration has to have an operating system adapted for it, and then checked on an appropriate physical configuration. For example, on our PDP-11 with RPO2 disk, we assembled the adaptable disk handler for DOS using the appropriate code for the RPO2, and the resulting program in assembly language had an error in it – obviously that had never been checked before the adaptable disk handler was distributed. This does not mean that I doubt the value of adaptive operating systems, but that they are even more difficult to debug than non-adaptive operating systems.

V. Perhaps the very point of slipping from high-level language to assembly language in debugging is that it forces the programmer to view his program in a different way.

H. The question of detecting faults is not necessarily one of "Which level of language do we use?" It is largely a matter of discovering when it is possible to detect possible faults – at compile time or at run time.

For example, consider the program

```
{local a ; a:=0;  a:=a+1,  a:=a+1}
```

where the comma denotes that the two statements 'a:= a+1' are done in parallel. This is not deterministic, I cannot imagine a situation in which this program is correct. Should we therefore have it flagged by the compiler as containing a semantic bug?

L. There is considerable structure within a group of interacting processes which may be deduced from the programs. The testing of the interactions between these processes is itself a real-time problem which cannot be truly tested in off-line environment. The program preparation facilities for real-time work should include:

1. Similation of interactions between processes.
2. Flagging possible conflicts which could arise at run time.

These are analogous to automatic flow-chart generation for single process programs.

E. Debugging is never fully complete. We have to decide some possibly arbitrary criterion to determine when to regard it as debugged. Perhaps we can say it has reached this stage when the customer pays the bill.

CLOSING ADDRESS

This is now the end of another European seminar on real-time programming. It is the second of its kind and the idea of a relatively small group which can have a lot of good discussion seems to have proven successful.

We could see that the general nature of the problems has not changed so much since the last seminar, but it was remarkable how much emphasis was laid this time on the areas of testing and debugging of real-time systems.

I would like to point out that this time we could welcome some guests from overseas and it looks as if this seminar could develop into a fully international institution. I am also glad to tell you that the 'tradition' of this seminar will continue as there are plans to have the Third Seminar at the Euratom Research Establishment at Ispra (Dr Metzdorf).

In closing, I would like to thank all those who made this seminar possible, especially the Physics Institute for providing rooms and other facilities and my colleagues who helped in the preparation of this seminar. Last but not least I thank all the participants for their efforts to come to Erlangen and for their contributions both in paper and discussions.

AUTHOR INDEX

EDITORS

IAN C. PYLE, MA, PhD (Cantab)
has been working with computer software since 1956, particularly concerned with programming systems (Hartran on the ICL Atlas Computer), multi-access systems (HUW on the IBM 360 computer at Harwell), and multi-computer command and control systems. He is now Professor of Computer Science at the University of York.

PETER ELZER, MA (Erlangen)
has been working with computers since 1966, at the Physics Institute of the University of Erlangen. His early work concerned the development of a programming system for application to nuclear physics experiments. He was one of the initiators of the PEARL project and is now leader of a PEARL implement team at the Physics Institute. Since 1971 he has now been involved with the LTPL project, of which he is now the European Chairman.

TRANSCRIPTA BOOKS