Real time languages for process control

J. G. P. Barnes

Imperial Chemical Industries Limited, Bozedown House, Whitchurch Hill, Reading, Berkshire

This paper describes in outline a programming language designed for the description and implementation of real time programs on medium-sized process computers.
(Received May 1971)

The Central Instrument Research Laboratory of Imperial Chemical Industries installed a Ferranti Argus 400 in 1966 for the control of multiple laboratory experiments. An operating system was devised and implemented by Cobb, Magee, and Williams (1967); the main purpose of this system was to provide each of several users with a pseudo multi-processor capability with storage protection from other users (see also Magee, 1970). The technique evolved by Cobb to provide storage protection by software in the absence of any assistance from the hardware involved the effective interpretation of all dynamic store and branch instructions in the users programs; these instructions were then costly in space and time. It is against the background of this operating system that the development of SML (Barnes, 1969), the laboratory's first attempt at a real time high level language, must be seen.

SML (Small Machine Language) was a simple variant of ALGOL but with static storage allocation. It included a scaled fixed point facility in addition to the normal ALGOL types. Multi-tasking was provided via a set of standard procedures (activate, delay, waitfor, etc.) essentially as proposed by Benson, Cunningham, Currie, Griffiths, Kingslake, Long, and Southgate (1967). Normal ALGOL parameter mechanism was provided; this parameter mechanism was re-entrant but coding was otherwise not re-entrant except inasmuch that individual procedures could be executed in parallel.

A compiler to translate SML text into Ferranti Argus Initial Orders Mk 3 was written in KDF9 ALGOL and programs were originally thus compiled off-line. The compiler was later rewritten in itself and bootstrapped to provide an on-line compiler.

Experiences with SML on experimental plant in the author's laboratory were entirely successful (Brisk, Davies, and Jones, 1969); subsequently the language was also used for programming the control software of production plant (Garside, Gow, and Perkins, 1969). In the latter case, however, the operating system itself needed considerable modification and being written in an assembly language this was a non-trivial exercise.

This earlier work provided the background to a new project whose objectives are to develop a language/compiler/package system for general use in process control and similar on-line applications. It was decided to undertake this project in stages, the language RTL/1, whose description now follows, is the result of the first stage.

Terminology

In this paper, the word 'thread' is used to describe an identifiable execution of a set of instructions. Thus, broadly speaking, a thread is defined by the sequence of values taken by a program counter during the execution of a logically coherent job by a processor.

A conventional computer with a single processor can be considered to be obeying at all times one unique thread. It is, however, usually more convenient to consider the various individual jobs as being serviced by separate threads and to think of the scheduler as converting the one actual processor into several pseudo-processors by the use of some algorithm.

The concept of a thread is called a task in PL/1 whereas we use the word task to designate a set of instructions themselves rather than their execution.

Criteria for RTL/1

The following were seen as the main criteria for the design of RTL/1:

1. Algorithmic features available in high level languages such as FORTRAN and ALGOL should be available and should be implemented as efficiently as is usual in these languages.
2. The dynamic creation and synchronisation of threads should be simple.
3. Subroutines should be re-entrant so that multi-thread programming is simplified.
4. It should be practicable to write large parts of a conventional executive in the language.
5. It should be possible to change, in a modular fashion, parts of a program complex while it is actually running.
6. No unpredictable timing problems should arise as might happen if a conventional garbage collector were used for storage control.
7. An RTL program should be secure.

Overall structure

An RTL/1 program complex consists of a collection of sections; a section may be of four types:

1. Task: this is a read-only block of coding; it is similar in structure to a parameter-less type-less ALGOL procedure.
2. External procedure: this is a re-entrant read-only block of coding similar in structure to an ALGOL procedure.
3. Data set: this is a static block of data which may be read from or written to; it is thus similar to FORTRAN labelled common.
4. Stack: this is an area used as workspace for the storage of links, dynamic variables, scratchpad and other housekeeping items; it can only be used by one thread at a time and when not in use is unstructured. It has a definite length.

These four types of section are interwoven as follows:

Whenever a new thread is created a stack is nominated as workspace and a task as coding to be obeyed. Later operations on the thread may be made by reference to its stack. The coding of the task may include calls of external procedures which may in turn call other external procedures. Both tasks and external procedures may access variables in data sets by the use of appropriate statements.

External procedures and tasks may have internal procedures declared within in the normal ALGOL style, but in order to simplify the implementation of RTL/1 the number of lexicographic levels of declaration has been restricted to four in all.

Data types

The simple data types provided are boolean, char, integer and real. The integer type has been extended to provide scaled fixed point arithmetic with automatic scaling provided by the compiler. The char type is simply an 8-bit unsigned integer which provides facilities for character handling. There is also the section type. A section value is essentially a pointer to an entry in a dictionary containing the information about the various sections in the complex.

The only compound structures are arrays, multidimensional arrays being treated as arrays of arrays. Sufficient information concerning the bounds of arrays is retained at run time to allow for subscript checking, but the philosophy of only checking store operations (and not fetch, etc.) has been adopted. It should be noted that strings are implemented as read-only one dimensional char array constants.

The parameter mechanism is of the usual ALGOL type; formal parameters may be scalars by value or reference, arrays, labels and internal procedures. Actual parameters correspond naturally with the notable addition of the ability to pass a slice of an array as an actual parameter to a formal array.

Scheduler

A primitive scheduler has been written independently of RTL to control the allocation of processor time to the various threads in existence at any time.

Each thread has the following attributes:

1. Priority—a signed integer.
2. Stop/start flag.
3. Free/suspend flag.
4. Address.

A thread is eligible for service if and only if the stop/start flag is in the start state and the free/suspend flag is in the free state. The scheduler runs the highest priority eligible thread with a 'round-robin' provision for several threads of the same priority.The stop/start state was intended for use by a roll-in roll-out algorithm but has instead tended to be used for program testing. The free/suspend flag is used to co-ordinate co-operation. A suspension (which may only be self imposed) is either a delay for a specified number of time units or a wait for a specified (software) event. The address attribute is interpreted by the scheduler as the scratchpad save area.

Access to the scheduler from RTL/1 programs is from procedure calls via an interface package. This interface interprets the scratchpad address as the stack address (so that the low end of the stack is used for saving the registers) and translates stacks into thread numbers.

At the scheduler level the basic requests include:

```
makethread (address, priority) -> thread
start (thread)
stop (thread)
killme
delay (time units)
wait (event)
stimulate (event)
```

whereas at the RTL/1 level the first three become:

```
makethread (stack, task, priority)
start (stack)
stop (stack)
```

The makethread procedure requests a new thread from the scheduler with the stack as address and with given priority; the thread number returned is then stored in a fixed location in the stack for future reference and the copy of the program counter in the scratchpad is set equal to the task address. A subsequent start(stack) statement will issue a start(thread) request to the supervisor to set the thread going; it will then enter the required task.

In order to kill a wayward thread it is necessary to induce it to kill itself; this is achieved by altering the copy of the program counter in the scratchpad to the address of a termination routine.

Cross linkage

The cross linkage between all sections is via a dictionary which includes the following attributes for each section:

1. Name.
2. Core address.
3. Usage count.
4. Type and length.
5. Status.
6. Disc address.

The status indicates whether the section is in core, on disc, both or neither. The usage count indicates the number of threads currently accessing the section; thus in the case of an external procedure this count is incremented on entry and decremented on exit.

The indirect linkage provided via the dictionary enables the simple construction of systems from separately compiled sections. Various supervisor procedures have been written to manage the sections. These include:

1. Load; checks section not loaded and then loads a new copy to specified address.
2. Unload; checks usage count zero and then changes status to not in core/disc.

and various core to disc commands.

Implementations

RTL/1 has been implemented on the Ferranti Argus 500, ICL System 4 and GEC-AEI M2140.

The Argus 500 implementation is the most complete and two distinct systems have been written. There is a normal essentially core-based system and a disc-based system with automatic roll-in roll-out features. The latter has been used with success to program the software for controlling two ICI production plants (Law and Moloney, 1971). The System 4 implementation is intended for off-line compilation and single thread program testing. Programs run under the standard J operating system and none of the multithread features have been implemented.

The M2140 implementation is incomplete and currently suspended.

Experiences

The use of RTL/1 to date has indicated that it has all the advantages normally associated with a high level language as compared with an assembly language. Features of RTL/1 which have proved of great value are:

1. Char type.
2. Re-entrant code.
3. Modular recompilation.

On the other hand some aspects of RTL/1 have not proved to be wholly satisfactory;

1. The concept of a section as a type and the dynamic inter-section linkage has caused the language to encroach on the system.
2. The ALGOL block structure has imposed rather more overhead than is really desirable.
3. The unpredictable stack size associated with dynamic storage caused difficulties for the system designer.
4. The fixed point variables have not been useful.

In the light of experiences with SML and RTL/1 a successor language (RTL/2) is being designed and is intended to be the final version of RTL. It differs significantly from RTL/1 and is judged to possess the correct balance of features needed for

real time programming. It is planned to publish and implement RTL/2 during 1971.

Acknowledgements

The author wishes to acknowledge the effort of the RTL Project Team in implementing the compilers and systems programs (H. R. A. Brown, M. H. Lloyd, R. J. Long, R. N. Magee, C. Miles, T. Mullett, and K. G. Whitehead). Particular thanks are also due to the ICI Application Team (led by W. M. Law) who elected to use RTL/1 on live process applications and thus helped to contribute the essential practical feedback now being incorporated in RTL/2.

References

BARNES, J. G. P. (1969). SML User's Guide. Imperial Chemical Industries Limited, Central Instrument Research Laboratory, Technical Note JGPB/69/35.

BENSON, D., CUNNINGHAM, R. J., CURRIE, I. F., GRIFFITHS, M., KINGSLAKE, R., LONG, R. J., and SOUTHGATE, A. J. (1967). A Language for Real Time Systems, *The Computer Bulletin*, Vol. 11, pp. 202-212.

BRISK, M. L., DAVIES, C., and JONES, M. (1969). Computer Control of Research Equipment for the Investigation of Gas-Solid Reactions, Part 2—Computer Software, *Instrument Practice*, Vol. 23, pp. 117-120.

COBB, A. J., MAGEE, R. N., and WILLIAMS, R. C. (1967). Bozedown Argus Real Time Multi-User Programming System, Imperial Chemical Industries Limited, Central Instrument Research Laboratory.

GARSIDE, J., GOW, J. S., and PERKINS, W. J. (1969). Computer Control of a Compound Fertilizer Plant, Proceedings of Nat. A.C.S. meeting, New York.

LAW, W. M., and MOLONEY, T. (1971). Computer Control of a Multitrain Batch Plant, Third IFAC/IFIP Conference, Helsinki (to be published).

MAGEE, R. N. (1970). Bozedown Disc Operating System on Argus 400—Users Instruction Manual, Imperial Chemical Industries Limited, Central Instrument Research Laboratory. Research Note RNM/70/3.

Correspondence

To the Editor
The Computer Journal

Sir,

The article 'High level languages for low level users', (Evershed and Rippon, 1971) was interesting, and I thoroughly agree with the authors' aim that program writing should be made as simple as possible. However, in detail, I find that I disagree with so many of their points that it is worth submitting alternative views (not that anyone will actually take any notice of either the article or this reply!)

The authors do not make clear the distinction between languages, and implementations of languages. They claim that they are discussing languages, but many of the points made are of implementation. Thus the complaint about Elliott 4100 ALGOL using < for *less than* but 'LE' for *less than or equal to* is surely explained by the hardware available not having a ⩽ character. Personally I should prefer < = but this is a matter of implementation, not of the ALGOL language as such.

I agree that FORTRAN input/output is absurdly, and quite unnecessarily, complicated. Almost ideal, to my mind, are the input/output procedures of the Atlas version of ALGOL 60. These are based on the equivalent procedures of Mercury Autocode and do almost everything a user could want with easily remembered simplicity. One of their great advantages is that they are, in every way, ordinary ALGOL procedures, and there is no need to remember a single special rule. I find it incomprehensible that Evershed and Rippon should prefer something like READ Z with a significant space after the READ and no brackets around the parameter.

The authors regard FORTRAN as superior to ALGOL in not requiring the programmer to declare variables. I regard this feature as a severe defect of FORTRAN. At first sight it looks attractive but experience convinces me that the disadvantages far outweigh the advantages. Two of these disadvantages are:

(*a*) spelling mistakes go unnoticed but merely lead to a program that does the wrong job;

(*b*) chaos can be caused by the trivial *fault* of starting an identifier with an inappropriate letter of the alphabet. Superior forsooth!

The authors regard the semi-colons, and the underlining of basic words, in ALGOL as avoidable nuisances. It is true that both these things do seem a nuisance until you get used to them and, other things being equal, they might be avoided. But other things are very far from equal; it is these two features that allow ALGOL to be totally *layout independent* and the advantages of that are immense.

An additional advantage of underlining is that restrictions do not have to be placed on allowable identifiers. It is all very well for the authors to be funny in claiming that one would not often wish to write

DO 25 REAL = INTEGER, GOTO, END

but *end* is in fact quite frequently used as a label in ALGOL, I have seen IF used as a variable in a published FORTRAN algorithm (the author really wanted to use F but needed an integer). Furthermore, languages are tending to introduce more and more words—a recent extended version of ALGOL has 57 identifiers forbidden. Underlining (or some equivalent device) means that the user does not have to be continually on his guard, and looking up long lists of words.

I am surprised that the authors approve of Atlas Autocode for *sensibly* using = with two entirely different meanings. The realisation that the operation of assignment is a different thing from the relationship of equality is a very important step in learning computing. The := symbol of ALGOL is thus a simplification, not a complication.

I am quite at a loss to understand why the authors should prefer

LOOP K, I TO N BY M

to ALGOL 68's

for *k* **from** *i* **by** *m* **to** *n* **do**

Among the several disadvantages of their version we have one of those fiendish arbitrary commas that are such a trap for the unwary in FORTRAN.

I am even more at a loss to imagine why they should consider the messy FORTRAN construction

DO 70 I = 1, 5, 1

as *the most satisfactory*.

In complaining that when trying to print a 7-digit integer using FORTRAN, after allowing only I6 as the format 'the programmer is rewarded with six stars! This is extraordinarily unhelpful' they should be grateful for their blessings. Most versions of FORTRAN would print the six least-significant digits of the 7-digit answer, and give no warning of anything wrong—to be unhelpful is better than to be criminally irresponsible. As long ago as 1959 Mercury Autocode output had the ideal solution—when a number is too long for the space allowed for it, print the correct answer at the expense of the layout. When, oh when, will this become standard?

Lastly, I find it surprising that the authors should have ignored, in their article, the language called BASIC. Some may like it, some may not like it, but to ignore it, in an article on simplifying programming languages, seems peculiar.

Yours faithfully,

I. D. HILL

MRC Computer Unit
242 Pentonville Road
London N1 9LB
4 August 1971

Reference

EVERSHED, D. G., and RIPPON, G. E. (1971). High level languages for low level users, *The Computer Journal*, Vol. 14, pp. 87-90.