Computer Science 773

Robotics and Real-time Control

REAL-TIME LANGUAGES

EARLY DEVELOPMENTS.

As one might expect, several programming languages have been invented for real-time control from time to time. I don't think that these are much used nowadays, but they're still interesting because they do illustrate some of the important features of real-time systems which are not catered for satisfactorily in traditional languages.

Real-time computing started off in the same way as everything else – with assembly language. Unlike almost everything else, though, a lot of it stayed with assembly language until not long ago. While there's certainly much less of it than there was, I wouldn't be too surprised to find that there was still a fair bit of assembly language going on in serious industrial work, as well as hobby computing.

The reason for this survival isn't just nostalgia; there are still real difficulties in using separate development systems to write programmes for embedded microprocessors and then downloading them, and more of the same in trying to debug the programmes once written. The fundamental difficulty is that to use a development system you have to write the programme and test it on a machine that isn't that which it will occupy when it runs, and its very hard to get the timing right that way if it's at all critical. For that sort of reason, there's always been a strongish feeling among real-time programmers that assembly language is the only *real* way to do real-time programming.

Nevertheless, higher level languages for real-time work have been available for about as long as they have been for conventional computing, and they have the sorts of advantage you'd expect from higher level systems. Timing remains a problem, but there are many benefits from using high level languages. Also like other sorts of language, there was quite a bit of experimenting in the early days before people worked out a good way to use the high level facilities in real-time computing. One can discern (or, perhaps, imagine, but I think there's a bit more than that) three lines of development in the early real-time languages :

•	Familiarity :	The language is strongly based on some well-known precursor. Compilers or other processors are readily available, there are software libraries and other supporting facilities, there is expertise about the language and its use.
•	Efficiency :	The language is designed in such a way as to promote the generation of highly efficient code.
•	Ease of programming :	The language is designed with vocabulary and structures which make it easier to encode and manipulate the entities required for real-time computing.

FAMILIAR PROGRAMMING.

The major member of this group is undoubtedly *Real-time Fortran*. For many years, Fortran was recognised as the computer language for technical computations, so it was the natural choice as the basis for an engineering development. It was also in some ways a very easy choice, because Fortran programmers were very accustomed to using specialist subroutine libraries to provide procedures useful in particular areas of work; a subroutine library providing operations useful for real-time control was therefore easy to implement and very acceptable to the practitioners. This approach leaves the Fortran language and system untouched, so there is nothing new to learn except the peculiarities of the new subroutines.

In the course of time, several versions of real-time Fortran were developed, and some are still in use. These were drawn together by the "Workshop on Standardization of Industrial Programming Languages" at Purdue University (USA) in 1972, and the resulting recommendations defined what became known as Purdue Fortran. These are interesting because they define what was seen as the extension to a conventional computer language which is necessary to make it into a real-time control language. Here's a summary of the recommendations.

The other main member of this class was real-time Basic. This was not so much a single languages as a family of proprietary languages, all implemented by stretching a conventional Basic interpreter by adding new keywords for the real-time operations. Such languages became fairly popular when minicomputers became cheap enough to buy for laboratories. Most were equipped with Basic interpreters, which were very easy to use, and extending Basic to provide a degree of real-time computing was a good selling point; you could use your computer to control simple experiments as well as just sit there doing sums.

EFFICIENT PROGRAMMING.

The concern about real-time constraints is illustrated by a small class of languages in which efficient execution was the overriding design factor. Combined with a (partly justified) suspicion about the efficiency of the code generated by compilers, this led to the paradox of real-time languages which hardly dealt with real-time considerations at all; instead, they provided a comparatively high level syntactic framework within which any serious real-time features are implemented by machine-dependent procedures or assembly language code.

Coral (several versions, with Coral 66 the best known) and RTL/2 are examples of this class of language. Syntactically, they resemble Algol and Algol 68 respectively. There isn't a lot more to say about Coral. RTL/2 has some interesting features, being very modular in design, with modules (called bricks) which could be data or procedures. (Compare objects.)

These are not to be confused with a curious organisation called RTL-Bricks, which you find quite easily if you search for RTL on the world-wide web. The web page includes this information : "I first had the idea to create RTL-Bricks while I was in college. I'm a mechanical engineer and I've always loved LEGO® Brand Plastic Building Bricks (PBB's) because I could build almost anything with them. Anyway, I stumbled on to an Internet newsgroup dedicated to them and found that all of the regulars were annoyed with LEGO® Systems, Inc. for discontinuing the sale of bulk pieces some years previously. Many of the dedicated LEGO® Maniacs wrote and signed petitions asking LEGO® Systems to reinstate bulk sales, but nothing ever became of it. .....". *RTL-Bricks stocks building blocks as produced by LEGO and lots of other companies. Weird.*

Finding it difficult to believe that the combination of RTL and bricks could have arisen by pure coincidence, I wrote to the address given and asked. Yes, came the reply, it really is a coincidence : "RTL is an acronym for the newsgroup rec.toys.lego where a great deal of my customers hang out. The bricks part is because the standard LEGO(R) Brand pieces closely resemble bricks."

A specimen of RTL/2 appears on the next page.

EASY PROGRAMMING.

Perhaps the most interesting group of languages are those which are designed from the outset to cover the requirements of real-time programming. In these languages, the vocabulary and structures provided for data and control are intended to suit the real-time application area. Examples of programmes in two such languages, Pearl and Iliad, follow.

AN ILIAD PROGRAMME.

The programme reads data from a counter at regular intervals, and prints a list of the collected data either when the list grows to a certain length or on request. It also receives and deals with certain instruction from a typewriter interface. This is a diagram of the plant :

The programme appears on the following pages. It is composed of three tasks, which run concurrently.

The first task is the main task. It looks after the typewriter interface, and also sets up the other two tasks.

The data logger task prints the contents of the log buffer when it becomes full, or when there's a request from the keyboard.

The reader task controls the regular acquisition of the current value of the counter, and resets the counter.

A PEARL PROGRAMME.

The Pearl programme controls the pressure in a pipeline. Under normal conditions, the average of two pressure readings is used as the signal to open or close a valve. (Compare the example for the PLC.) There's also a high pressure alarm signal, which must trigger the opening of a discharge valve. There's a diagram on the next page.

The programme is on the next page. It begins with some declarations, which resemble the Cobol data division in specifying how the external data are mapped into internal variables. This diagram of the hardware configuration might help to interpret the system declarations.

```
MODULE;                         /*EXAMPLE PRESSURE CONTROL*/

    SYSTEM;
        MULTIPLEXOR             <->   CPU*5;
        DIGITALOUT(0)           <-    MULTIPLEXOR*0;
        ANALOGIN(0)             ->    MULTIPLEXOR*9;
        VALVE             :     <-    DIGITALOUT(0)*0,2;
        LAMP              :     <-    DIGITALOUT(0)*3,1;
        DISCHARGE         :     <-    DIGITALOUT(0)*6,1;
        PRESSURESENSOR1   :     ->    ANALOGIN(0)*1;
        PRESSURESENSOR2   :     ->    ANALOGIN(0)*3;
        READY             :     ->    INTERRUPTINPUT*0;
        ALARM             :     ->    INTERRUPTINPUT*2;

    PROBLEM;
            SPECIFY     VALVE DATION OUT BASIC DIM() TFU,
                        (LAMP, DISCHARGE) DATION OUT BASIC,
                        (PRESSURESENSOR1, PRESSURESENSOR2) DATION IN BASIC DIM() TFU,
                        ALARM INTERRUPT;
            DECLARE     (PRESS1, PRESS2, MEANPRESS) FLOAT,
                        OPENVALVE BIT(2) INITIAL('10'B),
                        CLOSEVALVE BIT(2) INITIAL('01'B),
                        (TURNON, TURNOFF) BIT INITIAL('1'B, '0'B);

        START:                  TASK GLOBAL;        /*ACTIVATED FROM OUTSIDE*/
                    .
                    .
                    .
                EVERY 1 SEC ACTIVATE PRESSURECONTROL;
                WHEN ALARM ACTIVATE ALARMING;
                END;    /*OF TASK START*/

        PRESSURECONTROL:        TASK PRIORITY 5;
                TAKE PRESS1 FROM PRESSURESENSOR1;
                TAKE PRESS2 FROM PRESSURESENSOR2;
                MEANPRESS = (PRESS1 + PRESS2)/2
                IF MEANPRESS >= 30
                    THEN SEND OPENVALVE TO VALVE;
                    ELSE    IF MEANPRESS < 20 THEN SEND CLOSEVALVE TO VALVE;
                            FIN;
                FIN;
            END; /*OF TASK PRESSURECONTROL*/

        ALARMING:               TASK PRIORITY 3;
                SEND TURNON TO LAMP;
                SEND TURNON TO DISCHARGE;
                WHILE MEANPRESS >= 20
                    REPEAT;
                    AFTER 2 MIN RESUME;
                END; /*OF LOOP*/
                SEND TURNOFF TO LAMP;
                SEND TURNOFF TO DISCHARGE;
                END; /*OF TASK ALARMING*/
MODEND;
```

ACCEPTANCE.

Real-time languages, other than assembly language, have not always been welcomed with open arms by practitioners in the field. These two comments, one for each of the two "easy programming" languages just discussed, illustrate the point. For Pearl :

> Although the cooperation in the PEARL Workshop was very good on the part of the PEARL implementors, most companies needed another two or three years to complete and market their PEARL implementations. The main reason responsible for this slow progress is probably the fact that the PEARL issue in Germany was from the start not so much a technical issue but rathe an *educational process*, a process which very slowly pulled people away from machine oriented programming to a more advanced way of designing and implementing software for industrial automation. Another retarding effect was caused by the big manufacturers which preferably supply automated equipment as turn key systems where the programming language as a tool does not explicitly appear. The engineering staffs of such firms tend to stick to assembler languages.

For Iliad :

> A language cannot be fully assessed until it has been in use for some time. Indeed, gaining user acceptance is itself an issue. Some practitioners view assembler language as indispensable. Its many flaws seem to be tolerated like the chronic backache caused by a beloved old mattress. However, many others are enthusiastic about using a language like ILIAD once more applications experience has been gained. Similarly, support for design decisions in such application and machine-dependent areas as error processing can only be gained in the field.
>
> ILIAD was developed to bring some of the "theory" of concurrent processing within the reach of noncomputer scientists working in an industrial setting. The skeptical and strongly practical bent of this group had a wholesome effect on the language, tempering the zeal of its designers. The result is a language whose concurrent processing functions are well integrated with its sequential programming capabilities, rather than being tacked on as an afterthought. Experimental evidence and favorable response from users allow the provisional assessment that ILIAD has achieved its design goals, and that it will become a useful program design and implementation tool.

More generally, specialist real-time languages have not caught on in practice. All the languages described in this sheet, as well as several others, had, and in some cases still have, enthusiastic proponents, but it is probably true to say that the greater part of real-time programming today is carried out in Ada (which I haven't described, because I don't know enough about it), or in C or C++. Nevertheless, the specialist languages illustrate rather well how the requirements of real-time control extend the capabilities of traditional languages, and how they can in principle be satisfied.

That is not to say that interest in developing real-time languages has died out. The extract which follows comes from the November, 1995 issue of *Sigplan Notices*, which is a report on the 1995 ACM Sigplan workshop on languages, compilers, and tools for real-time systems. Anyone particularly interested in the topic of real-time languages could usefully read it; it covers a wide variety of subject areas, and contains many useful pointers to other work.

I've chosen this excerpt because it fits in as well as any with the material covered in lectures, and brings out some of the issues rather well. It's just the introductory part of a paper on the nature of "RTsynchronizers"; you might like to consider whether what they're trying to do is an answer to Wirth's proposal that it would be good to encapsulate the essence of real-time computing so that you didn't have to worry about it any more.

REFERENCES.

"Purdue Fortran" recommendations : J. Gertler, J. Sedlak : "Software for process control – a survey", *Automatica* **2**, 613 (1975), reprinted in R.L. Glass : *Real-time software* (Prentice-Hall, 1983). (For a full specification : W. Kneis : "Draft standard, industrial real-time Fortran", *Sigplan Notices* **16#7**, 45 (July 1981).)

RTL/2 : SPL International : *Introduction to RTL/2* (ICI, 1974).

ILIAD : H.A. Schutz : "On the design of a language for programming real-time concurrent processes", *IEEE Transactions on Software Engineering* **5**, 248 (1979)

PEARL : T. Martin : "PEARL at the age of five", *Computers in Industry* **2**, 1 (1981).

RTsynchronizer : S. Ren, G. Agha : "RTsynchronizer : language support for real-time specifications in distributed systems", *Sigplan Notices* **30#11**, 50-59 (November, 1995).

Alan Creak,
April, 1998.