Data Handling Division
DD/86/18
D.O. Williams
December 1986

LANGUAGE REQUIREMENTS FOR EMBEDDED SYSTEMS

(Presented at the ECA Real Time Distributed Software Seminar Brussels,
3 June 1986, and submitted to Computer Standards & Interfaces)

LANGUAGE REQUIREMENTS FOR EMBEDDED SYSTEMS

**David O. Williams
Data Handling Division
CERN
CH-1211 Geneva 23
Switzerland**

Abstract

This paper forms the basis of a talk presented at the ECA Real Time Distributed Software Seminar on 3 June 1986 in Brussels, Belgium. The seminar was organised by the European CAMAC Association (ECA), with the support of the Computer Science Department (FVI) of the University of Amsterdam, and is sponsored by the Commission of the European Communities.

The paper deals with the programming language requirements of embedded computer systems. Some of those requirements do not match well with the approach of conventional programming language design, which has aimed to hide details of the underlying computer hardware from the programmer. Furthermore, in embedded systems it is often necessary for the language to be able to deal with multiple tasks, multiple processors, and all of the related communications issues. Finally the language(s) used must not be treated in isolation, but as one (very important) component of the software engineering process.

CONTENTS

1.	INTRODUCTION	1
1.1	Disclaimer	1
1.2	Embedded systems	1
1.3	Language abstraction	1
1.4	Requirements analysis	2
2.	REQUIREMENTS - SIMILARITIES WITH CONVENTIONAL LANGUAGES	2
3.	REQUIREMENTS - CONTRASTS WITH CONVENTIONAL LANGUAGES	2
3.1	History - hiding the hardware	2
3.2	Storage abstraction	3
3.3	Bit manipulation and input/output	3
3.4	Interrupt handling	3
3.5	Error recovery	4
3.6	History - serial programming	4
3.7	Multi-tasking	4
3.8	Multiple processors and communications	5
4.	SOFTWARE ENGINEERING FOR EMBEDDED SYSTEMS	5
4.1	Analysis and Design	5
4.2	Host/target development	6
4.3	Remote debugging	6
4.4	Libraries and language mixing	6
4.5	Modularity	7
5.	LANGUAGES AND KERNELS	7
6.	COMMENTS ON THE REAL WORLD	8
6.1	Assembly language	8
6.2	C	8
6.3	Kernels	8
6.4	Modula-2	8
6.5	Ada	8
7.	STANDARDISATION	9
7.1	The problem of history	9
7.2	The problem of evolution	9
7.3	The problem of the runtime environment	10
7.4	The problem of program libraries	10
7.5	The future	10
8.	ACKNOWLEDGEMENTS	11
	References	11

1. INTRODUCTION

1.1 Disclaimer

This paper tries to identify, in a very general way, those features that must be present in a programming language in order that it can be considered suitable for programming embedded systems. Experts in the field may consider the approach too simplistic. However, one of my theses is that this field is not receiving the attention that it requires from academic computer scientists, and that there is a worrying gap between the rhetoric of some language proponents and the reality of programming embedded systems. I would like to narrow, or even close, that gap, and I hope that this paper will help to further that aim.

1.2 Embedded systems

Embedded systems normally involve the use of one or more *general-purpose* microprocessors in a *dedicated* environment. The microprocessor does not normally have classic peripherals such as disks, but is equipped with memory and "special" peripherals for acquiring and controlling data. It is not normal to "log-on" to an embedded system, except perhaps for the rather special cases of program debugging and system monitoring. Since serious program development will be either impossible or, at best, very uncomfortable on the embedded system, this development is normally carried out on another, better equipped, "host" computer. The software produced on the host can then be loaded into one or more "target" embedded systems for execution, either through point-to-point links or through a local area network.

1.3 Language abstraction

The history of programming language development has been marked by a reduction in the level of detail that the programmer had to be aware of. Increasingly powerful abstract concepts ("abstractions") were provided in order to hide details of the "specifics" of the computer hardware that was being programmed. Languages with a high degree of abstractive power allow the programmer to think in ways which are useful at the program design stage, rather than emphasising low level details of the computer hardware or program code.

An example of this concept comes from the area of transfer of control. Most computers have an assembly instruction of the form JUMP <alphanumeric label>. Early versions of Fortran, up to and including Fortran-66, had adopted the construct GO TO <numeric label>, which was even slightly less abstractive than assembler, since the power to give a sensible name to the label had been lost. Under pressure from the user community, Fortran-77 adopted the more abstract and powerful IF-THEN-ELSEIF-ELSE-ENDIF structure. Other modern languages carry control abstraction further, with CASE constructs and condition signalling. The CASE construct provides an excellent example of abstraction, because it is left to the compiler to decide whether to convert the programmer's abstract idea either into specific code that uses a jump-table or into code that relies on a series of conditional branches.

Besides control abstraction, expression abstraction (the ability to write $A = 3.5 * (B^{**2} + C)$ rather than specifying the individual operations) and data abstraction (where the programmer can define the characteristics of the data) have also been important in the development of conventional programming languages. Even though there is room for (considerable) improvement, most modern languages offer reasonable facilities in these areas. On the other hand few languages have been developed that provide effective abstraction in the areas which are important for programming embedded systems.

1.4 Requirements analysis

The most consistent approach to the use of requirements analysis in order to specify a language for programming embedded systems was made by the United States Department of Defence in the 1970s. A set of specifications was steadily refined through documents called Strawman, Woodenman, Tinman, Ironman and Steelman [1]. The final result of this process was the definition and implementation of the Ada programming language [2]. I will return to give some comments both on this approach and on Ada itself later.

It is important to realise the limits of requirements analysis. There are many different applications for which embedded systems are used, and they do not all have the same requirements. For example, in scientific research, it is often crucially important to be able to execute complex mathematical algorithms in embedded systems. On the other hand many embedded systems in industrial situations require little or no mathematical calculation. The languages appropriate in these two cases can be very different.

2. REQUIREMENTS - SIMILARITIES WITH CONVENTIONAL LANGUAGES

There are a set of standard requirements for all programming languages that cynics will be inclined to dismiss as "motherhood and apple pie". The cynics are wrong. Rather clear statements of the required features are to be found in Chapter 1 of Steelman and in Section 1.3 of the Ada Reference Manual.

I would like to highlight a few points that I find particularly important for embedded systems.

- The language should be available in efficient implementations on a wide range of machines at a reasonable cost.
- It should be well standardised, and there should be a body watching over the smooth evolution of the language.
- The language should be both as simple as possible and as complex as necessary in order to provide the required functionality. But above all it should be consistent.
- It should be possible for a team of programmers to collaborate on the production of large software systems using the language. This has strong consequences on the facilities for modularisation and separate compilation.
- It should be possible to produce reliable programs in the language. There are two consequences. First, as many programming errors as possible should be detected by the compiler. Second, the programmer must be given tools to manage recovery from hardware and software errors. Some people believe that this second point is only a requirement for languages being used for embedded systems, but I believe that it is necessary for any language that will be used for production programming, as opposed to teaching.
- It should be possible to produce readable, well-documented programs in the language. The detailed syntax of the language is very important in achieving this aim.

3. REQUIREMENTS - CONTRASTS WITH CONVENTIONAL LANGUAGES

3.1 History - hiding the hardware

The early language designers were concerned to introduce enough abstraction into their languages in order that the programmer did not need to worry about the trivial details of the hardware of the computer that would be used to execute the program. It was assumed that providing input data and producing printed output were things that would be done by an operating system behind the program-

mer's back, and that the program itself would need to specify little, if anything, about how that should be done.

This objective was clearly laudable in the context of the time, but meant that early languages were explicitly designed to avoid dealing with most of the real issues facing us in embedded systems.

3.2 Storage abstraction

In order to control special peripherals efficiently, it must be possible to provide a description of the computer memory in "machine" units. These units may be any, or all, of :- bits, groups of bits, bytes, words, long-words and so forth. The programmer must be able to force certain variables or data structures to be stored in well-defined areas of memory.

The requirement is particularly obvious for control and status registers, where a powerful language will allow the different bit-fields to be manipulated independently and symbolically.

When dealing with "systems" programs (interpreters, graphics packages etc.) it is often necessary to be able to constrain rather complex data structures to occupy a well-defined area of storage, and to provide several "overlaid" views of this storage, since the precise layout and location of the data structures will not be known until runtime.

3.3 Bit manipulation and input/output

In embedded systems it is not uncommon to need to manipulate the bits that represent the value stored in a variable. The manipulations typically involve reading, storing, masking, complementing and shifting in various ways.

One point that is also easy to forget is that the engineer developing computer hardware will almost certainly want to have facilities to enter and print bit-oriented data in octal or hexadecimal form. This is a good example of the fact that an irrelevant detail to one person is a welcome abstraction to someone else.

3.4 Interrupt handling

Almost all modern processor architectures contain an interrupt system. Whenever an important event occurs the processor is notified by an interrupt, which causes the status of the processor to be saved (with various degrees of hardware assistance, depending on the processor architecture), and which then invokes a new thread of control, frequently via a forced transfer through a fixed memory address. When the processing of the interrupt is complete the thread of control that was interrupted is resumed.

For a long time the programming of the interrupt system was normally carried out in assembler, and was something of a black art. Any mistakes tended to have catastrophic results, frequently blocking the interrupt system and forcing the computer to be restarted. I believe that it is important to find the correct abstraction for programming the interrupt system. From the point of view of the systems designer it can be regarded as a mechanism by which tasks are started up automatically in response to an asynchronous external event. This viewpoint permits the designer to deal with interrupt system as part of the more general inter-task synchronisation problem.

3.5 Error recovery

In general it has not been important for "conventional" programs to be very robust. The occurrence of an error, such as "floating point overflow" or "invalid memory reference", was normally an indication that the algorithm had been programmed incorrectly, and program execution could be terminated immediately, so that the programmer could take corrective action.

It is in the nature of most embedded systems that they are left to run for many hours or days once they are started. Indeed there are a class of embedded systems involved in the control of continuous process plants (oil refineries, power plants, etc.) which can only be stopped with major preparations.

Accordingly programs for embedded systems must keep running despite the occurrence of the vast majority of hardware and software malfunctions. The language used must contain an abstraction that provides information about the nature and locality of errors and allows the programmer to foresee appropriate recovery in the different cases.

3.6 History - serial programming

The early languages implicitly assumed that they would be used to generate programs that would execute serially. These programs typically would read some input data, invoke an algorithm to manipulate the data, and output the result(s) of that algorithm. Instead of being driven in this "top-down" way, the normal program in an embedded system is driven "bottom-up", since it needs to react correctly to the arrival of many different stimuli (clock ticks, data, alarms etc.).

Especially during the last decade attempts have been made to design languages that have the correct abstractive power to deal with the event-driven nature of embedded systems. In addition to coping with single processor environments, it is becoming increasingly obvious that a really successful language for embedded systems must be able to deal with the operation of systems that involve many processors, connected both through shared memory and through communications links.

3.7 Multi-tasking

In an embedded system the occurrence of any "event", such as an interrupt, or the arrival of an alarm, or the arrival of normal input data for processing, is the logical signal that a thread of control should be initiated. Exactly how the event "occurs" will depend on details of both the processor architecture and the device involved. Nevertheless each incoming event constitutes a request to the processor for some code to be executed.

In the most trivial case the code may be a few lines of assembler that record the event for later processing. Even if this code runs as an interrupt routine we should not lose sight of the fact that it is essentially a new thread of control. If the code is more substantial it will commonly be called a task.

It is not unusual to find that some events require a much faster response than others. A common way of dealing with this is to construct the software as several quasi-concurrent tasks. A priority is normally assigned to each task, and execution of a given task can be interrupted by another task that has a higher priority. When the interrupting task has completed the original task is resumed.

Modular program design teaches us that we should separate out the issue of queuing incoming events (de-randomising them) from the execution of the associated algorithms. A good way to do this is through input queues. Interrupt routines, or special tasks, create entries in these queues, and a task with a given functionality then executes whenever its input queue is non-empty. It may be necessary to use a similar mechanism to deal with output, since the tasks may need access to shared resources, such as communications links or special hardware.

Issues that must be addressed by the designer of multi-tasking software include:-

- The visibility of data from tasks, and the way in which write-access to this data is controlled.
- The way in which task execution is started, suspended, restarted and stopped.
- The way in which data can be passed between tasks.
- The way in which the whole system is modularised.

Real-time kernels have been used extensively to implement multi-tasking embedded systems, with the individual tasks being programmed in conventional languages. More recently languages have been developed (notably Modula-2 [3] [4] and Ada) with features aimed at dealing directly with the abstraction of multi-tasking.

3.8 Multiple processors and communications

While there are few languages that have enough abstractive power to deal with the programming of multi-tasking systems in an adequate way, there are even fewer that can cope with software that has to run as one system on several different processors, typically attached through communications links. The low bandwidth that is frequently available on these links all but excludes the use of variables that are shared between tasks running on different processors. Whether generality then forces all shared variables to be abandoned, even when they are being shared by tasks on a single processor, is an important architectural decision.

Another decision that has to be taken by the language designer deals with the naming and access methods that are available to the programmer for communication between machines. Does the name of a task in a remote machine have to be present in the code on the local machine, or can this information be parametrised in some way?

It is natural to expect that there will be some graceful degradation in a multiple processor system. If one or two processors are out of commission in a system composed of one hundred machines, we would not expect the whole system to be out of action, but rather that a few functions might be unavailable. This requirement has important consequences for the way in which information about the configuration of a multiple processor system is entered into the system. The implicit assumption of Ada and Modula-2 seems to be that the configuration is known at the time that the program is compiled. Any change in the configuration, including the addition or modification of a new task, therefore requires a recompilation and rebuild of the system.

The people who appear to have taken the correct approach, at least in this area, are the team behind the Conic Toolkit [5]. Each task can be separately compiled, and refers locally to the names of exit and entry ports, through which two types of messages can arrive. The connection between the entry port of one task and the exit port of another, is made by a configuration manager.

4. SOFTWARE ENGINEERING FOR EMBEDDED SYSTEMS

4.1 Analysis and Design

The idea that software is an engineering discipline, and that correct analysis and design are as much part of software as writing the code, is not new. What is new is that reasonable tools are becoming available with good graphics, and at affordable prices. There is now some quite extensive experience with the use of SA/SD techniques in several applications at CERN.

The analysis and design of real-time systems is normally considered to be much harder than the design of serial systems. An extra variable, namely time, has been introduced into the problem, and we nor-

mally have to consider many different possible threads of control, with their multiple possible interactions.

Mascot [6] is a design approach that has been extensively used in military embedded systems in Britain since about 1975. It is the only approach that I know of that tries to tackle the full problem, but I have little personal knowledge about its effectiveness. This may be partially because it was originally rather tied to the use of the Coral-66 language. I have also heard of American companies that are trying to modify SA/SD to deal with the full complexity of real-time systems, but the vast majority of existing analysis and design tools are not really satisfactory in this field.

When we have a reasonable model for multi-task and multiple processor execution, then we will have made enormous progress in the implementation of reliable embedded systems. Such a model will have immediate influence on the analysis and design process, and on the language constructs themselves.

4.2 Host/target development

It is normal to find that the "target" processor in an embedded system is rather primitive. Even though improved technology is continually driving the costs of these systems down, and optimists might hope that they will eventually become less "hostile" for the programmer, the pessimists will point out that there are economic reasons why this is unlikely to change quickly, since there are often tens or hundreds of these processors in real embedded systems.

The most important points to consider are the fact that the target processor may be very far away from the host ("down the corridor" is very common, and at CERN a distance of 5 to 10 kilometres is not unusual); the fact that the memory size will probably have been optimised for the production environment; and the likely absence of a mass storage device. Debugging may not be easy. One approach is to move physically to the target processor and to try to debug it directly. This will probably require, as a minimum, the temporary installation of a directly-connected terminal and extra memory.

4.3 Remote debugging

Even if the software environment is well enough developed to allow the programmer to sit on a host computer and load and debug a remote target processor, the requirements of embedded systems are still considerable. Any self-respecting debug system will naturally work symbolically, which may already pose problems if there is limited memory on the target. It is much better to keep all symbol tables on the host and to arrange for the host to carry out all of the necessary conversions. In addition the "breakpointing" techniques that are often appropriate to debugging serial programs risk to pose severe problems in a multitasking and multiprocessing environment. Trace techniques, where the time and nature of all important events are recorded, can be very valuable, and debuggers that allow to work in a "break on event" mode may be useful. I believe that insufficient brainware has been devoted to this issue, which is crucial to the speed with which much embedded software is implemented. It is important to remember that it is often hard to distinguish between intermittent hardware and software errors as embedded systems move into a production environment.

4.4 Libraries and language mixing

In the physics environment we can see the requirement to install "huge" programs on embedded systems. By "huge" we mean programs whose size ranges from tens of thousands of lines up to about 100,000 lines. These programs are the result of many years of work, and frequently are based on pre-existing libraries, which can represent an enormous investment, but which are not necessarily implemented in a "modern" programming language.

In this, and in many similar cases, one is driven to the conclusion that the ability to mix two or more languages on an embedded system is a serious need. An immediate consequence is the need for the existence of a common procedure calling and condition handling standard, which must be supported by all of the compilers that are to be used, and which should be understood by the debugging tools. There are many microprocessors, and incidentally also some extremely popular mainframes, for which such a standard does not exist.

4.5 Modularity

One of the reasons why Fortran obstinately refused to go away as Algol-60, Algol-68 and Pascal were all introduced was that it offered modularity and separate compilation at the level of the subroutine. This modularity was not perfect, since there was no check that the caller and callee had defined their interfaces compatibly, but it did allow large program suites to be developed independently by different programmers, which was much harder to achieve with the more "modern" languages.

More recently both Modula-2, with the module, and Ada, with the package, have both used essentially the same technique of separating a program unit into independent specification and implementation parts, so that units can be compiled independently while still being sure that the interfaces are compatible.

However, for embedded systems it seems obvious to me that we should concentrate on the task as (at least one) unit of design and compilation. Tasks implement algorithms, which may be more or less complex, they are started by the occurrence of one or more input events, they have access to various areas of memory that they share with other tasks on a read-only or (protected) read-write basis, and they generate various events as output. This is surely where we must concentrate our energies to achieve good abstractions, so that the programmer can be offered analysis and design tools, and eventually language constructs.

5. LANGUAGES AND KERNELS

People often try to make a religious war out of the issue of whether we need real-time kernels when we have real-time languages. My own position is relatively clear, and has been explained at length in [7]. When we have one or more good real-time languages then it will be possible to envisage doing away with real-time kernels. I do not think that we are at that point yet, despite the claims of Modula-2 and Ada.

I am waiting to see reports of people who have really used Ada tasking to implement production embedded systems before I am willing to moderate my scepticism about the general applicability of the Ada rendez-vous. In this context Modula-2 is more interesting, since people have used the language to implement real-time kernels, which is an interesting proof of its power. Indeed scientists with an experimental background might hope that work to define several different kernel paradigms might be the best way to move forward towards a correct tasking abstraction in a programming language.

On the other hand it is important to remember that when a adequate real-time language is available a major increase in programmer productivity will be possible through synergy. Programmers will understand each other's work more quickly, and the establishment of libraries of commonly required tasks for incorporation into new embedded systems will become feasible, thereby simplifying the design process and reducing the implementation time.

6. COMMENTS ON THE REAL WORLD

6.1 *Assembly language*

Many embedded systems in the real world are programmed in assembly language and use the interrupt system to give primitive multi-tasking. Especially while the size of the overall code remains reasonable and the requirements for subsequent modification remain small, this approach is not so easy to criticise as some people imagine. It certainly can be used to generate software that is both fast and compact, which is not always the case today for the other approaches.

6.2 *C*

For people who are afraid, for whatever reason, of languages with too much abstractive power, but who want to obtain more transportability than is available with assembler, C provides a reasonable compromise. Although it can be used to control the interrupt system in the same way as assembler it has no abstractions for dealing with more general multi-tasking, so it is frequently used in conjunction with real-time kernels.

6.3 *Kernels*

Developers of embedded systems are often constrained to port software and libraries that have been written in a conventional language on classical computers. Whether for this reason, or for any other, as soon as the choice of language has fallen on conventional products such as Pascal and Fortran, the implementor of any serious embedded system is forced to use a real-time kernel. Despite the perceived lack of elegance, many useful systems have been constructed in this way, including, I suspect, some that were programmed in (serial) Ada.

6.4 *Modula-2*

Modula-2 is a poor scientific programming language, missing in particular any control over extended precision and complex arithmetic, and having inadequate support for arrays of more than one dimension. In addition it has no error recovery, which as I indicated above, is desirable for conventional programming and is generally considered essential for the reliable programming of embedded systems.

With these reservations, which may have more or less importance in any given embedded application, it is a fine language, although the level of industrial support is disappointing at present. Its low-level tasking primitive is a good tool for the exploration of a number of tasking models.

6.5 *Ada*

Ada is an enigma, for several reasons. First of all it is a good scientific programming language, probably the best. My only reservations on this score come from the potential for user confusion that will be caused if several incompatible packages are developed in order to provide support for mathematical features such as complex arithmetic and array handling. This is, of course, not so much a feature of the language design, which is very flexible, but rather of how the community decides to exploit this flexibility.

In addition, it has features for dealing with error recovery that appear to be completely adequate.

But it has a very rigid model of tasking that I personally find very hard to believe provides the correct approach. There are a number of papers that have explored this issue in great detail, including [8] and

[9]. The most complete summary that I know about is to be found in an excellent long paper by Burns, Lister and Wellings [10]. Incidentally, the history of the requirements analysis approach to defining the functionality required for multi-tasking must be worth investigating in more detail. Steelman contains a whole chapter on Parallel Processing, which contains, for example, in Section 9I, the specification:- "It shall be possible to set a signal (without waiting)". As far as I can see this requirement is not met in Ada, which should tell us something, but I am not sure quite what.

My final comment would be that I am very concerned at the lack of rigour with which Ada is discussed by its supporters. I would be one of the first to proclaim that it represents a very great achievement, and is probably the best serial programming language available today. However, there is so much political and industrial prestige behind Ada that it does seem that we are sometimes blind to its shortcomings. Is it really a good language for programming embedded systems? Are people really using the tasking constructs for production software? Without assembly language additions? Only time will provide an answer.

7. STANDARDISATION

Language standardisation is much like the conduct of national or international economic policy - it affects a lot of people, and they all think that they could do the job better than the bunch who are creating the present problems! I suspect that both the management of economic policy and the standardisation of programming languages are much harder than even the experts are willing to admit.

7.1 *The problem of history*

There are at least five major programming language cultures in the real programmer's world today, namely Basic, C, Cobol, Fortran and Pascal. For each culture there are a large number of programmers who are more or less fluent in the language, but not many people are really conversant with more than one language. None of these language cultures are very well adapted to dealing with the requirements of embedded systems.

If you want to develop a language that copes well with the problems of embedded systems, then you have the choice between selecting one of the existing main languages and extending it for with new features, or of defining a completely new language. The trouble with the first choice is that the resulting language will not seem "comfortable" for programmers from the other language cultures. The trouble with the second choice is that it may not seem "comfortable" to anyone at all. Modula-2 and Conic have adopted the first approach, basing themselves (more or less) on Pascal, whereas Ada is essentially a complete redesign.

7.2 *The problem of evolution*

The need to cope with embedded systems has highlighted the fact that programming languages, like natural languages, must be capable of adapting to new requirements. It is not always easy to admit that one's favourite language, which had just been completed after five years of hard work, is less than perfect and should be updated, and even when that has been done, it is not easy to persuade the conservatives on the standardisation bodies that the language is not cast in stone.

7.3 The problem of the runtime environment

Conventional serial programming languages had rather simple requirements for interfacing to the computer on which they ran. These typically comprised input and output to slow peripherals and mass storage, and some error trapping. A rather small run-time library sufficed to interface between the user's requests and the underlying operating system.

For embedded systems, especially for those which run on multiple processors, the requirements are more extensive. Interfaces are needed to operating systems, which may be of radically different conception, including "full" systems, real-time kernels and executives, and to communications software for the dialogue with the remote machines over various different types of link. Besides the problem of providing and supporting all of these interfaces, there is a potentially more serious problem for portability. In the extreme the interface presented to the language by the operating system or by the communications software can prove to be incompatible with the sort of interface specifications required by the language itself.

7.4 The problem of program libraries

I believe that language designers frequently underestimate the impact and importance of program libraries. In many real-world situations we find that the programmer is programming against the specification of library routines rather than against the specification of the language. And, despite some preconceived ideas, for many applications of embedded systems the libraries are at least as important as in classical systems. After all, the reason for using dedicated embedded hardware is essentially economic - the resulting system is cheap. If all of the benefit is thrown away because the software requires years of investment in program libraries to be thrown out of the window, then there is food for thought.

The lifetime of many scientific programs is from 10 to 20 years. Ada development has taken about 10 years from first ideas to first commercial compilers, and the validity of one of the pillars of the language, the tasking model, is still unsure. The cycle time to revise the standard definition of a major language appears to be about 10 years as well. I find progress in these areas too slow. One consequence of this reality is that language designers should make sure that new languages can immediately use old libraries, and, moreover, that it is possible to maintain libraries that can be used by a mixture of old and new languages for a period of many years.

7.5 The future

Despite, or perhaps because, of my involvement in a number of standardisation activities, I am rather less optimistic than it is fashionable to be about the future progress of standardisation of languages for embedded systems.

I do not think that standards can be engraved in stone, since the problems that we are trying to solve, and the techniques that are available for us to use, keep evolving. The conservative elements on standards committees, however, would prefer that, once written, a standard should never ever change. And that, at the very least, no standard-conforming program should ever be invalidated by a future change in the standard, even though the modification required might be trivial.

If I could dream, I would like one or more of the major modern languages to evolve so that it became adequate for the programming both of scientific data acquisition and of control algorithms on multi-tasking, multi-processing embedded systems. To meet this goal, I think that Ada would require improved multi-task and multi-processor abstraction, and some clearer standardisation of scientific packages. Modula-2 would certainly need more work on multi-processing, error recovery features, and a complete rethink of the scientific programming capabilities. And Fortran-8X, which risks to become the best scientific programming language available, appears not to be interested to become involved with multi-tasking and multi-processing at all.

Standardising on the wrong solution might be even more expensive than not standardising on any solution at all. But if we do not standardise on anything then we ensure that we will waste enormous amounts of time and effort with respect to the ideal, correct, standardisation. We should all try to be honest, to listen to our colleagues using other languages, and to cooperate!

8. ACKNOWLEDGEMENTS

I have taken part in many formal and informal discussions with colleagues at CERN and elsewhere on the subject of programming languages, and it is a pleasure to acknowledge this help.

REFERENCES

- [1] U.S. Department of Defence. Steelman Requirements for High Order Computer Programming Languages. June 1978.
- [2] U.S. Department of Defence. Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815 A. January 1983.
- [3] N.E. Wirth. Programming in Modula-2. Springer-Verlag 1983. ISBN 0-387-12206-0.
- [4] N.E. Wirth. Schemes for Multiprogramming and their Implementation in Modula-2. Revisions and Amendments to Modula-2. ETH Zürich Informatics Institute June 1984.
- [5] M. Sloman, J. Kramer and J. Magee. The Conic Toolkit for Building Distributed Systems. May 1985. Proceedings of the 6th IFAC Distributed Computer Control System Workshop, Monterey, CA. To be published by Pergamon Press.
- [6] A.P. Hill and J.K. Slape. A MASCOT Approach using Ada. Pages 241-249 in Proceedings of the Third Joint Ada Europe/AdaTEC Conference, Brussels 26-28 June 1984. Cambridge University Press 1984. ISBN 0-521-30102-5.
- [7] D.O. Williams. Software and Languages for Microprocessors. November 1985. CERN Data Handling Division DD/85/27. To appear in Computer Physics Communications.
- [8] A.J. Wellings et al. A Problem with Ada and Resource Allocation. October 1983. ACM Ada Letters III-4, pp.112-124.
- [9] P.H. Welch. Structured Tasking in Ada? July-August 1985. ACM Ada Letters V.1, pp.17-31.
- [10] A. Burns, A.M. Lister and A.J. Wellings. A Review of Ada Tasking. University of York Department of Computer Science. YCS.78(1985).