CERN-Data Handling Division
DD/85/27
D.O. Williams
November 1985

SOFTWARE AND LANGUAGES FOR MICROPROCESSORS

(Presented at the 6th Summer School on Computing Techniques in Physics,
Nové Město na Moravě, Czechoslovakia, 17-26 September 1985)

SOFTWARE AND LANGUAGES FOR MICROPROCESSORS

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

Abstract

This paper forms the basis for lectures given at the 6th Summer School on Computing Techniques in Physics, organised by the Computational Physics group of the European Physics Society, and held at the Hotel Ski, Nové Město na Moravě, Czechoslovakia, on 17-26 September 1985.

Various types of microprocessor applications are discussed and the main emphasis of the paper is devoted to 'embedded' systems, where the software development is not carried out on the target microprocessor. Some information is provided on the general characteristics of microprocessor hardware. Various types of microprocessor operating system are compared and contrasted. The selection of appropriate languages and software environments for use with microprocessors is discussed. Mechanisms for interworking between different languages, including reasonable error handling, are treated. The CERN developed cross-software suite for the Motorola 68000 family is described. Some remarks are made concerning program tools applicable to microprocessors.

PILS, a Portable Interactive Language System, which can be interpreted or compiled for a range of microprocessors, is described in some detail, and the implementation techniques are discussed.

CONTENTS

1.	INTRODUCTION AND SCOPE	1
1.1	Microcontrollers	1
1.2	'Embedded' systems	1
1.3	Microcomputer systems	2
2.	MICROPROCESSOR HARDWARE - SOME BACKGROUND	2
2.1	Processors	2
2.2	Memories	3
2.3	Serial I/O	4
2.4	Mass storage	4
2.4.1	Magnetic disks	4
2.4.2	Network links	5
2.4.3	Magnetic tape	5
2.5	Special I/O & Bus systems	5
2.6	Transputer	6
3.	MICROPROCESSOR SOFTWARE - AN OUTLINE OF THE PROBLEM	6
4.	ROM-BASED MONITORS	7
4.1	Functionality	7
4.2	Input/Output	7
4.3	Debugging	8
4.4	Multi-task/multi-processor aspects	8
5.	SYSTEMS SOFTWARE	8
5.1	Hardware and Software Context	8
5.2	Software for 'naked' microprocessors	9
5.2.1	Polling	9
5.2.2	Status- or time-driven applications	9
5.2.3	Modula-2	10
5.2.4	Interpreters	10
5.2.5	Pure interrupt code	10
5.3	Real-time kernels	11
5.4	Highly efficient kernels	13
5.5	Single user operating systems	13
5.6	Full operating systems	14
6.	LANGUAGE SELECTION	15
6.1	Community requirements for adopting a language	15
6.2	Language Selection for Embedded Microprocessors	16
6.3	Mainline compiled languages	16
6.3.1	Control abstraction	16
6.3.2	Data abstraction.	17
6.3.2.1	Arrays	17
6.3.2.2	Bits	17
6.3.2.3	Enumerated types	18
6.3.2.4	Hiding of data structures	18
6.3.2.5	Real numbers, extended precision and complex arithmetic	18
6.3.2.6	Records	19
6.3.3	Debugging	19
6.3.4	Error recovery	19
6.3.5	Expression abstraction	20

6.3.6	Interrupt handling	20
6.3.7	Libraries and packages	20
6.3.8	Readability	20
6.3.9	Separate compilation	20
6.3.10	Standardisation	21
6.3.11	Storage abstraction	21
6.3.12	Task abstraction	22
6.3.13	Terminal dialogues	22
6.3.14	Transportable I/O	23
6.4	Other languages	23
6.4.1	Forth	23
6.4.2	Concurrent Pascal	24
6.4.3	Occam	24
6.5	Interworking and libraries	25
6.5.1	Runtime libraries	25
6.5.1.1	Standard I/O	25
6.5.1.2	User libraries	26
6.5.1.3	Mathematical libraries	26
6.5.1.4	Conventions	26
7.	CERN CROSS SOFTWARE ENVIRONMENT	26
8.	SOFTWARE TOOLS FOR MICROPROCESSORS	27
8.1	Language tools	27
8.2	Design methodologies to support multi-tasking	27
8.3	Debugging tools	28
9.	SUMMARY	28
9.1	Language selection	28
9.2	Integration	29
9.3	Multi-tasking	29
9.4	Multi-processing	29
10.	PILS LANGUAGE DESIGN	29
10.1	The drawbacks of BASIC	30
10.2	Names	30
10.3	Types and typing	30
10.4	Based numbers	30
10.5	Arrays	31
10.6	Subroutine structure	31
10.7	Library routines	31
10.8	MODULE structure	31
10.9	Control structures	31
10.10	File operations	31
10.11	Events and error recovery	31
11.	PILS COMMAND INTERFACE	32
11.1	General commands	32
11.2	Editor commands	32
11.3	File manipulation commands	32
11.4	Commands to control execution and debugging	33
12.	PILS IMPLEMENTATION	33
12.1	Choice of the implementation language	33
12.2	Transportable source code management	33

12.3	PILS/Host Interface	33
12.4	Ports	33
12.5	The checker	34
12.6	The emulator	34
12.7	The compiler	34
13.	BIBLIOGRAPHY	35
14.	ACKNOWLEDGEMENTS	35
	References	35

1. INTRODUCTION AND SCOPE

In this paper I do not intend to consider software or languages for microprocessors that are being used either as microcontrollers or as components of general-purpose microcomputer systems.

1.1 Microcontrollers

A microcontroller is an assembly of electronics that controls a given piece of equipment. In response to certain well-defined input signals it arranges to change the state of the equipment according to certain predetermined rules. In the old days (say roughly pre-1970) these controllers were nearly always designed from scratch using discrete electronic components. During the 1970s 4-bit and 8-bit microprocessors were sometimes used to provide this functionality. At present, special purpose devices such as the AMD 2910 Microprogram Controller or the Signetics 8X300 Microcontroller, or Programmable Array Logic chips (PALs), are nearly always preferred to general purpose microprocessors in this field.

Note, however, that there are few rigid dividing lines between processors, whether small or large, specialised or general purpose, simple or complex. Special purpose devices are applicable and economic where the functions that are needed are really well-defined and unlikely to change for many months (or for ever) once the 'program' has been debugged. If the functions required either will be in a state of continuous flux, or if they are complex to describe (for example, if they depend heavily on time-dependent data) then it may well be more appropriate to use a microprocessor nevertheless.

Interesting new software products such as ABEL [1], which permit the programming of PALs from state machine descriptions, are now becoming available for the programming of microcontroller applications.

1.2 'Embedded' systems

Having excluded some 'low-end' applications of microprocessors I would now like to describe the sort of applications that I do want to talk about before going on to exclude some 'high-end' applications.

The 'classic' microprocessor application in particle physics involves the following actions : —

- Receiving a combination of data and conditions as input
- Arranging to execute one or more algorithms to process this input
- Generating, as a function of the results of the algorithms, a combination of (modified) data and conditions as output
- Trying to ensure that the whole process is working correctly. This involves providing code to handle the inevitable hardware and software errors, and the supervision of all programs to ensure that they are producing sensible results.
- Informing the 'operator', who might be human or consist of programs in the same or a different computer, about the status of the system

We note the following : —

1. At certain points in time the application is declared to be 'ready' and it is normally left to run 24 hours per day without any modification for weeks or months on end.
2. The software can be very complex (tens of thousands of lines of code) and might need to be changed for any of the following reasons : —
 - The configuration of the microprocessor hardware on which it must run is altered.

- The configuration of the physics experiment changes, for example an additional detector might be added.
- The physicist or detector designer decides that a different algorithm will give better results.

An important point about this 'classic' use of microprocessors in particle physics is that the software that controls the hardware is not developed in the microprocessor(s), but rather on some other computer system that provides a more comfortable software environment. Two different names are sometimes used to describe this. 'Embedded systems' is the name used by the American Department of Defence in all of the Ada development. It refers to the fact that the systems are 'embedded' and become an integral part of a larger piece of equipment (in our context, part of a radio-frequency accelerator system, part of a fire-detection system, or part of a detector read-out system). 'Target systems' or 'Host/target systems' are names which are sometimes used to emphasise the fact that the software development takes place on a 'host' system which is different from the 'target' where the application will run.

1.3 Microcomputer systems

Since the processing unit of a microprocessor is frequently designed as a general purpose object, it has been very common for designers of personal workstations or general purpose timesharing systems to base their products on these microprocessors. For example, Apollo Computers, Masscomp and Sun Microsystems all have products [2], [3], [4], based on the Motorola 68000 family of microprocessors [5].

I consider that these products are outside the scope of this paper because they are intended as general purpose computer systems and are not typically used for dedicated 'embedded' systems. The software and languages that are available on these machines should be compared and contrasted with that available on more traditional computers with their specially designed CPUs.

To emphasise the point, I am going to consider the needs of users of 'embedded' or 'target' systems rather than users of powerful general purpose computers that happen to be based on microprocessors.

2. MICROPROCESSOR HARDWARE - SOME BACKGROUND

For the sake of completeness I will discuss something about the typical hardware configurations that we meet in modern microprocessor systems. Most of my practical examples are taken from the Motorola product range. This is because it is the product range with which I am the most familiar. This emphasis is not intended as an endorsement of Motorola products - many other manufacturers make very interesting microprocessor products.

2.1 Processors

Brief histories of the development of processor units can be found in [6] and [7]. 4-bit processors are little used in CERN today, and the use of 8-bit processors, such as the Motorola 6809, is most common in applications that are relatively simple and/or very price sensitive. 16-bit processors such as the Motorola 68000 or 68010 provide a general instruction set oriented to the efficient manipulation of 8-bit bytes, 16-bit words and 32-bit long words, and to performing integer arithmetic. The instruction set has some limitations, for example, when dealing with division of two 32-bit quantities. The normal performance rating of these CPUs is about 0.1 to 0.25 million instructions per second (Mips) for integer work (a Vax/11-750 rates at about 0.5 and a Vax/11-780 at about 0.8 and an IBM 370/168 at about 3.0 on the same scale - all numbers to be taken with the normal large error bars of benchmark tests).

Compared to the 68000 processor the 68010 is much better adapted to work in a virtual memory environment. The key point is that it is possible to restart correctly an instruction that had to be aborted owing to the fact that the required information was not present in physical memory.

The recently introduced Motorola 68020 is a full 32-bit processor, in the sense that all of the limitations of 16-bit integer division are removed and the external data buses are 32-bits wide. The improvement in semiconductor technology that has taken place in the five years between the introductions of the 68000 and 68020 has also permitted a significant increase in raw speed (16 MHz clock as against 8 MHz in the original 68000) and the introduction of some more addressing modes, and some extra features, including a barrel shifter. Taken together these factors seem to increase the power of a 68020 to about 0.5 to 0.8Mips.

Whereas the 68000 and 68010 had no instructions for floating point arithmetic, which was a rather important drawback in many particle physics applications, the 68020 will soon be able to operate as a floating point engine. The mechanism for this will be the 68881 floating point co-processor chip, which was sampled by Motorola in early 1985. When the 68020 finds a floating point instruction it will (simplifying somewhat) suspend its own activity, pass the calculation to the 68881 for execution, and resume when the results are ready. There are, in fact, several reasons for adopting what looks like a rather complicated strategy : -

- The 68881 is not required if the user is sure that (s)he will not need to carry out floating point calculations. Although this may be rather rare in particle physics applications it will be rather common in some other fields.
- It allowed Motorola to design the chips (more or less) independently of each other. This is an example of the principle of modularity, which is at the heart of good software design, applied to hardware design. However, the clinching argument was probably that, given the technology of the time, it would not have been possible to incorporate the features of both parts on a single chip.
- The technique of co-processors is rather powerful and allows for co-processors other than the 68881 floating point chip to be designed. In particular Motorola has announced a 68851 paged memory map unit (PMMU) co-processor which will work together with the 68020 to provide virtual memory facilities.

The present day prices for small volumes of the 68000 processor are about 25 SFR, whereas the 68020 processor is only available today in very low volume, and sells for about 1000 SFR.

2.2 Memories

It is common to find at least two different sorts of memory associated with microprocessors. To avoid lengthy reloading procedures, the software that will control the hardware is often loaded or 'burned' into Programmable Read-Only Memory (PROM). In pure (fusible link) PROMs it is not possible to make any corrections, but most equipment used at CERN for program storage is more likely to make use of UV-EPROMs, which can be erased by exposure of the memory chips to an ultra-violet light source and then re-programmed. A more recent development is the EEPROM which can be erased electronically and then re-programmed in situ.

The 27256 series of EPROMs provides 32 Kbytes of read-only memory on a chip, with a typical access time of 250 nsecs. The access time is often a critical parameter of the memory because if the program or data is just too slow arriving at the processor then the processor has to wait for another timing cycle (60 nsec at 16.6 Mhz, 100 nsec at 10 Mhz and 125 nsec at 8 Mhz) before continuing. Consistently introducing these 'wait states' can severely reduce the inherent performance of the processor. It is normal to refer to the memory/processor combination as having 0, 1 or more 'wait states'.

Since PROMs are read-only they cannot be used to store the variables of a computation, nor can they provide stack or heap space for language or operating system software. This has to be provided by Random Access Memory (RAM). There are two major kinds, Dynamic RAM (DRAM) and Static RAM (SRAM). In DRAM the charge that represents the memory contents leaks away over a fairly short period of time (some milliseconds) and the designer is responsible for reading the contents of each memory cell and rewriting (refreshing) it before it has leaked away. Various more or less complicated methods have to be adopted to do this without interfering too severely with the continuing requirements for asynchronous access to random memory cells.

SRAM chips are flip-flops which are able to retain their contents indefinitely while under power. Some systems are designed using long-life batteries and SRAM to hold the 'state' of a microprocessor system or of the hardware that it is controlling during shutdowns of many days or weeks. Remember that EPROMs can only hold the 'state' that existed at the moment that they were programmed or 'burned'.

At the time of writing 16 Kbit SRAM chips can be purchased in small volume for about 10 SFR each, while the 64 Kbit (8 Kbyte) chips sell for about 30 SFR each. 32 Kbyte DRAM chips cost about 30 SFR each. 16 Kbyte UV-EPROMs sell for about 25 SFR and the 32 Kbyte version sells for about 35 SFR. For all of these chips we are talking about medium-speed access times (200 to 300 nsec.).

2.3 Serial I/O

The use of the V24 (RS232-C) serial interface standard for terminals is so common that it requires an effort to realise that it is not a universal constant. In the fairly short history of microprocessors it has been very frequent to attach a terminal directly to the processor for debugging and control. In fact it is not unusual to have three serial connections, one for the local terminal, one for down-line loading from a 'host' computer and one for attaching a cheap printer. These connections typically work at speeds in the range 4800 bps to 19200 bps.

2.4 Mass storage

2.4.1 Magnetic disks

The subject of mass storage for microprocessors is a source of some confusion. Floppy disk drives were frequently attached to early machines, especially those based on the 8-bit processors. These devices typically store 256, 512 or 1024 Kbytes of data on both sides of a flexible magnetic disk. Access times of about 100 msecs and 'head on track' transfer rates of 60 Kbytes/sec. are typical. This sort of performance permitted the development of single user operating systems based on the microprocessor and floppy.

More recent developments have produced Winchester disk drives that match well with 16- and 32-bit microprocessors, having capacities of tens of Megabytes, access times of about 20 msecs. and 'head on track' transfer rates of many hundreds of Kbytes/sec. This sort of performance permits the development of full multi-user, multi-tasking operating systems based on the microprocessor and Winchester disk.

It seems to me, however, that this type of mass storage has more to do with the 'host' systems than with the 'target' or 'embedded' systems that we are discussing. The systems that we need to build for physics will certainly not have one disk per processor. The 'target' systems will typically not have any disks, but might well have multiple processors. So my conclusion is that the interesting field of disk technology should be left to the experts and we should concentrate on diskless 'target' systems.

2.4.2 Network links

Network links are not normally regarded as 'mass storage' items but in the world of the 'embedded' system they can well play such a role. It is the 'host' computer, with its comfortable software environment, that is much better adapted to dealing with mass storage equipment, whether disk or tape. The reasons for this can include environmental factors (the 'host' can be put in an air-conditioned, dust-free room, if necessary), service factors (diagnostic programs can be provided and service engineers trained for the 'host' system rather than requiring the users staff to become expert in diagnosis on the 'target' system) and operating system software factors. Accordingly, if fast network links are available between 'target' and 'host' then we can discuss where the mass storage facilities should be positioned.

Most people's definition of 'fast' excludes conventional serial I/O facilities running at up to 19200 bps. Note however that RS422 lines running at up to 1 Mbps are becoming more widely available, notably as one of the few standard connections to the Apple Macintosh [8]. Local Area Networks (LANs) are the real hope for the future, and Ethernet, and the related Cheapernet, is just starting to come into general use with microprocessors at CERN. Depending on the network configuration, the software protocols adopted, and the effort devoted to optimisation, real transfer rates in the range 20 Kbytes/sec. to 200 Kbytes/sec. can be achieved. Though not reaching the transfer rate of a 6250 bpi magnetic tape operating at full speed, these figures are respectable and should cause us to think carefully before installing mass storage facilities on 'target' microprocessors

2.4.3 Magnetic tape

The argument given in the previous paragraph depends on the availability of a network connection between the 'target' and the 'host', and this requires that an appropriate cable is in place. In the fast moving world of the physicist carrying out tests on equipment, this may not always be easy to arrange, and it may be necessary to record the data for later analysis. During the 1970s half-inch magnetic tape units attached to mini-computers such as the Norsk Data ND-100 or the Digital Equipment PDP-11 were used for this job. Attaching such units to microprocessors does not look attractive, however, for reasons of cost and maintenance. Furthermore, floppy disk units are not particularly reliable and their capacity is inadequate for the storage of experimental data.

The most promising development in this field is the Quarter Inch Cartridge (QIC) tape [9]. These tapes are being developed in order to provide backup capability for the new generation of Winchester disks. They typically hold 50 Mbytes on a single small cartridge and transfer at about 100 Kbytes/sec. Interfaces to the SCSI bus, some of which allow joint control of QIC tapes and Winchester disks, are becoming available.

2.5 Special I/O & Bus systems

Research institutes frequently need to build special electronics in order to acquire data from detectors that they have designed. Several bus system standards exist which simplify the construction of such electronics. They typically define the mechanical and electrical nature of the boards that have to be constructed, and the interconnections that are available between boards in a system. One enormous advantage of this approach is that when the special purpose electronics has been produced it can easily be integrated (at least at the hardware level) with other boards, such as processors and memories, which can be purchased rather than designed.

For about twenty years Camac was the main bus system used in particle physics laboratories, but recently both Fastbus and VMEbus have started to take over from Camac. VMEbus, defined by a group of semiconductor manufacturers (Motorola, Mostek, Signetics/Philips and Thomson), uses a Eurocard form factor and provides full 32bit address and data interconnection between up to (typically) 20 boards attached to one backplane. This interconnection uses 128 pins out of 192 available on the two

96-pin DIN connectors. Apart from some requirements for ground and power, the remaining 64 pins are free for use at the system designer's choice. If a memory extension bus is required then a recommendation exists (VMXbus) as to how to do this on these 64 pins.

2.6 Transputer

An interesting development in the field of microprocessors is the 'transputer' [10] from Inmos. What is important about this device is not so much the fact that both memory and processor are incorporated on a single chip, but rather that the same chip also supports bidirectional channels for message passing between transputers. It has been known for some time that if you can limit yourself to message passing for communication, whether between processes in a single processor, or between a process and the underlying operating system, or between programs on different processors, then there can be major benefits. These benefits arise essentially because of the greater clarity that comes from the very clean decomposition between the processes - there can be no interference effects arising because of poor synchronisation of access to shared variables, for example. However the price to pay in the past has been one of efficiency. The transputer, when it finally appears, may change this situation fairly dramatically because the message passing primitive is supported directly in the hardware, and this will make the programming of collections of multi-processors much easier.

3. MICROPROCESSOR SOFTWARE - AN OUTLINE OF THE PROBLEM

Some people believe that providing microprocessor software is simple, and some people believe that it is very difficult. In many ways both sets of people are correct. 'Classic' computer languages do not have constructs for handling asynchronous events (such as hardware interrupts), and are not designed for execution on multiple processors. Many existing real-time kernels assume that they are operating on a single processor. Software environments, including the possibility of 'host/target' debugging, normally assume that there is a single target, often going so far as to assume that it is executing a single process.

There has, of course, been academic research on the general problems of multi-processors and of languages capable of handling asynchronous events. But this work is only just starting to filter through to industry and advanced customers. In practice, products based on this research are not yet mature enough for widespread use in the physics community, because of a combination of factors, including a lack of transportability between hardware, a low level of industrial support and relatively high cost.

The absence of a fully coherent approach has not, of course, stopped inventive hardware and software engineers from implementing systems based on existing languages and monoprocessor software. When the scope of the problem can be restricted, for example to a single 'host' environment, to a single type of 'target', to a monoprocessor, to one 'classic' language and a simple interaction with the interrupt system, then software for microprocessors is sometimes easy.

While we discuss components of the software that is needed for microprocessors it is wise to bear in mind the dichotomy that sometimes exists between what is used in practice, and what a fully coherent approach would require. After discussing all of the elements we will return in Chapter 9 to review the problem as a whole.

4. ROM-BASED MONITORS

4.1 Functionality

It is common to install some read-only memory in association with each microprocessor, and to install a program in this read-only memory that is capable of performing the following functions :—

- Halting the processor
- Loading part or all of the memory contents by transferring a file from a variety of sources (down-line load)
- Dumping part or all of the memory contents to a variety of destinations (up-line dump)
- Starting the processor and letting it run freely
- Single stepping the processor
- Installing potential breakpoints
- When a breakpoint has been encountered, interrogating and altering the state of the machine in response to user commands

There are a number of commonly available ROM monitors that are provided with M68000-based systems, such as Versabug, VMEbug, Tutor etc. As we discuss further, at CERN we frequently use a locally-developed ROM monitor called MoniCa [11] to provide a uniform environment on processor boards from different manufacturers.

4.2 Input/Output

In order to ensure that the processor can be halted at all times it is normally necessary for the ROM monitor to reserve the highest priority interrupt for its own use. It also has to be able to communicate with a terminal, in order to receive commands and transmit the results, and with at least one device for down-line loading.

Although the host and target may be physically close together during development they may well be separated by quite some distance in the production phase. It is then very inconvenient to have to go physically to the target every time that the ROM monitor must be invoked. In addition there is a saving to be made if a terminal on the host system can also be used as the terminal for the target. The logical conclusion of this analysis is that it would be very useful if a ROM monitor were to be controlled by messages rather than by commands typed on a directly attached terminal. This would also allow the user freedom to select the most appropriate communication method, which may well be a Local Area Network rather than a serial communications line. It would also permit the development of a suitable protocol so that one terminal attached to a host system could control several targets at the same time.

In the case of MoniCa it was decided to use the interrupt system for all I/O. A simple channel structure allows drivers for new chips, which might be needed either for terminal communication or for I/O handling, to be incorporated into MoniCa. One feature of this approach is that higher-level software can then take advantage of the presence of the MoniCa I/O facilities. For example, when the ROM monitor has been programmed to load or dump over Ethernet it is a pity not to be able to provide an equivalent file-transfer capability to the higher levels of software.

Most commercially available ROM monitors use programmed I/O rather than interrupt-driven I/O. Especially since there are few possibilities available for reasonable debugging (it is the debugger that is being debugged), the development of drivers for I/O chips in a ROM monitor is not trivial, and the use of programmed I/O can be simpler than the use of interrupt-driven I/O. However, it is clear that this approach forces all I/O requests to be processed serially, and puts very severe constraints on the integration of the ROM monitor with the higher levels of software.

4.3 Debugging

As discussed above, debuggers allow both the creation of potential breakpoints, and then the inspection and alteration of the machine state when a breakpoint has been reached. Many existing ROM monitors provide little or no symbolic information during the debugging phase. This means that breakpoints have to be set at absolute addresses, forcing the programmer to work with several printed listings, and that the contents of memory locations containing code or data are not displayed in a way that reflects their inherent meaning. Since the CERN cross software can generate appropriate symbolic information, MoniCa is able to build up symbol tables and interpret addresses and data in various symbolic forms.

As soon as we consider debugging a multi-tasking application then a little thought shows that the ROM monitor must share the access to the interrupt system with the multi-tasking applications. If the human response times involved in debugging are not to bring the system crashing to the ground, then the dialogue between the ROM monitor and the user who is debugging one task must not block the use of the interrupt system by the other tasks. A clean implementation further requires that there is a mechanism for directing the terminal input to a named task.

As we just discussed, it is important that the debugger is able to react to messages from the host, rather than only reacting to input typed on a local terminal. This extra level of abstraction also permits the debugger to be driven by a program in another machine, rather than only by a human typing on a terminal. This can be important for the automatic testing of new or repaired or modified equipment.

4.4 Multi-task/multi-processor aspects

In a simple monoprocessor application it is easy to take the ROM monitor for granted. After all, provided that the monitor is able to download the program from the host computer, then inventive users will find some more or less adequate 'ad-hoc' method for debugging. As we move through more complex multi-task applications towards multi-processor applications, the quality of the monitor becomes more and more important in determining the overall speed at which software can be developed.

5. SYSTEMS SOFTWARE

5.1 Hardware and Software Context

Suppose that a process running on a processor has to be suspended and then resumed at a later time. If we wish to ensure that when the process resumes it executes as if it had never been interrupted, then we must typically ensure that the contents of all processor registers that might be used by other processes are saved at the moment of suspending the process, and restored at the time of restarting. The information that must be saved and restored is referred to as the process 'context' or 'state'.

The context of a process typically consists of several hierarchical components. At the lowest level there will be a hardware context that is normally well described in the manufacturer's manual documenting the processor architecture. One obvious component of this hardware context is the value of the program counter at the point of suspension, but there will typically be several other elements, such as the contents of general registers, and the internal state of the processor.

At the next level up in the hierarchy of context we normally find some information about the stack pointer(s) being used by the suspended process. In modern microprocessor architectures it is standard for procedure calls to use a stack to record the context of the calling procedure at the point of call, and for the creation of local variables. It is common for these stacks to be held in memory and for the

stack pointers to be held in registers. When a process is suspended it is important to avoid that any other (non-privileged) process gains access either to these registers or to the stacks that they define. Accordingly, the registers holding the stack pointers must be set up to reflect the stack pointer context of the next process to execute, and the system as a whole must ensure that the memory areas that hold the stack contents remain distinct.

Still higher in the hierarchy of context some language systems manage areas of memory where information is held for more or less long periods of time (certainly across procedure calls). Examples are the 'heap' used for the storage of data structures (records) created by the programmer in Algol and Pascal derived languages, and open files with their associated buffers and status in almost all languages.

In my opinion much of the discussion about the ease or difficulty of microprocessor software development should be examined in the light of the mechanisms that are being used to preserve the program context. The most effective systems that have been built in the past often needed to make major simplifications in order to ease development. On the other hand some newer systems seem to offer well preserved context in a more general environment.

5.2 Software for 'naked' microprocessors

The term 'naked' microprocessor normally refers to embedded systems containing only a processor and memory, that is to say to very small and unsophisticated target systems without mass storage. There exist a range of approaches for programming these systems where the only assumption made about the target environment is that a ROM monitor has been installed that is capable of loading the target processor from a host. The software loaded into the target processor has to be completely self-contained, and must not depend on the presence of any underlying software, whether ROM monitor, real-time kernel, stripped down operating system or whatever.

5.2.1 Polling

One approach that is very simple (but frequently adequate for the job in hand) is to load a single program into the processor and to leave the interrupt system of the processor turned off. The status of the peripheral equipment of interest must then be inspected at regular intervals and appropriate action taken as necessary.

As long as there are no severe time constraints, and if the processing required when a peripheral device requires attention can always be completed long before any other action is needed, then this approach is adequate. If reading from and writing to the peripheral device is handled in a library routine, then most languages, low- or high-level, can be used to express the algorithms of the program.

We see that the 'context' of the process running in the microprocessor has been reduced to a fairly strict minimum. Since there is only one process executing, and there are no interrupts, then context switching is not required. In the simplest examples, output is either completely suppressed or limited to printing messages on the directly connected terminal.

5.2.2 Status- or time-driven applications

There is a related approach that will deal with slightly more complicated systems. The essential idea is that on receipt of a hardware interrupt a procedure is executed which modifies a description, held in the microprocessor memory, of the state of the system that is being controlled. When no interrupts are being handled, or at fixed time intervals, the background process is executed. This process examines the state of the controlled system, according to the description that it finds in the memory, and executes the required algorithms.

The context interactions are more complex than in the previous case. To deal with the interrupt procedures first, there will normally be a special set of rules about what these procedures can do. For example they will probably need special declaration as an interrupt procedure (so that the compiler can generate the calls to save and restore the full context that is required both by the hardware and by the language), they may be limited to accessing variables that have been declared global to the background process, and calls to other procedures, including I/O procedures, may well be disallowed. This sort of restriction of generality allows for the hardware interrupt to be fielded by a high-level language procedure which can set an appropriate global variable, which will then be treated by the background process.

As before, since the background process is essentially the only fully-fledged process running in the machine, the overall software context remains very stable. The main point to watch is that the signalling mechanism between the interrupt procedures and the background must be sufficiently well-protected.

5.2.3 Modula-2

Wirth, in a very interesting paper [12], points out that an interrupt routine should be viewed as forming part of a complete cyclic process that controls a device, and that the interrupt itself should be seen as a non-scheduled coroutine transfer.

Modula-2 is a language that has the power to express the concepts of 'context' or 'state' that we have already referred to. In Modula-2 the coroutine is expressed in terms of a procedure (which can, of course, incorporate many lower-level procedures - even those whose algorithms are defined in other modules) and a workspace. It is the responsibility of the language to use the memory occupied by the workspace so that the context represented by the hardware and stack levels of Section 5.1 is correctly preserved for each coroutine across the Transfer primitive. The context of the heap and file systems is treated as common to the whole program (set of coroutines) and is therefore managed in a uniform way.

Modula-2 is therefore the first more or less available high-level compiled language which allows for the concept of the quasi-concurrent process (including an interrupt-driven process) to be expressed without recourse to assembler. Ada has similar capabilities, but has not been so widely available - that situation will change. There is much more discussion of compiled languages in the next chapter - the point being made here is that it is possible to completely program a 'naked' target system in Modula-2.

5.2.4 Interpreters

Similarly many effective systems have been completely programmed in interpretive languages. There is more discussion of the details of some of these languages in the next chapter. Generalising somewhat, we can say that interpreters are conceptually single task systems, which provide little structuring of code or data. The very simple 'context' that they require means that they themselves can be very small, and they are frequently downloaded into a 'naked' microprocessor with the program. Furthermore it is not too difficult to arrange for interrupts to trigger sections of code, in much the same way that interrupt routines are written in assembler.

5.2.5 Pure interrupt code

For the sake of completeness I will mention embedded systems that demonstrate the use of pure interrupt driven code. By 'pure' I mean that there is no background process, and that all tasks are started by an interrupt and run to completion (unless they are, in turn, interrupted by a higher priority interrupt). One example of such a system is the control software for the ERASME film measurement tables [13]. This consisted of some twenty tasks that managed several separate pieces of equipment that had either been designed at CERN or purchased from industry and integrated at CERN. The tasks were all started by an interrupt, either generated directly by the hardware or by the software in another (higher priority) task.

This example of an embedded system was implemented during the period 1972-1975 in about 16 Kbytes of PDP-11 assembler, and was then left in operation with rather little modification. It is still running today on some 15 PDP-11s and LSI-11s.

The fact that the system is written entirely in assembler has drawbacks, especially the fact that very experienced staff, understanding the assembly language, the scheduling conventions and the data structures, are required for any necessary modifications. However, leaving aside the lamentable state of compilers for any high-level languages on the PDP-11 in the early 1970s, the problems of preserving the context of such a (relatively) complex piece of software could not have been solved at that time without recourse to assembler, where all of those issues were under direct control.

It is important to realise that lack of adequate software for expressing the interactions between the multiple loosely coupled processes of embedded systems has never prevented the implementation of some very complex systems, which, moreover, demonstrated adequate reliability in the field. The real issue is to try to improve the descriptive power of the software so that embedded systems can be implemented and maintained more efficiently (faster, and with fewer people) in future.

5.3 Real-time kernels

In the past, while there have been few or no languages available that could adequately express multi-task concepts, it has been common to build multi-task software for embedded systems on top of a 'real-time kernel'. These kernels typically provide facilities for : -

- *Task management*: This covers the creation and deletion of tasks, either at system initialisation or subsequently. Frequently there are restrictions placed on some of these features - for example only the original creator or a task with privileges might be allowed to delete another task.
- *Synchronisation and communication between tasks*: Typical systems will include one or more of the following possibilities - semaphores for mutual exclusion; monitors for guarded access to shared data; mailboxes for message passing.
- *Scheduling*: Suspension of task execution (when there is nothing to do) and resumption when certain events, including timeouts, occur are normally catered for. Tasks are frequently able to modify their priority in order to influence the scheduler in case more than one task is simultaneously runnable.
- *Clock management*: A time-of-day clock will be maintained, and tasks will be able to suspend execution for a given interval. More sophisticated systems might support task resumption at a specific time-of-day. A facility that is not easy to provide correctly is a timeout associated with an input/output operation. It is important that one and only one of these actions causes execution to be resumed.
- *Memory management*: Kernels can typically be generated to run either with or without memory protection hardware. Functions will be available for the allocation of local memory for each task, normally with some rather finer divisions. For example it may be possible to allocate separate areas, or even pages, for the code and data, and further subdivision of code and data might be supported. Especially in memory-mapped systems several or all tasks may be permitted to share some areas of memory, with a variety of different addressing models being possible. This is normally useful for providing access to global data, and for saving program space by using a common library for all tasks.
- *Interrupt system management*: It will normally be possible to divert specific interrupts to be handled by a designated task.
- *Input/output, including integration of special equipment*.: Most kernels support a mechanism, frequently called a driver, which allows several tasks to obtain synchronised access to I/O devices. Several drivers for 'standard' devices, such as terminals, printers and mass storage

equipment, might already be provided. However it will be possible for the user to write a new driver for special equipment, and then to integrate this driver with the system, perhaps at various different levels depending on the exact requirements. Note that real-time kernels do not typically assume the existence of any ROM monitor, and integration of the I/O handling between the kernel and the ROM monitor, which is highly desirable from the programmer's point of view, may be very difficult to achieve as a result.

- *Limited debugging:* Some kernels offer debugging facilities. Since there are few assumptions that can be made either about the languages that have been used to generate calls to the kernel, or about the underlying ROM monitor, these debugging facilities are likely to be rather primitive. In particular they are unlikely to be able to deal with the symbols that the programmer has used.

There are a number of commercial products available, including RMS68K [14], which we recommend for use at CERN with the M68000, and pSOS [15], MTOS [16], and VRTX [17], which are all made available on a relatively wide range of microprocessors.

For people who are familiar with operating systems like RSX-11 on the DEC PDP-11 series, most of these real-time kernels provide equivalent facilities, although the emphasis is on the facilities for the target environment rather than on supporting disk-based software packages (e.g. an orientation towards memory only RSX-11S systems rather than full-blown RSX-11M).

The normal way of working with these systems is to prepare a complete set of tasks using whatever cross software is required, and then to link the tasks together with the kernel code and tables, prior to down-line loading the resulting image into the target microprocessor. In principle each task can be programmed in a different language, even including assembler, since the communication between tasks is limited to the set of available kernel calls, and does not depend on the language used.

This rather rigid separation between the code used to express the algorithm of each task itself, and the kernel calls used to express the inter-task relationships, is a rather crucial issue, and there are strong arguments both for and against the approach.

In favour of this separation we can mention :—

- The better overall modularity that it provides, since a separate base of knowledge and products is created for the 'classical' algorithmic languages and for the 'new' inter-task interactions. The base of knowledge includes the accumulated experience of the language designers, implementers, and users. Remember that we know much more about 'classical' languages than we know about expressing inter-task relationships. Do programmers really want to learn everything from scratch each time that they try out some new ideas for inter-task abstractions?
- The fact that the programmer is in control. (S)he can decide how to allocate variables to different categories of memory (shared/local, read-only/read-write, etc.) and how much protection is appropriate.

The arguments against the separation are really only a rephrasing of these points :—

- Because the syntax used by real-time kernels is too loose, it will never be possible to make any formal verification of the correctness of the inter-task synchronisation for applications that have been programmed in this way.
- The real problem is one of protection of shared variables, and this is just the area where the programmer's intentions should be made explicit to the compiler.
- If the program is to be safe we cannot allow the programmer complete freedom. The language designers know best and must be in charge.

There is also a further argument which both sides try to claim for themselves. I believe that, in practice, real-time kernels are much more likely than multi-tasking programming languages to exploit memory protection hardware. To be precise, the language approach tends to assume that the set of tasks will be placed, under compiler control, in a uniform address space. For example, the Vax Ada compiler places multiple Ada tasks into a single address space. There are two general comments to be made. Firstly, I can see no theoretical reason why a multi-tasking language processor could not exploit memory-protection hardware. Secondly, the strongest supporters of putting multi-tasking into languages will claim that hardware protection is an unnecessary luxury. In embedded systems their case is not quite as ridiculous as it may seem at first sight.

I must confess that I find it difficult to decide between these two approaches. While the majority of complex embedded software projects completed to date have probably been based on real-time kernels, there is no reason why big projects based on an integrated approach like Ada will not be successful. It will be very interesting to see whether Ada, for example, gives the programmer the feeling that the problem must be bent to the language, or the feeling that the language permits the problem to be easily expressed.

If Ada, Modula-2 and Occam represent the forefront of the push to express multi-tasking inside the programming language, then we should also recognise that there are a number of second generation real-time kernels starting to become available. These include DRM [18] from Philips, which is message based. Building the system on efficient message passing allows DRM to provide support in a multi-processor environment, and the host computer is rather well integrated with the targets, which should help significantly with the problems of debugging.

5.4 Highly efficient kernels

It is fashionable to criticise the efficiency of real-time kernels. While some of this criticism is probably justified, I personally doubt that their quality is either much better or worse than the quality of most other software. However there are some important trade-offs that can be made between a fast kernel and one that is safe. The programmer naturally expects to obtain the speed of the first allied to the security of the second, but it is an illusion that this can be obtained.

By making assumptions about addressing and memory layout, and by letting short code sequences triggered by interrupts run in supervisor state, the manufacturers of some network gateways have produced kernels for their own use which allow for the tasks to be programmed in higher-level languages (typically C) and which provide very good performance.

5.5 Single user operating systems

The 'minimum' requirement that a user normally puts on an operating system is that it manages the mass storage space available, and provides both a command interface and a programmable (system call) interface to the operating system functions. On simple hardware, such as program development systems built up from home computers or from bus-system based microprocessors, the user normally needs access to an editor, compiler and/or assembler, linker, and debugger. The programs are typically loaded from a (system) floppy disk, the input files are called up from a user floppy, and the output files and perhaps temporary work files are written back to the same user floppy.

Writing and maintaining the operating system is (relatively) simple because it is assumed that only one single user is on the system at any time, and only one single task can execute at any moment for this user. Much of the complex software required in a full operating system to provide protection between multiple users of main memory and mass storage disappears.

In CERN there are users of Flex, CP/M and MSDOS or PCDOS on systems based on the M6809, M68000 family, and IBM PC series respectively.

5.6 Full operating systems

Although single user operating systems can provide an extremely useful and productive environment, they have two sets of drawbacks. The first arises because of their (typical) limitation to single tasking. The only task that can be executed is normally the last program that the user has loaded from the mass storage. Therefore if an unsolicited request arrives, such as any message over a network, then there is no code available to process this request. Accordingly, simple operating systems can only act as clients in a network, for example running programs on user request to interrogate other computers in order to see if there are any messages, rather than receiving these messages automatically themselves.

The second set of drawbacks arises because of the 'flat' nature of a single user file system. The user disk typically does not have structuring facilities, such as separation of one user's files from those of another user (remember that there is only supposed to be one user), or the possibility for the user to separate the files relating to different activities into different logical areas. While this lack of separation does not matter much if the system is based on the use of floppy disks, it becomes much more of a nuisance when Winchester disks are being used. It is rare that any development system based on a Winchester is really used only by one person and nobody else, and sharing files on a large flat filestore is not an easy business.

To summarise, a full operating system supports multiple users, both on the mass storage, and in main memory, where users typically run in separate address spaces. Furthermore each user can typically run several tasks concurrently, with inter-task communication and scheduling. Unix and Concurrent-DOS are examples of full operating systems that are frequently made available on microprocessors.

In the context of developing software for host/target embedded microprocessors, operating systems are normally more important for the host systems than for the targets. The distinction is not, however, rigid and several examples exist where people do as much development as possible on a host system, and then transport the partially debugged application to a target which provides a (minimal) subset of the host operating system, accessible to programs through the system call interface. A major advantage of this approach is that the host and target environments are fairly similar and, as a result, the process of user education is simplified. At CERN much development of 8-bit M6809 systems is carried out on hosts running the (single user and single task) Flex operating system, and then moved to targets that support an extremely limited set of Flex facilities.

It is also interesting to note that although some real time kernels were defined as such, others are nothing more nor less than the base on which operating systems were built. For example Versados, which is a single user, multiple task operating system from Motorola, is constructed on top of the RMS68K real time kernel. At CERN we use RMS68K extensively, and Versados hardly at all. Most operating systems for technical minicomputers (PDP-11, Nord-100, etc.) have been able to support both the multiuser requirements of host systems and the needs for inter-task scheduling of target systems. This has been convenient, again since it gives the user a more integrated environment across the host and target, but these systems are normally only available for the manufacturer's machines. It is indeed a pity that there is nothing corresponding to the 'real time kernel' of Unix, since Unix is an obvious choice for the host development system.

6. LANGUAGE SELECTION

6.1 Community requirements for adopting a language

Much of our education as children is devoted to acquiring and improving the capacity to express our thoughts in our mother tongue. We devote a major effort to this process in order that we can later communicate with the whole community. In Europe, depending on where we are born, we may find that we have been taught one of about twenty major languages (Czech, Danish, English, French, German, Polish, Portuguese, Russian etc.) While some people have no need to speak any language other than the one that they are first taught, others find that they need to learn additional languages.

To solve the problem of communicating between different countries, Latin was used as a lingua franca for monks and other clerics during the Middle Ages. Diplomats in the eighteenth and nineteenth centuries really did negotiate in French more or less everywhere. And many people today learn English as a subsidiary language in order to communicate with business or professional colleagues whose mother tongue is different from their own. The reasons for the emergence of English as a lowest common denominator of international discussion are multiple, but one result is that some level of communication is normally possible between any two people attending a scientific congress.

We note that although each individual (human) language is very general purpose, and can be used to express ideas on electricity, magnetism, computing, ecology, gardening, life and love, a normal individual will only be able to express and understand a limited subset of such ideas. (S)he may well lack the background and vocabulary to extract full benefit from a given lecture, conversation or book. Nevertheless we do not normally use this fact to argue that the general purpose nature of human languages is a bad thing.

In much the same way that physicists around the world decided on English as their common language for human communication, they adopted Fortran as their common language for communicating computer-oriented information. This fact has often puzzled, even perhaps annoyed, specialists in computer science, because Fortran is clearly not the 'best' computer language in the world. I believe that the confusion arises for a number of reasons. The oral and written development of European languages has taken place on the timescale of centuries, but has produced general purpose languages which, with appropriate vocabulary, can be used to express thoughts in many fields. The development of computer languages has been crammed into the past thirty years, and has often given rise to special-purpose products.

The adoption of Fortran by the physics community came about because it was the best approximation to the general purpose language, capable of abstracting most of the computer-oriented ideas that one physicist wished to express to another. Other factors were, and are still, involved, such as the fact that Fortran was the first compiled language, that it was developed specifically with the abstraction of mathematical expressions in mind, that the shortcomings of the language have led to the production of software libraries (vocabularies) that alleviate many of its defects, and that there is a huge investment in accumulated expertise. Nevertheless I feel that the ambition of the physicist to learn only one general purpose language for his or her computing needs has never been really understood or accepted by the computer science community. The physics community wants to exchange data and programs around the world, and to be able to study them in depth, when necessary. They do not want to have to learn a new computer language merely in order to benefit from another colleague's experience. Although Fortran is certainly not the ideal way to express some ideas, the advocates of any successor must realise the size of the task that they are facing.

6.2 Language Selection for Embedded Microprocessors

There are four major areas that receive extra prominence when we evaluate a language for use in an embedded system, over and above its suitability for general purpose use.

1. The ability to express tasking, including task dispatch on interrupt.
2. The ability to deal with special purpose hardware, including bit-fields in control and status registers, and the associated interrupts.
3. The ability to debug the embedded system from a host computer. As we have already discussed, it is not so much the design of the language itself, but rather how well it has been integrated with the ROM monitor and real time kernel, that is important in this context.
4. The ability to express allocation of processes across several processors. This is a very new field, and only Occam really addresses the issue.

6.3 Mainline compiled languages

By the time that they are expressed in a form that most humans can understand, language definitions are rather long, say over 100 pages. Most people only get to know a subset of a given language, and it takes an enormous time to get to know even a single language really well. The following sections try to make a comparison of several mainline compiled languages according to several different criteria. Complete rigour in such an analysis is almost impossible, and certainly takes much more space than I have available. I apologise in advance if my summary or prejudices do not conform to the reader's liking.

The main languages treated in this section are (in alphabetical order) Ada, C, Fortran-77, Modula-2 and Pascal. Comments are occasionally made about other languages, including interpretive languages. Forth, Concurrent Pascal and Occam are briefly discussed in Section 6.4, and some aspects of Basic are treated as part of the presentation of PILS in Chapter 10.

6.3.1 Control abstraction

Task abstraction is treated separately (below). Although it is possible to spend many pages discussing the exact differences between the languages, and although I believe there are significant potential gains to be had by providing better structured control transfer facilities [19], I will summarise by saying that all languages, except Basic and its derivatives, have similar, and more or less adequate control abstraction.

Basic is an unmitigated disaster from this point of view, allowing GOTOs everywhere, not providing an IF-THEN-ELSE, and not supporting parametrised subprograms. Of course individual Basics have been upgraded to resolve these issues, but normally in different ways.

Fortran-77 is deficient with respect to the other languages since it neither provides a WHILE or UNTIL style of loop, nor a proper multi-way choice (CASE). On the other hand it does support single-entry multiple-exit subroutines, which are a potentially powerful and well-structured way of treating unusual circumstances that have been detected by software, and which are missing from all of the more 'advanced' languages.

All languages except Modula-2 still allow access to the GOTO primitive.

6.3.2 Data abstraction.

6.3.2.1 Arrays

Fortran has been extensively used for mathematical computing, and provides reasonable support for array handling. In Fortran-77 arrays may be of any dimension between 1 and 7, and the subscript references can be arbitrary expressions that are truncated to integers. Subarrays can be passed as arguments to subprograms, but the programmer is responsible for arranging that the definitions in the calling and called subprogram are consistent, and this is not trivial.

In Fortran-8X, which is being actively worked on in the X3J3 committee of ANSI, vastly improved facilities for array handling have been proposed. These would include, for example, expressions whose value is an array, inbuilt functions such as MAXVAL, PRODUCT and DOTPRODUCT, and facilities for passing sub-arrays to subprograms.

From the point of view of the scientist one defect of Fortran is the fact that the array notation is exactly the opposite of the standard mathematical notation. This defect has been corrected in all subsequent languages, which serves to increase the problem of migration from Fortran to anything else, since the array handling libraries will be incompatible.

In Pascal array declarations are fairly similar to those in Fortran, but there are problems in passing arrays as arguments to subprograms. In the original Jensen and Wirth definition [20] it was impossible to pass an array of arbitrary size and/or dimensionality to a subprogram that would perform some general operation (such as finding the largest element, or calculating the cross product). ISO Pascal Level 1 now defines conformant arrays so that an array of arbitrary size, but not of arbitrary dimensionality, can be passed as a parameter.

Modula-2 supports conformant array parameters, but in a way that is both different from that used in the ISO Pascal Level 1 and which is inadequate for dealing with multi-dimensional arrays.

Ada has an extensive and presumably adequate range of array definitions, and allows subprograms to handle arbitrarily sized arrays. However there is no attempt to provide intrinsic mathematical operations on arrays, since this is regarded as a job for an Ada package, rather than for the language itself.

6.3.2.2 Bits

Compiler writers have always been reluctant to let the programmer have access to the bits in a computer word, presumably because much of the early development of compilers was precisely in order to protect the programmer from needing to worry about trivial details of the machine architecture. However when controlling hardware in an embedded system we must be able to deal with things like control and status registers, where the operations are defined in terms of bit patterns.

Ada appears to be the most powerful language in this context. It is possible to give symbolic names to arbitrary bit fields at arbitrary memory locations, and to manipulate these variables using all of the (formal) protection accorded to any enumerated type. In view of the power of these features, functions for bit manipulation become redundant, and are not provided.

C also manages fairly well. Bit fields of variable width can be incorporated into user-defined structures, and a pointer type can be used to locate the structure at a given address in memory. Problems can arise because the method of packing the field into a word is at the discretion of the compiler writer (in particular there is freedom to pack from the left or from the right), and because it is up to the programmer to avoid overflowing the field that has been defined. An alternative approach using C is to operate on a storage location, reserved say with an 'int' declaration, and then to use the logical shift operators '>>' and '<<', and masking operators ('&' etc.) to effect the required bit manipulations.

The other languages do not allow for the symbolic naming of arbitrary bit fields, but frequently provide some support for bit manipulation, normally on the assumption that the user has defined a variable that will occupy one storage location. Bit manipulation functions defined by the Instrument Society of America (ISA) [21] have become available as intrinsic functions with many Fortran-77 compilers during the past year, although the Fortran-77 standard itself has nothing to say in this area. Modula-2 has rather fewer facilities than we might expect to find, given the general orientation of the language, and Pascal is almost entirely deficient in this area. Of course the programmer is free to use library routines to work around the deficiencies of any given language, but that has a strong impact on readability and transportability.

6.3.2.3 Enumerated types

Most of the languages that we are considering give the programmer the facility to define private data types by enumerating all of their possible values. This is missing entirely from Fortran-77, although the use of PARAMETER statements can help to avoid the worst errors arising from conventions like '17 means a pi meson and 25 means an antiproton'. That is only a small consolation, however, since the compiler is unable to make the valuable checks that these user defined types are being used correctly (in a formal sense).

6.3.2.4 Hiding of data structures

The module concept of Modula-2 and the package concept of Ada permit the programmer to define data structures that are appropriate to the application, together with appropriate operations that may be performed on the data, while protecting ('hiding') the data structure from other, unauthorised, access by the program.

This facility can be particularly useful in embedded systems, where concepts such as the queue and the buffer can then be modelled in the language in a very elegant way. While Pascal and C can manage the definition of the data structure, they do not afford the protection provided by the module or package. As we see in the rest of this section, Fortran-77 does not provide for user-defined data structures, a serious defect that will be corrected in Fortran-8X.

6.3.2.5 Real numbers, extended precision and complex arithmetic

Only Fortran-77 defines all of these data types as part of the language. Pascal and Modula-2 define real arithmetic and are rather ill-at-ease when treating double precision and complex. It is possible to map these types onto user-defined structures, but the individual components must be manipulated separately, which results in the loss of the clean mathematical syntax.

Ada defines real (float) in the language and support for extended precision and complex arithmetic can be provided through packages in a reasonable way, since overloading of operators allows the mathematical syntax to be preserved. However, although packages have some advantages, especially to language designers, they do reduce the portability of programs, since the package may not be available (paid for and installed) everywhere that a compiler is installed, and there may be several different packages for the same task, which risks to complicate user education severely.

In addition Ada provides extensive support for a whole new range of facilities where an algorithm can itself define the required precision, rather than forcing the use of the precisions provided by the machine. This support comes through statements like

```
type myreal is digits 7
```

which indicate that the algorithm requires 7 decimal digits of precision for all variables of type myreal.

C defines real and double in the language, and support for complex arithmetic can be provided using the typedef and struct facilities, although again without retaining the mathematical syntax.

6.3.2.6 Records

The concept of the record allows the programmer to define and operate on storage units that (typically) consist of several variables and occupy more than one memory location. The idea is completely absent from Fortran, but present in all of the other languages.

6.3.3 Debugging

The strong type checking of Pascal, and the related checking of separately compiled modules seen in Ada and Modula-2, do indeed catch errors at compile time rather than letting the mistake propagate through to execution, where it is difficult to localise. As languages, neither Fortran-77 nor C can claim to pay much attention to ease of debugging.

However, in practice, for a professional programmer setting up an embedded system, the quality of the symbolic debugging provided by the ROM monitor, and the degree of integration between the monitor, the real time kernel and the cross-compiler, is likely to be at least as important in determining the speed of development as the language that has been used. This will be even more true if the hardware is not operating completely reliably, which experience indicates to be often the case.

Interpreters, since they avoid the long turnaround time of the edit-compile-link-download sequence, and because they do give symbolic access to the data, are often claimed to facilitate debugging. While this is certainly true for small programs, it is difficult to maintain the advantage as the program size increases. For large programs the ability to structure both the code and data becomes crucial, and these are not areas where interpreters have a very good reputation.

6.3.4 Error recovery

It is normally important that an embedded system keeps running. The programmer must therefore make sure that the occurrence of any abnormal hardware or software condition is detected, and appropriate action initiated. The cleanest solution is to provide language abstractions that permit the programmer to be informed, through an 'exception' or software interrupt, whenever anything unusual happens.

Of the languages we are considering only Ada treats this point as part of the language. Exception handlers can be defined at various positions, and both forced entry (hardware conditions) and programmable entry (to cope with a problem recognised by the software) to the currently active exception handler are supported.

It would presumably not be too difficult to arrange to recover from some hardware problems in a Modula-2 program by trapping the associated interrupts, but the concept of a software exception is missing from the language.

Fortran-77 treats the recovery from I/O errors very thoroughly but the language has nothing to say about other errors, such as overflows or addressing problems. C and Pascal have no language features dealing with error recovery.

Note that some operating systems do provide a reasonable amount of error recovery, accessible to programs written in any language, through a library. A particularly uniform implementation is available on the Vax under VMS [22].

6.3.5 Expression abstraction

To first order all of the languages considered have an adequate power of abstraction for mathematical expressions. Fortran supporters always complain about the restrictions on exponentiation that are found in most other languages, and, as we have already discussed, Fortran-8X is likely to have extremely powerful abstraction for matrix (array) handling.

6.3.6 Interrupt handling

Only Ada and Modula-2 attempt to deal with the programming of an interrupt handler directly in the language. C, Fortran and Pascal have nothing to say in this area, and need to rely on the facilities of a real-time kernel if they are to be used for handling interrupts.

The unstructured nature of Basic maps fairly well (like assembler) onto the interrupt system of a normal microprocessor, and some manufacturers have provided extensions to Basic that handle interrupts. One example is Power Basic from Texas Instruments, which gives the programmer control over the interrupt system of the TI9900 microprocessor. It is possible to mask interrupts, attach a given line of the program to a specific interrupt, and return from interrupt.

6.3.7 Libraries and packages

From very early days Fortran programmers have been used to working with pre-compiled libraries, and the requirement in some standard Algol and Pascal systems to recompile all libraries at each occasion was always seen as a serious weakness.

Modula-2 and Ada have now corrected this defect, and, by incorporating checks to make sure that the interface between the invocation and the library is consistent on both sides, they offer major advantages over the other languages in the area of supporting large software packages.

6.3.8 Readability

My opinion, or perhaps prejudice, is that languages like C and Forth are too terse for the general reader to understand. Of course specialists can understand them, and even prefer them over all other expressions of algorithms, but they are not languages that I would like to have to read in order to fix a problem in some malfunctioning embedded system in the middle of the night. They are just too obscure for my taste. I am happy with the level of verbosity of Ada, Fortran-77, Modula-2 and Pascal.

6.3.9 Separate compilation

The analysis of a typical modern physics experiment requires the production of some 100,000 lines of software (mainly written in Fortran-77). The several programs comprising this software will all make heavy use of libraries, such as the CERN program library, which contains some 250,000 lines of code. We are installing programs in Motorola 68000 based embedded systems whose size can be measured in units of tens of thousands of lines.

Given the size and performance of present computer systems, which compile at the order of a few thousand lines per minute, it is a practical necessity to be able to compile elements of the program separately.

Freewheeling Fortran, which avoids checking anything if it can help it, has always allowed for separate compilation, and C is similar. Pascal, at least in the standard incarnations, specifically excludes this possibility as dangerous, since the compiler cannot check that the definitions provided in the different compilations are the same. Both Modula-2 and Ada allow separate compilation with adequate checking.

6.3.10 Standardisation

Ada is standardised (ANSI/MIL-STD-1815A-1983) very strictly by the American Department of Defence, and is on the way to international standardisation. It is claimed that neither supersets nor subsets of Ada will be permitted, and that all products claiming to be Ada related will have to be periodically resubmitted for testing that they conform to the specifications of the standard.

The *de facto* definition of C is the Appendix of Kernighan and Ritchie's book [23]. However, an ANSI Draft Standard has recently been published (X3J11/85-045, April 30 1985) and this will probably become the formal standard, perhaps after some final modifications.

Fortran-77 is defined as American National Standard FORTRAN-77 (ANSI X3.9-1978). The previous version of the standard was ANSI X3.9-1966. Work is under way on the next version of the Fortran standard, normally referred to as Fortran-8X.

Modula-2 is defined by Wirth's book 'Programming in Modula-2' [24] and the additions given in [12].

Following a long period when the Jensen and Wirth book [20] was the *de facto* standard, a formal standard [25] for Pascal (ISO 7185/BS 6192) was adopted in 1982. For scientific work the Level 1 version of the standard, which supports conformant arrays, is normally used.

6.3.11 Storage abstraction

Note that this section deals with the abstraction of the memory of the microprocessor, as seen by the programmer, and not with the abstraction of data, which is treated separately.

When programming 'systems' type applications in a microprocessor, it is frequently necessary to manage an area of memory that contains many different types of data (such as characters, short integers, booleans, long integers and floating point values) at random locations. There can be two related requirements on a language, firstly the need to be able to manipulate storage units that the microprocessor hardware understands (bytes, halfwords, words etc.) rather than the abstract data types that the compiler normally deals with, and secondly the need to overlay several different views of this area of storage, for example to be able to treat it as containing either all bytes or all words.

The correspondence of data types with underlying storage units (bytes, words etc.) is normally regarded as something to be hidden from the programmer. With the exception of Ada, none of the languages considered really tries to come to grips with this difficult area. C offers a rich set of fundamental data types, but offers no guidance as to how they should map onto storage units. On any given machine it will normally be possible to select a fundamental data type to represent 8-, 16- and 32-bit units. However there will be little chance to keep such a definition portable. The best that can be done is normally to program using private data types of the required size, and modify an extremely localised definition for each new machine. Pascal has fewer fundamental data types, and it may well be necessary to use packed enumeration types to achieve the desired effect. Defining 8- and 16-bit quantities in Fortran can be non-trivial, although some compilers have such data types as extensions. Modula-2 allows access from the module SYSTEM to types WORD and ADDRESS, which is better than nothing, but not does not give everything that is needed either. I believe that Ada solves the problem correctly, and in a (relatively) transportable way, but I would like to confirm my ideas on real compilers before committing myself.

As for the second requirement, the overlaid view of storage, this can be achieved in C using a combination of structures and unions. In Ada there is a generic function UNCHECKED_CONVERSION that can be used to deal with the situation in what looks to be an adequate way and it has the nice feature that it is made clear exactly where the strong typing rules are being flouted.

Strictly speaking this sort of storage abstraction is not allowed in Pascal and Modula-2. Variant records, which programmers regularly use to provide this facility, are supposed to generate a run-time error if they are used to extract a value from a record that was inserted as a different type. The cost of such checking is, however, prohibitive on most existing computer architectures.

The Fortran-77 standard is also rather restrictive, but in a different way. Storage equivalence has always been allowed in Fortran, but the standard is less permissive than is commonly supposed. It is not legal to equivalence characters with numeric or logical data. Furthermore numeric data (integers and single precision reals) and logical data are required to occupy the same amount of storage space, which is a little bizarre.

6.3.12 Task abstraction

Only Ada and Modula-2 provide task abstraction. The other languages have to depend on real-time kernels or the related approaches discussed in Section 5 if they are to be used in a multi-tasking environment.

The approach of Modula-2 and Ada to tasking is very different. Modula-2 provides a very low-level primitive, the co-routine transfer, whereas Ada uses a very high-level concept, the inter-task rendezvous. In [12] Wirth shows how the co-routine transfer can be used to provide libraries that support different high-level models, including signals and the Occam channel.

Papers have been written which indicate potential problems with the efficiency [26] and with the robustness [27] of the Ada inter-task rendezvous. Note however that some changes have been made to the Ada tasking scheme since the paper by Roberts et al. was written. Another paper [28] indicates some more general problems with Ada tasks, seen from the point of view of structured design. Although it is too early to judge whether the critics are completely correct, I do feel that Wirth's approach gives the programmer greater freedom to choose a method for tasking that is appropriate to the application.

It is perhaps important to note that the approach of both languages assumes that the 'program' for the whole application is fixed at compile time. This restriction can be very serious, or even intolerable, in a truly distributed application such as the control of a large accelerator or physics experiment. The need to be able to add additional tasks to an application running on one or more processors, without interrupting the system for a complete reload, is very real.

A very common requirement in embedded systems is the need for a task to sit waiting for one of several events to occur, and then to dispatch depending on which event took place. This is sometimes referred to as the generalised event wait. The speed with which the waiting task is resumed, and the action that is required is recognised, governs much of the responsiveness of many embedded systems. The choice (if any exists) between a suspended wait and a processor loop can also have an important impact on the performance of the overall system. Note also that it is uncommon to be able to associate directly and unambiguously time-outs to specific events (or input/output operations) in either multi-tasking kernels or languages.

6.3.13 Terminal dialogues

Part of good programming design is making sure that the human interface presented by the program can be easily understood by an inexperienced user. The ability to format a meaningful message for information, or as part of an interactive dialogue, is frequently important. Ada, C and Fortran-77 (the latter to the surprise of many people who remember Fortran-66) all deal with this problem quite well. Pascal, and to a lesser extent Modula-2, have trouble because of the rather rigid definition of the length of a string.

6.3.14 Transportable I/O

For quite a time Fortran was unusual in defining the allowed input/output operations as part of the language. Especially with the advent of Fortran-77 there is a clear definition (with several warts it must be admitted!) for formatted and unformatted access to terminal, printer and mass storage.

Pascal provides similar facilities to Fortran, but they are not quite so comprehensive and do not cater for unformatted (binary) data. C and Modula-2 both define a standard library (the module InOut for Modula-2 and the library stdio for C). The Ada programmer has access to a set of predefined packages, such as `DIRECT_IO`, `TEXT_IO`, or `LOW_LEVEL_IO`, and the `with` and `use` constructs can be used to select the appropriate set.

It should be noted that the transportability of input/output tends, in any case, to be more important when using these languages for off-line programming than in an embedded environment. Not infrequently the only input and output for an embedded system is likely to come from special purpose hardware, so bit and interrupt handling possibilities will be much more important than predefined I/O packages. Besides the transportability of the programs, the transportability of the resulting data from one processor architecture to another is a thorny subject.

6.4 Other languages

6.4.1 Forth

Forth is special, and seems to arouse either strong positive or strong negative reactions. It is above all a functional, rather than an algorithmic, language. Probably the best known functional language is APL, and we can see the same type of strong reactions there too. Functional in this context means that the programmer starts with the pre-defined set of words (function names) supplied with Forth, and by defining new functions, which are typically very short, or re-defining the supplied versions, arrives at the point where one function can be invoked that 'does the job'.

The special nature of Forth does not stop there, however. The requirement to run on really small (8Kbyte) microprocessors was originally a very strong influence on the design. It has led to a number of choices being made that I believe make the life of the programmer more difficult, but simplify things for the implementer. In particular the language is very stack oriented, and requires that expressions are given in reverse Polish notation. Accordingly an expression which looks like

$$\text{scale} * (\text{reading} + \text{offset})$$

in most of the languages that we have been discussing becomes

$$\text{reading offset} + \text{scale} *$$

in Forth. Note, however, that the Forth programmer is much more likely to define functions called `scale` and `offset`, rather than have variables of those names.

Forth provides neat facilities for integrating assembly language fragments, required for example for handling microprocessor I/O or interrupts, into the function definitions. Furthermore there is support for multi-tasking provided by a round-robin scheduler. This runs each task until it requires to wait, and then picks up the next runnable task in the predefined sequence.

The overall performance of Forth programs seems to be rather respectable, and the space overhead of the system itself is very small. In addition Forth is more than a language, and does offer a reasonable degree of integration of language, multi-tasking and ROM monitor. If you are able to work in reverse

Polish (I admit that I am not willing to) then Forth may well be something that you could profitably use for developing software for embedded systems.

6.4.2 Concurrent Pascal

Concurrent Pascal [29] is an extension of the Pascal language to support concurrent processes. Probably the main advance of Concurrent Pascal was the introduction of the four new types, process, monitor, class and queue. Conceptually a process defines a data structure and the sequential program that operates on the data structure, while a monitor defines a (shared) data structure and the operations that concurrent processes may perform on it by calling procedures. A process has exclusive access to the shared data structure while it is executing a monitor procedure, and this means that monitor procedures can be used to synchronise processes and exchange data safely among concurrent processes.

The class is a degenerate case of the monitor, defining a (private) data structure and the operations that a sequential process may perform on it by calling procedures. The queue can be used by monitor procedures to control medium term scheduling, and in particular can be used to delay a calling process until the monitor can complete the requested function. Note that the word monitor is being used here in a much more precise way than when we talk about a ROM monitor.

As far as I am aware Modula-2 has achieved a far higher degree of acceptance (especially in industry) than Concurrent Pascal. As far as I can judge, this is probably because Concurrent Pascal makes no attempt to deal with either separate compilation or with microprocessor oriented features such as bit handling or interrupt handling. Nevertheless Brinch-Hansen's book is a very clear and thought provoking introduction to many of the problems of concurrent programming.

6.4.3 Occam

Occam [30] is a very interesting language that can be thought of as the assembly language for the transputer. In particular it tackles the issues of expressing algorithms that can be executed on multiple processors. However it is also a good example of the problem noted in Section 5.3, where the programmer is required to throw away most accumulated experience in order to benefit from these exciting new concepts. One example of the sort of syntactic salt that the pioneer needs to swallow is

```
x = [start FOR 1 + (stop - start)]
```

in place of

```
FOR x = start TO stop
```

Important new concepts are SEQ and PAR, which give the programmer the duty to think carefully about which program actions can proceed in PARallel, and which are constrained to be performed in SEQUENCE. Processors are able to send or receive values on I/O channels connected to other processors.

```
thatchannel ? thisvariable
```

will set thisvariable from the value coming down thatchannel, and

```
anotherchannel ! anothervariable
```

will send the value of anothervariable onto anotherchannel. It is fundamental to the idea of Occam that processors should not communicate through shared memory, but rather through I/O channels. This method is, of course, much safer. The problem has always been one of efficiency, but the balance is certainly swinging towards the Occam concepts.

6.5 Interworking and libraries

It is not uncommon that a project decides that it would like to program a new application in one language while making use of already existing libraries that have been programmed in a different language. At the simplest level it has been very common to call libraries programmed in assembler for ultimate efficiency from applications programmed in Fortran or Pascal. At a more complex level, PILS (which we will discuss later) is programmed in Pascal and is able to call libraries written in Fortran-77 and assembler.

If you wish to make use of different languages in this way, which is sometimes called 'interworking', then you need to take care of a number of compatibility issues. These include :—

- The data type conventions, especially making sure that parameters passed to subprograms are defined in compatible ways. While it is relatively easy to transfer numeric information, since most languages have conformable views of integers and reals, there can be problems with logicals. In many implementations Pascal booleans occupy one byte whereas Fortran logicals are required to occupy the same space as numeric data (typically four bytes). Accordingly, transfer of logical parameters or function values between Fortran and Pascal tends to be a mess, and a mapping onto numerics or bit-strings is required on one side or the other.
- Array conventions. As we already noted, Fortran stores multi-dimensional arrays with a non-mathematical convention, in contrast to all of the other languages. This invalidates the use of array handling libraries that have been written in Fortran, or in assembler assuming Fortran storage conventions. On the other hand the transfer of singly dimensioned arrays (vectors) of fixed length between different languages does not give too many problems.
- Calling conventions. Any given compiler must make assumptions about how parameters are passed to a subprogram, how values are returned, which machine registers are freely available, what memory areas are available for temporary workspace, how abnormal conditions are handled, and so on. If we wish to support interworking then all compilers must produce code that respects the same calling conventions. Although these conventions are normally clearly defined by the manufacturer of a mini-computer, see for example [22], they are not well defined for most microprocessors. At CERN we have reluctantly (because we do not think that it should be our job) defined such a convention for the M68000 family of microprocessors.

6.5.1 Runtime libraries

When interworking is supported between several languages, then, ideally, only one library is needed to support the runtime systems. These runtime libraries typically support the I/O and mathematical functions defined in the language, together with any more exotic requirements such as error recovery and tasking. Of course not all of the languages use every feature, but it is necessary nevertheless to provide access from each language to every feature that it requires. Under Vax/VMS there is a common runtime library that is held in a shared virtual segment. This means that each compiler generates code to call, as appropriate, the entries in the common runtime library. Since that segment is virtual, only those entries being used will occupy physical memory.

6.5.1.1 Standard I/O

One obvious area that should be available to all languages is support for I/O to standard devices. For an embedded microprocessor there may be few, or even no directly attached devices. Libraries are commonly available for controlling a terminal and a printer, and sometimes for supporting I/O to disks, serial communications lines and networks. As we discussed in Chapter 4, at CERN the MoniCa ROM monitor makes these libraries immediately available to any language processor that wishes to use them.

6.5.1.2 User libraries

A given organisation or project is likely to have selected a set of libraries for use in many applications. This may include libraries for graphics, where GKS [31] and related standards are starting to have a major influence, for histogramming, and for special I/O. At CERN we make extensive use of the in-house HBOOK [32] for histogramming and of standard routines [33], [34] for dealing with Camac and Fastbus equipment.

6.5.1.3 Mathematical libraries

Mathematical functions such as sine, arctangent and logarithm need careful coding in order to cope accurately with all possible values of the input arguments. This should be done once for all languages, insofar as they have compatible requirements. However, to give one example of a problem in this area, some versions of Pascal have a definition of the MOD function which does not correspond to the mathematical definition of modulo.

6.5.1.4 Conventions

We have already discussed some of the more obvious issues of programming conventions for interworking above. A clear definition of the stack layout, and allocation of responsibility for entering and removing information from the stack at procedure call, is obviously needed. Some languages, however, manage a 'heap' that holds user defined data of (a priori) unknown size. If more than one such language wishes to have access to heap data then it is clearly important to define clear rules for the extent of the heap, and how it is managed when there is no space left, etc.

If we are dealing with a multi-tasking environment, which will be common in embedded microprocessor applications, then the library should be capable of dealing with two 'simultaneous' calls to the same procedure. The cleanest way to do this is to separate the code and data spaces needed by the library, and to allocate the data space for each task in a separate address space. This requires, of course, that the library should be coded in such a way that the data accesses do not depend on the position of the code, since the relative position of the code and data will be determined at link time. While this remark sounds almost trivial, it is not always easy to achieve, especially if we wish to use a compiler to generate the library.

Finally, if we start dreaming, we could imagine languages that support multi-tasking having a compatible definition of the task context and invocation methods, which would enable a task written in one language to interwork with a task written in a second language. But we must keep our feet on the ground, and learn to walk before we run.

7. CERN CROSS SOFTWARE ENVIRONMENT

At CERN we have been working on the integration of a cross software suite for the Motorola 68000 family of processors [35]. At the lowest level this is based on the MoniCa ROM monitor, which provides a uniform user interface, symbolic debugging for all supported languages, and channel-oriented I/O support for languages that wish to use it. If a real-time kernel is required then we recommend the use of the RMS68K kernel from Motorola, for which we provide a system generation procedure. MoniCa is integrated with RMS68K in the sense that it is possible to debug one (or a subset) of the tasks loaded into a target. A cross assembler, and cross compilers for C, Fortran-77, Modula-2 and Pascal are available.

At the time of writing both the C and Fortran-77 cross compilers are used by CERN under licence, and they are not as well integrated with the debugger as the Modula-2 and Pascal products. During the

next twelve months we intend to write new front-ends for these compilers, and to extend the ability of MoniCa to support 'remote' debugging of (one or more) targets from a single host. Work is also under way to exploit the full power of the M68020 processor.

8. SOFTWARE TOOLS FOR MICROPROCESSORS

8.1 *Language tools*

When developing algorithms on a host computer the programming team needs access to the same sort of tools as are required in a normal (off-line) mainframe environment, namely design methodologies, editor(s), source code maintenance, and version control including history and multi-version generation. The possibility to generate the text to be compiled by using a pre-processor which will concatenate input from various files, and conditionally select elements of text, can be of enormous benefit to the people who have to write these tools. If this preprocessing can be undertaken by the compiler itself, there are additional benefits of efficiency and portability. As far as I am aware, C is the only language that achieves this effect in a standard way, using the `#include` statement, and the `#if` - `#endif` and related constructs.

Of the languages we have considered both Modula-2 and Ada support separate compilation with checking, and we have a set of tools at CERN for the maintenance of Modula-2 modules in a consistent way.

The Unix tool 'make' often helps enormously in the development of a complex set of software, since it allows for the process of regenerating the application following a modification to be handled in a consistent and optimal way.

8.2 *Design methodologies to support multi-tasking*

The classic overview of methodologies for designing software is provided by Freeman and Wasserman [36]. It must be admitted, however, that there has been little practical use of these techniques in data handling for physics until recently. This was probably because there were no adequate tools available to support these methodologies, and the size and complexity of physics programs excluded the use of the techniques in pencil and paper mode. Recently, however, reasonable tools for the support of Structured Analysis and Structured Design (SA/SD) on Vax computers running either Unix or VMS have become available, and SA/SD is being used by one LEP experiment, and for some other projects at CERN.

Furthermore, none of the techniques reviewed by Freeman and Wasserman were really applicable to the world of real-time processing. Probably the most widely-used methodology in that environment is MASCOT (Modular Approach to Software Construction, Operation and Test), which is recommended by the British Ministry of Defence, and has been in use since 1972. MASCOT is, in principle, language independent. For a long period, however, the main toolset required the use of the Coral language, and this may well have prevented the wider dissemination of the MASCOT philosophy. A toolset for use with Pascal is now also available, and a discussion of the use of Ada with MASCOT can be found in [37].

There are rumours that modifications of SA/SD to deal with the problems of real-time have been undertaken successfully, but I do not have any more definite information yet.

Considerable effort is also being invested in the development of program support environments (APSEs) for Ada. However these are typically designed to keep track of the use of a consistent set of modules, and to aid in debugging, rather than constituting a design methodology.

8.3 Debugging tools

People who believe that better language design and programmer training will eliminate the need for debugging are in for a rude shock. Even if improvements in the fields mentioned will have a beneficial effect, it is important to remember that the hardware involved in embedded systems is very complex and unlikely to work without error when first tried out.

An integrated set of debugging tools should be message based. This permits the programmer to use a terminal on the host computer, where (s)he has developed the program, to down-line load several target microprocessors, put them all into execution, set appropriate breakpoints or traces, and examine the results, without needing to push buttons on each of the (potentially very remote) systems. Message based systems also permit the debugging dialogue to be controlled by a program as well as by a human operator. This possibility is not (yet) likely to be very helpful during the debugging phase itself, but it can provide a very useful test tool for verifying the correct operation of hardware with on-board software.

We should not, however, neglect the possibilities for more reliable programs that will occur when we use languages that provide task abstraction.

Finally there is a class of problems that arise when hardware is malfunctioning very infrequently. These problems are typically tackled today using sophisticated tools like logic state analyzers. It would be very useful if these devices could be really programmed and could take account of the symbolic names used in the programs loaded into the microprocessors.

9. SUMMARY

9.1 Language selection

At the risk of being accused of cowardice, I would express the opinion that there is no single 'best' language for programming embedded microprocessors. The costs of maintaining old programs and re-training old programmers are such that any new language has to offer outstanding advantages at close to zero cost before all the world will start using it. It may be possible, however, to indicate some types of user who may decide to program in one or other of the languages that we have discussed.

Although it has been rather widely used for the programming of microprocessors by technical staff, Pascal is starting to look rather old-fashioned, especially when pressed into service for multi-task applications. However, several excellent (but non-standard) compilers exist for microprocessor applications, and we should not expect their users to convert to other languages quickly.

Physicists transporting off-line programs down onto microprocessors will continue to use Fortran for many years to come. If the new Fortran-8X proposals are soon completed and accepted, then this statement is likely to remain true for a very long time indeed. The main danger to the supremacy of Fortran in physics will come if Ada really sweeps the world of embedded systems, since this might lead to pressure to standardise on Ada in order to enhance portability of programs between mainframes and microprocessors. This is crystal-gazing, however.

Engineers and programmers involved in building special purpose hardware boxes are increasingly using C. It allows them to do most of the things that they want fairly efficiently, compilers are widely available at reasonable cost, and their applications are not so complex that they will be quickly driven to Ada out of choice.

Some people carrying out operating systems style programming, and who appreciate the low-level possibilities of the language, allied to better readability than is possessed by C, are choosing Modula-2. In terms of its comprehensive nature we might expect these people to be candidates for a move to Ada, when it becomes available. What is difficult to predict at this moment is whether Ada will be cheap enough for wide acceptance, and whether the very serious investment required for training, and the doubts about the tasking primitive, will turn out to be serious issues or not. In my opinion there is no doubt that, even though the design was supervised by a committee, it is the most complete of all of the languages that we have discussed.

9.2 Integration

As I have pointed out quite a few times already, the nature of the language itself, although important, is not overwhelming. If the most advanced language is not integrated with the ROM monitor, so that the programmer can sit at a terminal on a well-equipped host computer, develop code, and then easily start, stop and debug that code on attached target microprocessors, then (s)he would be better off with a poorer language but better integration.

9.3 Multi-tasking

The battle between multi-tasking languages and real-time kernels is far from decided yet. I would expect that for 'black box' type applications the better possibilities for improved software engineering will work in favour of the language approach. However as the application increases in complexity and the need for continual addition to or modification of the incorporated tasks becomes more pressing, then the restrictions imposed by 'freezing' the application at compile time will become more serious. So I would be surprised if there are no powerful real-time kernels around in the year 2005.

9.4 Multi-processing

The field of multi-processing will see the same sort of battle between the language approach and the distributed kernel. Message-passing will obviously be the underlying mechanism used in both cases, and the incorporation of message-passing channels into the processor, as seen in the transputer, will surely become the norm.

10. PILS LANGUAGE DESIGN

The remainder of this paper is devoted to a presentation of the Portable Interactive Language System (PILS). This emphasis on PILS arises not because of any belief that PILS is the 'best' interactive language for microprocessors, but because I have been asked to spend some time discussing PILS, of which I am one of the authors. However, you will see that many of the topics that we have considered in the more general part of the paper recur here.

We decided to embark on the development of PILS in July 1981. Our main motivation arose because of the problems of porting interactive programs for testing new or faulty electronics equipment. In contrast to our work with compiled off-line programs, where we could build up libraries to go with programs written in Fortran-77, there were no *de facto* standards for interactive languages, and, what was even more worrying, there looked to be little chance of major progress in this field.

10.1 The drawbacks of BASIC

Before starting on the drawbacks, it would only be fair to recognise the immense impact that Basic has had on computing in general, and the teaching of computing in particular. However, for our work with electronics testing we could see the following set of problems :-

- Basic was poorly standardised. The only ANSI definition existing at the time was Minimal Basic, and this was, indeed, very minimal. For the reasons outlined below it was inadequate for our job, and we can assume that it was inadequate for many people, since it never really caught on as a language in its own right. Many manufacturers, on the other hand, had more powerful versions of Basic, addressing some subset of our own concerns, but neither addressing all of our worries, not dealing with the same problems in the same way as their competitors.
- Basic did not define subprograms with symbolic parameters, and this made it very difficult to write modular programs. The related lack of any standard way of calling libraries was also a serious problem.
- Basic also showed very poor program structuring, with the FOR-NEXT loop being the principal tool available. We certainly felt the need for modern control constructs and, ideally, the possibility to develop programs using independent modules.
- Finally the data types offered by the language, namely character string and numeric (floating point), did not match well with our normal need to treat 16- and 32-bit hardware registers.

I would like to re-emphasise that there were, and are, Basics available that address many of these issues, but the problem that we perceived was that we could not obtain portability of interactive programs written in Basic, because of the lack of standardisation.

10.2 Names

In PILS identifiers can be of any length. For practical purposes identifiers of 20 characters are probably adequate for most people. The theoretical limit in PILS is something over 100 characters, and is determined in the first instance by the need to fit each statement on a line of less than 150 characters.

10.3 Types and typing

Six data types and their associated literals are supported. These types are 16- and 32-bit integers, 16- and 32-bit words (unsigned integers), reals and character strings. It is easy to switch between implicit typing for 'quick and dirty' programming and full explicit typing for production quality software. Either Fortran or Basic style implicit typing is available.

Character strings are dynamic, of length 0 to 255, and there are related string operators (concatenation and search) and string functions, such as length determination, substring extraction, case conversion, and date and time determination.

10.4 Based numbers

The use of octal or hexadecimal numbers is very important when dealing with the control registers of electronics equipment. PILS supports numbers in bases 2,8,10 (default) and 16. When prompting for input the programmer can modify the base that will be used as default, and this will be indicated to the user in the prompt.

10.5 Arrays

Multidimensional arrays are supported, and when these arrays are defined inside subprograms they are dynamic, in the sense that the space required is allocated at the time of the call. The lower bound of an array may either be defined explicitly, or can be defaulted to either 0 or 1, as preferred.

10.6 Subroutine structure

Users may define named subroutines and functions with named parameters. The parameters may be of three types, input, output and input/output and there is full checking that the declarations are consistent with any calls.

10.7 Library routines

Access to non-PILS library routines is supported. At present this support assumes that the library routine conforms to FORTRAN parameter passing conventions.

10.8 MODULE structure

A module structure is supported to simplify the writing of PILS programs by more than one person. In PILS the module is the unit of editing and compilation. Although several modules can be LOADED into memory at any time, only one of these modules (frequently the main program module) can be in the workspace where it can be edited. In general terms modules can be used to make constants, subroutines and functions globally available to other modules. For example, many CERN programs used to test Fastbus-based equipment will first LOAD a module of Fastbus declarations which defines the names and parameters of all of the NIM/ESONE Fastbus routines, and their related constants, including error numbers.

10.9 Control structures

Labels may be symbolic, and for reasons of compatibility with Minimal Basic GOTOs are permitted. Complex program structures such as subroutines, functions or IF-ENDIF pairs may be labelled at both start and end, and PILS will check for the consistency of this labelling. The full FORTRAN-77 IF-THEN-ELSEIF...ELSEIF-ELSE-ENDIF control structure is available, together with a LOOP-EXITIF-ENDLOOP control structure. As from the release of PILS 2.0 a multi-way SELECT-CASE structure is also supported.

10.10 File operations

Character input/output to files is supported.

10.11 Events and error recovery

Although the syntax for event control has been defined ever since 1981, and even partially implemented, it has never been released. We hope that we will soon find time to complete this area.

Since the release of PILS 2.0 it has been possible to recover from run-time error conditions by use of the RECOVER, SIGNAL and RESTART statements. At the level of a subroutine, function or mod-

ule, RECOVER defines a block of code that will be entered whenever a run-time error is detected, SIGNAL permits the error to be handled at a 'higher' level, and RESTART enables the normal program flow to be resumed. Incorrectly handled errors cause execution to be terminated with an appropriate message.

11. PILS COMMAND INTERFACE

When PILS completes its internal initialisation it will issue the Command? prompt to the user. In reply the user either can input a PILS statement for 'immediate' execution, or can input a line number together with a PILS statement, or can issue a PILS command. 'Immediate' statements can include PRINT statements to print the value of any active variables, or statements to call subroutines, for example to perform a Camac or Fastbus transaction. When inputting a long program it is normal to use the COLLECT command which prompts the user for the line numbers.

11.1 General commands

General commands are HELP, RUN, CHECK, SET, SHOW, STEP, CLEAR, EXIT and QUIT

11.2 Editor commands

Editor commands are LIST, POSITION, FIND, SPOOL, AFTER, BEFORE, COLLECT, DELETE, UNDELETE, REPLACE, CHANGE, COPY, MOVE, NUMBER and EDIT. While the meaning of most of these commands is fairly self-evident, it is perhaps worth mentioning a few features that are not so common. LIST <name> will first provide a listing of all definitions of that name, followed by all references. UNDELETE normally permits the programmer to recover interactively any lines that have been removed from the active file since the last time that it was consolidated.

PILS programs that have been accepted as input are not stored in their original text form, but rather in a processed PILS-code that permits complete regeneration. Advantage is taken of this fact to carry out some transformations on the text. The case of PILS input is normally ignored, and when redisplayed all user-defined names are in lower case and all reserved words are in upper case. Furthermore most redundant blank space is ignored and automatic indentation of control structures is supplied by PILS itself. Finally several constructs are accepted on input, mainly for compatibility with existing languages, that are never re-displayed. For example redundant LET or CALL keywords are accepted so that Basic code can be USED directly, but these keywords will then disappear.

EDIT provides in-line editing based on the use of control characters. Although we are not very happy with this approach, it was dictated by the set of terminals initially available on our target systems. We would hope to provide some better facilities when effort can be made available.

11.3 File manipulation commands

File manipulation commands are USE, OLD, MERGE, SAVE, EXTEND, SCRATCH, RENAME and DIRECTORY

11.4 Commands to control execution and debugging

The ease with which the program can be modified and then restarted has always been one of the major attractions of interactive languages, such as Basic, when compared to the relatively long and complex edit, compile, link and rerun sequence of the classical compiled languages. In PILS there is a further improvement when debugging, because of the major checking that is performed to ensure that, as far as possible, the program is self-consistent with respect to program structures and data types before execution is attempted. Extra assistance is given through commands such as SET TRACE which turns on full tracing of all stores and transfers of control, SET TRACE <id> which provides selective tracing, and SET or SHOW PROFILE which provides information on the frequency with which lines of PILS code have been executed. The SET BREAK command allows the user to set breakpoints in a selective manner, and when a breakpoint is encountered 'immediate' mode PRINT statements allow the program state to be examined. The STEP command may be issued at any time and forces the PILS program to execute one line at a time.

12. PILS IMPLEMENTATION

12.1 Choice of the implementation language

PILS is implemented in a 'lowest common denominator' subset of Pascal that has proved to be highly transportable. When considered against the Pascal standard the only 'corner cutting' is the use of variant records to simultaneously manipulate the area of memory that holds the 'PILS-machine' as 8-, 16- and 32-bit storage (See Section 6.3.11).

12.2 Transportable source code management

In order to deal with the slightly different conventions required for separate compilation under different compilers we always pass the master copy of PILS through an editor prior to any compilation, removing the few lines that are not required for that machine. We normally work with PILS source divided into about fifteen modules. The division has been made in such a way that PILS can be overlaid on those machines, such as the Nord-100, where it is necessary.

12.3 PILS/Host Interface

In order to port PILS onto widely different computers we have defined a PILS-Host Interface consisting of five layers, which we refer to as the PHI. The zero level of the PHI is concerned with bit-handling and some other small utilities not available in standard Pascal, the first level with the terminal interface, the second level with the interface to the file system, the third level with handling traps and interrupts and the fourth level with non-PILS libraries.

12.4 Ports

When porting PILS to a new computer it is necessary to understand the separate compilation requirements of the compiler, and if necessary to modify the editor input to produce the required code; to define the size and layout of the 'PILS machine', which is held as a set of constants in a separate file; and to implement the PHI. The PHI can be implemented in phases, as required. We are aware of essentially complete PILS implementations on the following systems :-

- Vax under VMS
- Vax under Unix (4.2BSD)

- Norsk Data ND-100 under Sintran
- Apple Lisa under Pascal Workshop
- M68000 running CERN Cross software with MoniCa
- M68000 running CP/M
- HP 9816 (M68000-based workstation) running UCSD Pascal

We believe that a port is under way for the PDP-11 running under RSX-11M Plus.

12.5 The checker

In order to preserve runtime efficiency, as much of the checking of the consistency of a PILS program as possible is carried out in a Check pass which is separate from the actual running of the program. It is during this pass that, for example, any type conversions that may be required are resolved, and any inconsistent program structures are indicated. The Check pass is normally invoked implicitly whenever a RUN command is issued and the program has been modified since the previous RUN command was issued. It can, however, be invoked explicitly by the CHECK command if the user so wishes.

12.6 The emulator

The PILS emulator takes PILS programs that have passed through the checker and 'executes' them by emulating the 'PILS-instructions' that it finds on the computer where it is running. This emulation is sometimes simple or even trivial, as in sequences such as load <name1>, add <name2>, store <name3>, and sometimes more complex, as with calls of PILS subroutines. Subroutine calls require that PILS manages the 'PILS stack', which holds the parameters (either directly or as pointers) and all local variables, which might be of different types or even arrays.

String variables, and the names used in the PILS program, are managed in a separate part of the 'PILS machine'. Garbage collection techniques are required when there is no more space available.

Remember that the PILS emulator is a Pascal program.

12.7 The compiler

Under PILS 2.0 running under Vax/VMS and Vax/Unix it is possible to compile a PILS program. A compiler for the M68000 family implementations is under field test at CERN at present. The compiler compiles from 'checked' PILS-code into memory. The speed of compilation is typically 1000 lines/minute while timesharing on a Vax-11/750, and the speed of the compiled programs is very close to the speed of Fortran programs. The normal range is from the same speed to some 20% slower than Fortran. On the other hand PILS dynamic strings and input/output are handled by exactly the same code as when the program is emulated. We anticipate that the major interest for the compiler will be to speed up the test programs on those occasions when the user wants to follow signals on an oscilloscope. At present some of the flexibility and ease of debugging is lost when the program is compiled, although the fact that a compiled module can co-exist with an emulated main program helps to attenuate this effect.

13. BIBLIOGRAPHY

The only book that I have seen that covers roughly the same area as this paper is 'Programming Embedded Microprocessors - A High-level Language Solution' [38]. This is a clearly written review of the subject, aimed at people who want to select a language and software development environment for programming embedded microprocessors. My main reservation would be that it devotes too much attention to PL/M, PLZ and MPL, which are manufacturer specific medium level languages that are, in my opinion, of little general interest. There are also chapters on Coral-66 and RTL/2, which have been widely used in Britain, but nothing on either Modula-2 or Ada. To be fair to the author, the book is very down-to-earth and neither Modula-2 nor Ada were in any sense 'available' at the time that he was writing.

For an excellent overview of programming languages in general (with no particular emphasis on the needs of embedded systems) I would recommend the Second Edition of Horowitz's 'Fundamentals of Programming Languages' [39]. This also contains a very thorough set of references, some 12 pages long.

For people who wish to pursue the subject in more depth there are a number of journals which publish relevant papers, including Software-Practice and Experience, ACM Sigplan Notices and IEEE Transactions on Software Engineering. However it has to be admitted that the good material is rather widely scattered in a combination of journals, books, reference manuals and conference proceedings. This is especially true when searching for material with a specific orientation, such as embedded systems.

14. ACKNOWLEDGEMENTS

Many of my colleagues at CERN have helped me in discussions about the theory and practice of computer languages. I am especially indebted to Julian Blake, David Foster and Horst von Eicken, who developed the CERN cross software, and have been patient with my questions, and to Michael Metcalf. Professor R.D. Russell (University of New Hampshire) is the principal designer and implementor of PILS. Louis Tremblet and Bernard Pierre have also designed and implemented parts of the language. Brian Carpenter, Les Robertson and Professor C.T. Zahn (Pace University) were kind enough to read the whole manuscript and provide me with very valuable comments. It goes without saying that any remaining mistakes (and I would be surprised if there are none) are my own responsibility.

REFERENCES

- [1] DATA I/O. ABEL 1.1. October 1984. Manual Assembly 981-0120-003 REV C.
- [2] Apollo Computer Inc. Getting Started with your DOMAIN System. Order Number 002348.
- [3] Dave Cane, Steve Mullen. Triple-Bus architecture gains speed, versatility. In Computer Design, February 1984.
- [4] Sun Microsystems Inc. Beginners Guide to the Sun Workstation. January 1984. Part Number 800-1111-01 Rev. D.
- [5] Motorola Inc. MC68020 32-bit Microprocessor User's Manual. Prentice-Hall. ISBN 0-13-541467-9.
- [6] Robert N. Noyce and Marcian E. Hoff, Jr. A History of Microprocessor Development at Intel. February 1981. IEEE Micro 1(1).
- [7] James J. Farrell III. The Advancing Technology of Motorola's Microprocessors and Microcomputers. October 1984. IEEE Micro 4(5).
- [8] Apple Computer Inc. Macintosh Manual. Product # M1500.

- [9] Larry D. Hemmerich, Peter Grohmann. Secondary storage devices look to the long term. In Computer Design, October 1 1984.
- [10] Iann Barron et al. Transputer does 5 or more MIPS even when not used in parallel. November 1983. Electronics 56(23).
- [11] Horst von Eicken. MoniCa Symbolic Debugger. Prerelease Information - May 1985. CERN/DD/85.
- [12] N.E. Wirth. Schemes for Multiprogramming and their Implementation in Modula-2. Revisions and Amendments to Modula-2. ETH Zürich Informatics Institute June 1984.
- [13] D.R. Myers. Programming for ERASME, a Film Data Processing System. September 1977. CERN/EF 77-5.
- [14] Motorola Inc. M68000 Family Real-Time Multitasking Software Users Manual (9th. Edition). March 1985. Order Number M68KRMS68K/D9.
- [15] Software Components Group, Santa Clara, California. pSOS Reference Manual.
- [16] Industrial Programming Inc., Jericho, New Jersey. MTOS Reference Manual.
- [17] Hunter and Ready, Palo Alto, California. VRTX Reference Manual.
- [18] N.V. Philips, Eindhoven, Netherlands. DRM Reference Manual.
- [19] D.O. Williams. Structured Transfer of Control. October 1984. ACM Sigplan Notices 19(10).
- [20] K. Jensen, N.E. Wirth. PASCAL User Manual and Report. Springer-Verlag. ISBN 3-540-90144-2.
- [21] ANSI/ISA-S61.1. Industrial Computer System Fortran Procedures for Executive Functions, Process Input/Output, and Bit Manipulation.
- [22] Digital Equipment Corp. VAX Procedure Calling and Condition Handling Standard. Version 9.4 (13 March 1984). In Introduction to VAX/VMS System Routines. Order number AA-Z500A-TE. Also in VAX Architecture Handbook.
- [23] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice-Hall Software Series 1978. ISBN 0-13-1101163-3.
- [24] N.E. Wirth. Programming in Modula-2. Springer-Verlag 1983. ISBN 0-387-12206-0.
- [25] British Standards Institute. Specification for the Computer Programming Language Pascal. BS6192:1982.
- [26] Eric S. Roberts et al. Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors. 1981. Software-Practice and Experience 11, 1019-1051.
- [27] A.J. Wellings et al. A Problem with Ada and Resource Allocation. October 1983. ACM Ada Letters III-4.112-124.
- [28] P.H. Welch. Structured Tasking in Ada? July-August 1985. ACM Ada Letters V.1.17-31.
- [29] Per Brinch Hansen. The Architecture of Concurrent Programs. 1977. Prentice Hall. ISBN 0-13-044628-9
- [30] Inmos Limited. Occam Programming Manual. Prentice-Hall International 1984. ISBN 0-13-629296-8
- [31] International Standards Organisation. Graphical Kernel System (GKS). July 1985. International Standard ISO/IS 7942.
- [32] R. Brun et al. HBOOK Users Guide. Revised June 1984. CERN-Data handling Division DD/US/72.
- [33] D. Burckhart et al. ESONE/NIM Standard Camac Subroutines. May 1984. CERN/DD/OC/80-4.
- [34] E.M. Rimmer et al. CERN Fastbus Software. August 1984. CERN/DD/OC.
- [35] J.D. Blake. Use of Microprocessor Cross Software under VAX VMS. April 1985. CERN/DD/SW - LM/T9.
- [36] Peter Freeman and Anthony I. Wasserman. Tutorial on Software Design Techniques. Third Edition. IEEE 1980. Available from IEEE Service Centre, 445 Hoes Lane, Piscataway, NJ 08854, USA.
- [37] 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.

-
- [38] R.J. Foulger. Programming Embedded Microprocessors. A High level Language Solution. 1982. NCC Publications. ISBN 0-85012-336-4.
 - [39] Ellis Horowitz. Fundamentals of Programming Languages. Second Edition. 1984. Springer-Verlag. ISBN 3-540-12944-8.