INTERPRETIVE KERNELS FOR MICROCOMPUTER SOFTWARE

B.R.Gaines*

The initial main applications area for microcomputers has been in small, mass-produced systems where they replace hardwired, random logic. These present few problems of 'software' development because the 'programs' required are small and fixed. However, it is clear that the technology has now reached the state where the 'microcomputer' is in every sense a 'computer' with all the capabilities of much larger and more expensive machines. Increasingly many applications are looking towards its programmability, and continuing, in-use re-programmability. This is generating requirements for a level of software support not generally provided with microcomputers. More importantly it is forcing organizations with long experience of hardware manufacture to move into the area of software and systems development, maintenance and support. This paper is concerned with software engineering techniques that allow the same discipline of modularity, documentation, quality control, etc., that has previously been imposed on hardware to be applied to software.

1 INTRODUCTION

The role of software in microprocessor-based (µproc-based) systems is as yet unclear. In the majority of current applications the use of a µproc is justified on the grounds of total engineering cost compared with other implementations. That is, the 'computer' does not have to provide additional features of its own in order to be cost-effective as a replacement for a conventional hardware implementation. In these applications it may be regarded as a collection of logic elements whose functions are established by a ROM, and the tools necessary to 'program' that ROM (assembler, debugger) are just developmental aids that play no part in the final system. The manufacture of such µproc-based systems should be amenable to the design, development, production, marketing, maintenance and customer-support techniques and disciplines that are already well-established for the previous generation of 'hardware' systems.

And yet 'software' problems are already significant in what appear to be continuations of previous product lines, e.g. instrument or display terminals that have been re-engineered for lower cost production around a µproc but are otherwise unchanged. In general, the problems are arising because the use of a 'computer' is expected both by manufacturer and by customer to provide a new level of flexibility that was previously impossible. The overall

---

*Man-Machine Systems Laboratory, Department of Electrical Engineering, University of Essex, Colchester, UK.

system has become programmable so that a product can serve a greater variety of applications, is readily modified to new requirements, and this flexibility can even be put in the hands of the end-user. What has not yet been adequately discussed, defined and understood is the cost of this new-found flexibility - a cost which is substantial in product and customer support rather than in development, and which is yet inadequately controlled.

In discussing 'software' problems and requirements in more detail it is useful to summarize the above discussion in terms of 4 levels of use of microprocessors:

(a) The processor replaces hardware elements. Its programming is equivalent to computer-aided design (CAD) of these elements. This affects product development only and the fact that it is itself 'computer-based' is irrelevant to all later aspects of marketing, application and support. The advantage of the µproc is that it replaces boards of random logic simplifying production and maintenance. There should be no associated disadvantages, particularly if the design group have already been using CAD packages for logic design, board layout, etc.

(b) Basing a product range on µprocs allows new product development to be largely an extension of the range through software modification rather than new hardware production. This probably entails changes in production techniques with 'software modules' being treated in the same way as hardware modules in terms of documentation, testing, etc., but does not affect marketing or the end-user. The additional advantage of the µproc is that it allows total hardware re-engineering to be avoided in what may be very substantial product changes. The disadvantage is that production procedures have to be introduced for installing and checking software modules, but this is a reasonably straightforward problem that can be treated using the well-established methodologies for hardware production.

(c) The customization and field upgrading of individual products is made part of their specification and a key factor in their marketing. Technically this is, in a sense, 'already available'. If new product ranges involve 'only a change in software' why should not the range become a continuum with each customer selecting the facilities appropriate to his application. The advantages are clear on the marketing side - one of the biggest attractions of computers has always been that they can cater for individual requirements and that, if these turn out to have been misconceived, the computer can always be reprogrammed for something else ! However, the new problems that now arise are substantial - documentation and customer support has also to be customized and instead of having a standard product range one is now in the 'systems' business. This is a viable proposition and such businesses can be profitably managed, but they are not simple extensions of (a) and (b). They have an entirely different cost structure involving a large element of uncertainty, and the final product cost is largely in terms of people not hardware.

(d) The ultimate level of exploitation of the computer-based system is to put its programming in the hands of the customer, i.e. to market the product as one which may be tailored to individual requirements and where the user himself may perform this tailoring. Technically this may be regarded as an extension of switches and dials on the front panel, but the magnitude of the extension can be such that a major qualitative change takes place. Clearly the level of support the user requires is substantial - a far more complex instrument has to be explained to him. The major change in most applications is that the instrument becomes capable of complex, time-dependent procedures. However, there has also been a change in the customer-supplier relationship that has to be recognized. The end-user now has a realtively simple piece of equipment whose complexity and problems lie in the way he programs it, whereas previously they lay in the way that the manufacturer had 'programmed' it. The level and types of responsibility and support that the user expects of the supplier have got to change drastically if the cost-structure is to remain the same. The supplier no longer has 'system responsibility' for his product. He does not know enough about the way in which it is being used to foresee and warn of all possible problems, and so on. Clearly a new relationship can be established, but it is in doing this that many of the current problems are arising.

I must apologize in what is intended to be a technical paper on certain aspects of microcomputer software for dwelling so long on the 'sociology' of the use of such devices. However, the technology, both hardware and software, is itself simple, and it is its commercial application and control that is difficult. The use of interpretive 'high-level languages' to be described in this paper is an important technique of software engineering that bears on problems at levels (b), (c) and (d), particularly these last two. If I had claimed at the outset that one major advantage of the technique was to <u>control and restrict the flexibility of computer-based systems</u>, it might have seemed ridiculous - I hope the reasons for doing so are now more apparent. The other advantage is to <u>ease the programming and documentation of software</u>, a more readily appreciated virtue but again one that is closely related to the problems outlined. Key features of the approach are to:

(i) Impose functional modularity on software - a production technique well-established for hardware - a module is something that does a well-defined, and usually comparatively simple, task and can be tested thoroughly and used safely according to known rules;

(ii) Allow systems to be built up from modules directly from a specification in a well-defined and readily understood problem-orientated language.

<u>Minicomputers and microcomputers</u> The techniques described in the following sections were originally developed for minicomputers. However the current generations of µprocs provide generally better instruction sets than the previous generation of minis. Notably index registers and byte-addressing are provided which machines such as the PDP8 lacked and had to emulate through subroutines. I shall not link the instructions and addressing structures in the examples to particular machines, but none of them tax the facilities of current µprocs.

One significant difference between minicomputer and µproc applications is that storage utilization has become less important in many mini applications because the costs of larger stores have fallen so dramatically. In general it is still significant for many µproc applications where economies of size do not apply to the store. Hence compact programs are desirable and I shall illustrate how these may be achieved. Additionally, backing stores are less often available on µproc systems so that program entry is a problem, and again I shall illustrate how this may be minimized.

The techniques described are all well-proven, having been used in a wide variety of commercial, medical, industrial and scientific applications (Facey and Gaines (1), Gaines and Facey (2), Gaines et al (3,4), Green and Guest (5), Moore (6), Rather and Moore (7), Baltzer et al (8)). Although involving high-level languages, they are not expensive in machine resources (the initial development of our system was on a time-shared PDP8 allowing only 4K 12-bit words per user, and an interpreter for a BASIC-like language with integer arithmetic and extensive string-handling was fitted in 2.7K allowing 1.3K per user program overlay which proved ample for a range of data-processing and record-keeping applications (2), Kennedy and Facey (9)) and do not necessarily involve substantial speed losses compared with assembly code.

2 VIRTUAL MACHINES AND MODULARITY

One of the most useful concepts to have been developed in the computer science literature in recent years is that of a *virtual machine* (Gaines (10), Goldberg (11)). Broadly interpreted it recognizes that a computer with certain software in it has become *another computer* with its own characteristics. Anyone who has transferred from a basic machine to one with an operating system, or between different operating systems, will be aware of the distinction - the machine changes in character and power. Anyone who has used a *library* of standard subroutines will have noticed that the routines themselves may be regarded as *instructions for a more powerful machine*.

The concepts of *modularity* (Dennis (12)) and virtual machines are closely related. We attempt to split software into modules each of which has a clearly defined function and is relatively independent of other modules. Generally the modules are linked together to form a system by a series of subroutine calls. These calls may alternatively be regarded as instructions for a new computer, the virtual machine we have created by developing the modules.

Once one takes this viewpoint certain very useful related concepts may be developed. The differences between a computer designer, a micro-programmer, a programmer, a system designer, etc., become less apparent - we are all both computer and system designers !
In practical terms it means that much of the work and literature on computer architecture and run-time systems for languages is very relevant to application programming. There are few designers of IBM 360's, Burroughs B1700's, etc., but those design studies and textbooks based on them are relevant to a far wider audience, e.g. data-descriptors and tagging (Gaines et al (13), Feustel (14)) are useful in interpreters. Similarly, studies of FORTRAN, ALGOL, SNOBOL, etc., support software suggests many techniques that are useful in computer-based systems not using the entire construction of these languages.

Secondly, the virtual machine construct is naturally hierarchical (10) - we can build another level of virtual machine by linking together some of the modules that exist at the lowest level into larger modules at the next level. Each level defines a new machine, a new *product*, and each level remains programmable in terms of the modules available at that level. Fig.1 illustrates a 5-level virtual machine hierarchy in which the '*development engineer*' sees a computer and designs subroutines for it, e.g. to control certain peripheral devices and to make certain calculation facilities available (e.g. data smoothing, floating point arithmetic, message communication

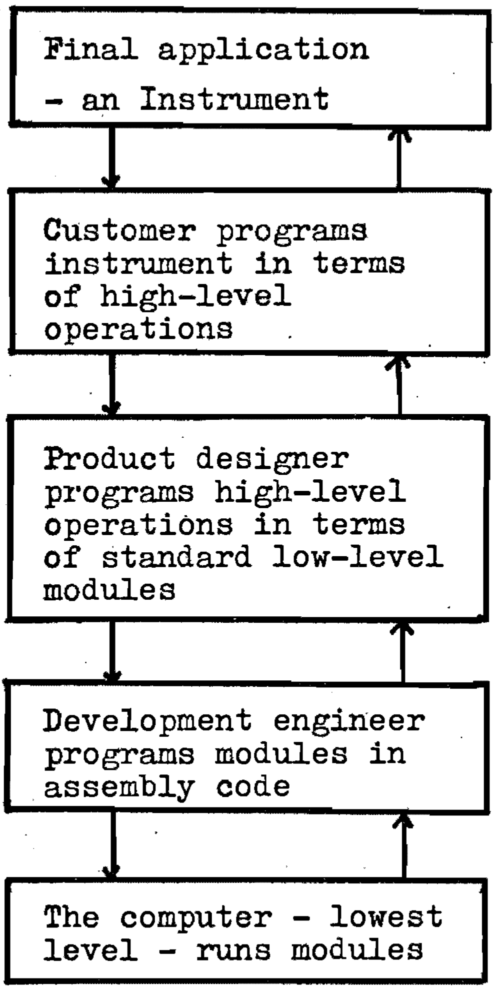

Figure 1 A 5-level virtual machine hierarchy

protocols, etc.); the '<u>product designer</u>' configures a system and writes routines using the modules provided by the development engineer to give the required control, data-logging and data-processing facilities for the system in terms of a 'problem-orientated language'; the customer develops a program in this language for his ultimate application, thus finally defining the top-level 'virtual machine', an instrument to the end-user.

The advantages of this approach are many-fold, but primarily: (A) At each level the task of the person responsible for the development is well-defined and reasonably circumscribed. He sees the virtual machine of the level below him and is responsible for constructing that of the level above him. The final applications programmer does <u>not</u> have to worry about details of the instruction set of the µproc at the lowest level, nor even of the operation of the language system at the next. He sees functions that make sense in terms of his problem area and in terms of the type of system he has purchased. Equally, in this illustration, neither does the 'product designer' have to concern himself with the details of the µproc and the software support of standard peripherals. He sees a library of routines and an operating system flexible enough to support a range of products, yet with most of the 'technical details' of control, timing, etc., already taken care of. (B) Because of the high degree of independence between levels, changes in technology at one level need not propagate beyond the level above. For example, if the µproc at the lowest level is replaced by a cheaper, faster one, then only the 'development engineer' need be affected. The product designer, customer and end-user do not need to modify their systems - they have only become cheaper and faster.

The approach to system design based on defining modules and then linking them into larger sub-systems, etc., is called a <u>bottom-up</u> approach - it is clearly appropriate to the development of a product range. The converse approach of commencing with an application and analysing it into sub-systems, etc., is called the <u>top-down</u> approach and is clearly appropriate to a systems division. Both types of approach are necessary in practice - the gap between µprocs and applications is such that a bottom-up approach has a long way to climb before it is useful, whereas a top-down approach has a long way to fall before it hits actual hardware. In the climb or fall there are too many pitfalls, diversions, and ranges of complex possibilities for adequate development disiplines to be exerted if the product requirements fall into categories (c) and (d) of section 1. The virtual machine approach illustrated in Figure 1 may be seen as a way of splitting the development into well-defined levels in each of which the bottom-up and top-down approaches meet and can be integrated together. In human terms, in particular, there need generally be only one person with design responsibility and authority for a given level. The overall system development has been split into well-defined, comprehensible, and manageable sub-tasks.

3 DESIGN AND IMPLEMENTATION OF VIRTUAL MACHINES

This paper is primarily concerned with the principles and implementation of virtual machines and I shall not consider the design in detail. However, there are certain aspects of the design that relate closely to the implementation, and these are primarily of a 'linguistic' nature. Whereas the actual functional modules that make up the machine are clearly dependent on the type and range of applications enviaged, the way in which their control, interconnection, etc., is specified is a more general human factors problem. It is possible to regard each module as a separate entity with its control specified in some specific way. However, with a wide variety of modules this imposes a memory burden on the user, or programmer, who has to remember not only what a module does but how its use is specified. If, for example, WIM and WAM are the names of two data-acquisition modules, each requiring a source and desination plus two numerical parameters, then specifying them respectively by,

```
LDA PARAM1     /one param in acc
LDX PARAM2     /one param in index reg
JMS WIM        /call WIM subroutine
SOURCE         /source routine call address
DEST           /destination routine call address

WAM(SOURCE,DEST,PARAM1-expression,PARAM2-expression)
```

is confusing to say the least ! Such an example is exagerated but users of even well-established languages such as FORTRAN and BASIC will have noticed anomalies that make programming more difficult. Such anomalies tend to be far more prevalent in specialist software packages.

Thus, consistency and uniformity is the way in which modules are specified and controlled is highly desirable - if the specification of a parameter can be an arithmetic expression in one case then this should be possible in all cases, etc. Such considerations are important at all levels of virtual machine and I have discussed them for computer design (13) and man-computer dialogue design (2) elsewhere. There is one further design consideration worth emphasizing here because it highlights one of the defects of assembly code programming, and that is the way in which the structure and facilities of a virtual machine should guide the programmer in its use.

We tend to think of the negative aspects of constraints such as those imposed at each level of the virtual machine hierarchy of Figure 1 - the customer is prevented from corrupting the software, slowing down other users, misusing certain peripherals, etc. However, much of the freedom lost is not only unnecessary but also positively misleading because it is the freedom to do one thing in a thousand different ways. This is particularly so at assembly code level where even simple routines may be coded in innumerable ways. Such flexibility may seem attractive in catering for all possible styles and requirements. However, it calls for high information-content decision making and high information-content documentation at every stage, both sources of problems and costs. The best virtual machine is one in which for each task that is natural to it there is one, and only one, way of programming it and that way is obvious - the structure of the machine should so guide the programmer that a statement of the task implies how it should be programmed.

The virtual machine concept in itself gives little information as to its implementation. It is technically simple to write a software package as a set of modules, sub-programs or subroutines, that are linked together by GOTO's or procedure CALL's. This is good practice at all levels and every machine has its calling mechanisms to enable this to be done. There is wide variety in the method by which parameters are passed to the sub-program and results returned but, even at assembly code level, information flow between sub-programs can be standardized so that modules may be interfaced freely provided certain conventions are obeyed. The basic assembly language rarely provides a rich enough syntax in itself to make the information flow linguistically natural. However, the use of a macro-generator (Brown (15)) before the assembler can overcome this, replacing:

```
LDA X
LDB Y          with  BINGO(X,Y)  or even  FROM X BINGO TO Y .
JMS BINGO
```

Thus, the virtues of clarity, modularity, etc., are not to be claimed by any one language or technique alone. However, certain approaches do make them easier to attain and easier to impose.

The system that actually causes the instructions to a virtual machine to be executed is called its interpreter. This itself will generally be programmed in the instructions of a lower level machine, down to the actual computer instructions being interpreted by a micro-program. The kernel of an interpreter is the general logic associated with fetching and decoding instructions, passing parameters, etc., as opposed to executing specific operations. In the following sections I shall describe some simple and compact interpreter kernels that have been used successfully in commercial applications and are well-suited to µprocs. The particular systems also have the advantages of overcoming some of the addressing limitations of µprocs and of being interactively programmable, in that programs may be entered at a terminal, executed, interupted, modified, and execution continued. Such interactive capability is particularly desirable at the higher, end-user, levels.

4 STRUCTURE OF A BASYS INTERPRETER

BASYS (1,2) is a BASIC-like language (Schur (16)) that is in wide use for applications ranging from instrumentation and data-logging to financial dealing and medical record keeping. A BASYS program consists of a sequence of lines ordered by their, not necessarily consecutive, line numbers. A line consists of one or more statements separated by colons, and a statement consists of a meaningful key-word followed by an expression, or sequence of expressions, e.g.:

```
25 LET P=15
```

```
 37 PRINT 'P IS ' P
 50 DRIVE P+7 15 K :[SET UP MOTOR
 52 IF K=0 :LET Y=P/2 :GOTO 100
 54 PRINT 'PROBLEM ON MOTOR 15' :GOTO 2000+10*K
100 LOG P+5 4 U :GOTO 137 :[GET DATA FROM A-D 4
```

and so on. The data types in BASYS include variable-length integers, arrays, variable-length character strings, and reference variables. The normal range of arithmetic operations, and an exceptionally powerful range of string-processing operations, are included in the general structure but, in addition, provision is made for the ready addition of special processors such as DRIVE and LOG above.

BASYS itself is extremely interactive and easy to use and the programs are particularly clear because of the expressive key-words and two-dimensional form of the language. The evaluation of arithmetic expressions is slow compared with machine code, but this does not matter because fast machine code modules are readily added as new processors when required. These new modules are activated by a key-word and parameter list like the existent processors, and hence integrate simply and naturally with the existing language. In its implementation BASYS is essentially a string interpreter and all the important routines may be viewed as processors that transform strings.

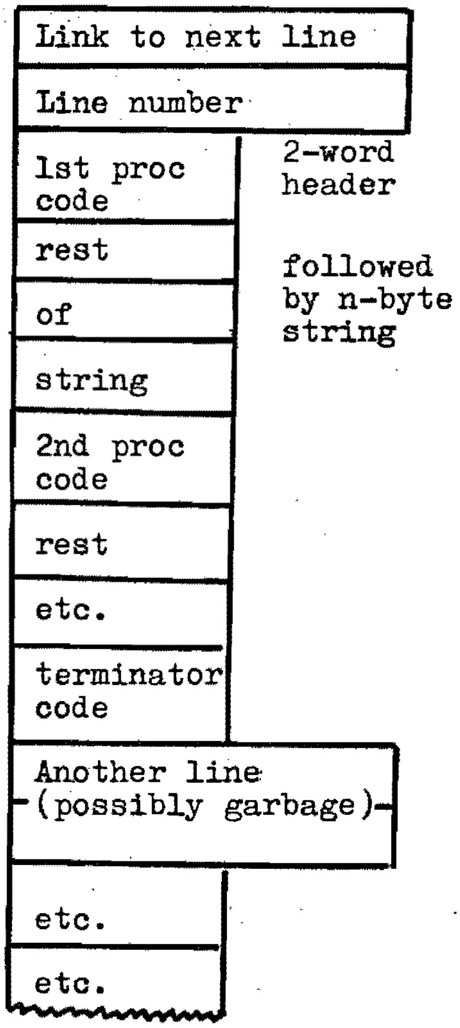

Figure 2 Program line structure in BASYS

Figure 2 shows the structure of program lines in BASYS. They are stored as a linked list commencing with the lowest line number, and the first word in the 2-word header is a link to the next line in the list. The next header word is the line number itself. There follows a variable length string consisting of single-byte codes for each processor name followed by the actual parameter strings. A termination code indicates the end of the string and there follows the header of another line (not necessarily the next one in line number sequence).

Figure 3 shows the overall storage structure for BASYS program and data. Both are dynamic structures whose size varies at run-time (character strings are stored as 'program lines') and share a single freespace area.

The interpreter consists of the following parts:

(1) Routines for storing, inserting, deleting and garbage-collecting strings, and maintaining the program statements in the correct order,

(2) A main control loop which determines which statement is to be processed, picks up the processor codes, and transfers control to the corresponding processor;

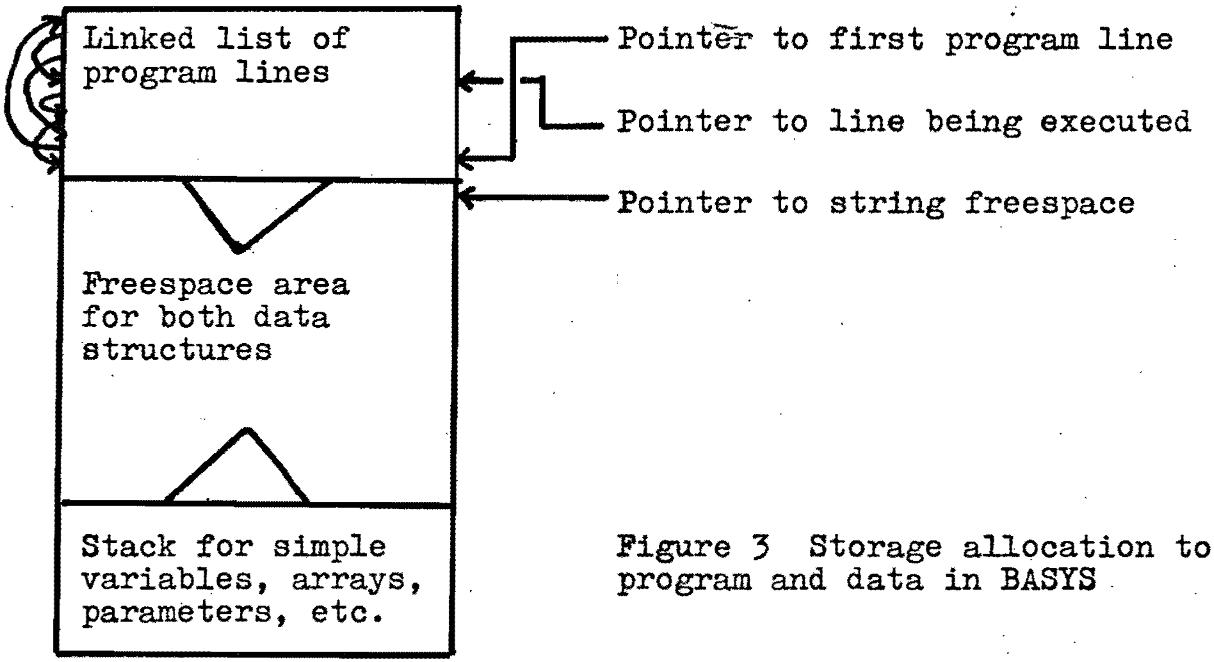

Figure 3 Storage allocation to program and data in BASYS

(3) Processors corresponding to each command-word/processor-code;

(4) A set of general procedures which are called by the command processors and which do most of the work of evaluating and interpreting argument strings.

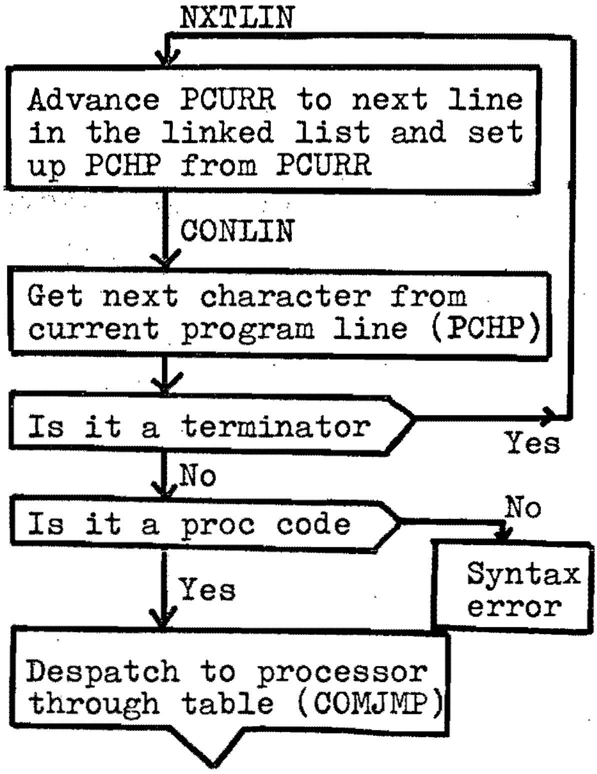

Figure 4 Main control loop

Figure 4 shows the flow of the main control loop. PCURR is a pointer to the current line being executed and it is set up at NXTLIN to point to the next line. PCHP is a pointer to the next character in the current program line and at CONLIN it is expected to point to a processor code. In an 8-bit byte machine these codes will typically have the top bit set to distinguish them from ASCII 7-bit characters. When a processor code in found it is used to transfer control to the appropriate processor through a table of processor entry points, COMJMP shown in Figure 5.

Most command processors themselves involve little code since they use general routines for expression evaluation. For example the command LET X=5 contains the command word 'LET' which becomes a single-character processor code followed by the string 'X=5'. When the interpreter finds the code it transfers to the LET processor which first calls a general ASSIGN routine that sets up a pointer to X, then a general arithmetic EVALUATE routine that returns the value 5, then a general data transfer routine that moves the value 5 to the locat- indicated by the pointer to X.

The IF processor applied to the same string would call the EVALUATE routine immediately to yield a value TRUE if X=5 and FALSE otherwise. It then exits to CONLIN to continue execution of the program line if the result is TRUE, but goes straight to NXTLIN if the result is FALSE, thus executing the conditional as required. This use of the two entry points in the main control loop to give conditional execution of the remainder of the line is used in many processors, e.g. an input/output process will return to CONLIN only if it is successful so that one can write -

```
DRIVE X 7 Y :PRINT 'DRIVE OK'
```

and the PRINT will only occur if the drive operates properly.

```
COMTYP  ASCII  'LET' <TERM>
        ASCII  'GOTO' <TERM>
        ASCII  'PRINT' <TERM>
        ASCII  'IF' <TERM>
        ASCII  'RUN' <TERM>
        ASCII  'STOP' <TERM>
        ASCII  'LIST' <TERM>
        ASCII  'DRIVE' <TERM>
        ASCII  'LOG' <TERM>
        ASCII  <TERM>

COMJMP  LET
        GOTO
        PRINT
        IF
        RUN
        STOP
        LIST
        DRIVE
        LOG
```

Figure 5 Command string table and processor despatch table in BASYS

Incorporating a new processor in BASYS is extremely simple since command names and entry addresses are held in two open-ended tables as shown in Figure 5. The special command DRIVE, for example, has been inserted by putting its name as a character string in the table COMTYP, and its entry point as an address at the corresponding position in the table COMJMP. When the editing phase of the interpreter encounters the string 'DRIVE' it encodes it as a single character processor code. When the main control loop of the interpreter encounters this code it transfers control to the entry point, DRIVE. The processor takes 3 parameters, two values and one address, and might look like:

```
DRIVE  JMS EVAL      /evaluate arith. expression - result in acc
       TAX           /put result in index register X
       JMS EVAL      /evaluate arith. expression - result in acc
       STA,X IOTAB   /send acc to address in X in IO table
       STA TEMP      /status information is returned in acc
       JMS ASSIGN    /get pointer in X to variable
       LDA TEMP      /get status back
       STA,X         /and store in location indicated by X
       JPZ CONLIN    /continue program line if transfer was OK
       JMP NXTLIN    /go to next line if transfer not OK
```

The preceeding discussion and examples illustrate the structure of the BASYS interpreter and the way in which it can be used to link machine code routines together under program control in a fairly high-level and readily comprehensible language. The kernel of this interpreter consists fundamentally of the control loop shown in Figure 4 but I would include as part of it the general arithmetic assignment and evaluation routines that are common to virtually all applications. Once this kernel has been written and thoroughly debugged it can be used a foundation for a wide variety of special systems into which new facilities are 'plugged' in the simple way shown. The kernel typically consists only of some 2K machine instructions and hence is readily transferred from machine to machine. Utility routines for listing and storing programs, etc., are actually written in BASYS as 'hidden' procedures. This trick of writing as much as possible of the non-real-time part of the interpreter in itself is widely used and saves much programming effort.

In the next section I will discuss a variation of the technique used in BASYS which enables the interpreter itself to run substantially faster at some cost in flexbility and size of code.

5 THREADED CODE TECHNIQUES

One of the simplest and most effective techniques for linking routines together and overcoming the program and addressing limitations of small computers is that of threaded code, originally described by Bell (17) as implementation of the run-time environment for PDP11 FORTRAN. A further development of it was used by Dewar (18) to support a fast, machine-independent SNOBOL compiler, and an extension of the type of technique forms the basis of FORTH (6,7), a very successful, fast, interactive language used in small astronomical computing systems.

The concept behind threaded code is extremely simple - it is to use a table of routine addresses to cause the actual hardware processor to 'thread' its way through the routines in the specified sequence. The left hand side of the code below shows a conventional sequence of subroutine calls and an example of a normal subroutine and return. The right hand side shows the same sequence effected by jumping to the address pointed at by the index register X, i.e. effectively load the program counter (PC) with the word pointed at by X and then increment X.

```
START  JMS ROUTA                 START  ROUTA
       JMS ROUTB                        ROUTB
       JMS ROUTA                        ROUTA
       JMS ROUTC                        ROUTC
       .........                        .....

                                 INTER  LDX [START    /get address
ROUTA  ---------                        JMP,X+        /jump to it
       ---------                                       with post-inc
       .........                 ROUTA  ---------
       RETURN                           ---------
                                        .........
                                        JMP,X+
```

The 'subroutines' themselves differ only in that they do not have a normal RETURN, but exit by transferring control to the next address pointed at by the (updated) X. Thus X itself may be regarded as a pseudo program counter (PCC) and the routine entry addresses as instruction codes for a virtual machine.

One important advantage of the technique is that the addresses in the subroutine calls on the left can occupy only part of the instruction whereas those in the control table on the right are full words (a similar consideration applies to the despatch table of BASYS in Fig.5). Either the subroutine call have to be double-length (as in PDP11) or they have a substantially shorter address scope than does a full word (probably expanded by transferring indirectly via a table, e.g. in page zero, that corresponds to our despatch table). The control table of threaded code is thus more compact in giving full access to the store. In many machines execution through a PCC in this way is not significantly slower than the overhead of the subroutine calls (on the PDP11 it is faster !).

If the subroutines require arguments then their addresses can also be imbedded in the control table. For example:

```
LOAD   /address of routine         LOAD  LDY,X+ /get address arg
ARG1   /address of argument        ............   and increment X
STORE  /another routine            etc.    JMP,X+ /exit
ARG2   /and argument
```

The routine LOAD picks up the address of its argument and advances the PCC past it ready for the next transfer. The technique described in (17) avoids the double word required for a sequence like LOAD ARG1. The PDP11 FORTRAN compiler actually generates a routine LARG1 that loads ARG1 to the operand stack. There is thus a load and a store routine for every operand, but since an operand will generally be used many times this uses less code than having multiple word entries in the control table.

Dewar (18) goes one step further and uses 'indirect threaded code' in which the table entries themselves point to the address of a routine, i.e. there is double indirection. The advantage is that the actual argument can be associated with the addresses of routines to load and store it. For example, a simple variable will have two pointers with it, one to a routine to load its value to the stack, and the other to a routine to set up its value from the stack. These routines will be common to all simple variables of a given type and themselves pick up their parameter from the calling address. The header block of an array would contain pointers to routines that use the number on the top of the stack to generate an offset into the array and then load or store to it. The technique has the advantage of even greater compactness of code and it allows a clean separation between program and data structures. The major advantage claimed by Dewar is that since the compiler itself generates only addresses of routines and data structures it can be completely machine independent. A related advantage is the way in which the selector/updater routines for data are associated with the data itself. This clearly allows operations such as input/output transfers to look like variable manipulation, and it allows for complex data structures of varying types including 'data-driven interupts' (19,20,21) and the advanced programming techniques of languages such as PLANNER (22), POP2 (23) and EL1 (24).

The difference between the implementation of BASYS previously described and that by threaded code is that the 1-byte processor codes become processor addresses and the parameter evaluations have also to be compiled into threaded code sequences. This makes dynamic program change more difficult (but still possible) since pointers to data structures have to be updated whereas their symbolic names did not. The advantage gained in using threaded code is one of speed because the interpreter is calling routines and accessing data far more directly in terms of actual addresses rather than codes and symbolic names.

In FORTH the linguistic structure for programming is made virtually the same as that of the threaded code so that the programmer himself has to 'compile' the algebraic form of a statement into its execution form, i.e. to say:

X LOAD Y LOAD + Z STORE rather than $Z = X + Y$

However, since the natural programming technique in FORTH is hierarchical with complex operations composed of simpler ones, themselves composed of simpler ones, etc., the effort of this compilation is not necessarily tedious. Indeed, since it corresponds to the actual sequence in which operations are carried out, it may even be more natural to someone who is hardware-orientated ! It is quite feasible, however, to provide automatic translation from the command on the right above to that on the left, and in GLUE (5) this is done to give a more conventional syntax than in FORTH.

In both FORTH and GLUE the translation from text to code is done by routines written in the language itself, and is incremental so that programs are compiled line by line and do not have to be available as a whole. In FORTH, because of the close relationship between the language and the code, the translation is reversible so that compiled sequences of code may be listed in their source form. Multiple indirection in the code is exploited in both languages so that a routine will consist of pointers to routines that themselves consist of pointers to routines, etc. This makes the building of virtual machine hierarchies in these languages both natural and efficient - the advantages of languages in which this can be done within a single consistent framework have been amply demonstrated by the complex systems that have been built, level by level, in LISP and POP2.

6 CONCLUSIONS

The two main points made in this paper are that -

(1) In the commercial exploitation of microcomputers software engineering has to be managed as rigorously as hardware engineering has been in the past;

(2) That certain software engineering techniques are particularly amenable to management and control in ways which naturally reflect the product structure and customer-supplier relationship.

Additionally certain technical concepts have been outlined and references given by which these techniques may actually be applied to microprocessors.

7 ACKNOWLEDGEMENTS

I would like to thank Peter Facey of this laboratory for many stimulating discussions of the topics in this paper. In addition I have benefited greatly from similar discussions with many colleagues in industry. In particular I have Lindsey Shaw of Instron to thank for drawing my attention to FORTH.

8 REFERENCES

1. Facey, P.V., and Gaines, B.R. (1973) Real-time system design under an emulator imbedded in a high-level language, Proc. BCS DATAFAIR 73, Nottingham, April, 285-291.

2. Gaines, B.R., and Facey, P.V. (1975) Some experience in interactive system development and application, Proc. IEEE, 63, 894-911.

3. Gaines, B.R., Facey, P.V., and Sams, J. (1974) An on-line fixed interest investment analysis and dealing system, Proc. Eur. Computing Congress (EUROCOMP 74), 155-169.
4. Gaines, B.R., Facey, P.V., and Sams, J. (1976) Minicomputers in security dealing, Computer, 9, Sept., to appear.
5. Green, T.R.G., and Guest, D.J. (1974) An easily-implemented language for computer control of complex experiments, Int. J. Man-Machine Studies, 6, 335-359.
6. Moore, C.H. (1974) FORTH: a new way to program a mini-computer, Astron. Astrophys. Suppl., 15, 497-511.
7. Rather, E.D., and Moore, C.H. (1976) FORTH high-level programming technique on microprocessors, Proc. ELECTRO '76, Boston, Mass., 23-4, 1-8.
8. Baltzer, P.K., Weisbecker, J.A., and Winder, R.O. (1976) Interpretive programming of small micro-processor-based systems, Proc. ELECTRO '76, Boston, Mass., 23-3, 1-4.
9. Kennedy, T.C.S., and Facey, P.V. (1973) Experience with a mini-computer-based hospital administration system, Int. J. Man-Machine Studies, 5, 237-266.
10. Gaines, B.R. (1975) Analogy categories, virtual machines and structured programming, in Muhlbacher, J. (ed.) GI-5 Jahrestagung, Lecture Notes in Computer Science, 34, Springer-Verlag, Berlin, 691-699.
11. Goldberg, R.P. (1974) Survey of virtual machine research, Computer, 7, 34-45.
12. Dennis, J.B. (1973) Modularity, in Bauer, F.L. (ed.) Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, 81, Springer-Verlag, Berlin, 128-182.
13. Gaines, B.R., Facey, P.V., Williamson, F.K., and Maine, J.A. (1974) Design objectives for a descriptor-organised minicomputer, Proc. Eur. Computing Congress (EUROCOMP 74), 29-45.
14. Feustel, E.A. (1973) On the advantages of a tagged architecture, IEEE Trans. Computers, C-22, 644-656.
15. Brown, P.J. (1974) Macro Processors, John Wiley, London.
16. Schur, L.D. (1973) Time-Shared Computer Languages, Addison-Wesley, Reading, Mass.
17. Bell, J.R. (1973) Threaded code, Comm.ACM, 16, 370-372.
18. Dewar, R.B.K. (1975) Indirect threaded code, Comm.ACM, 18, 330-331.
19. Morgan, H.L. (1970) An interupt-based organisation for management information systems, Comm.ACM, 13, 734-739.
20. Zelkowitz, M. (1971) Interupt driven programming, Comm.ACM, 14, 417-418.
21. Kohout, L.J., and Gaines, B.R. (1976) Protection as a general systems problem, Int. J. General Systems, 3, 3-23.
22. Hewitt, C. (11971) Procedural imbedding of knowledge in PLANNER, Int. Joint Conf. on Artificial Intelligence, London, Sept.
23. Burstall, R.M., Collins, J.S., and Popplestone, R.J. (1971) Programming in POP-2, Edinburgh University Press.
24. Wegbreit, B. (1974) The treatment of data types in EL1, Comm.ACM, 17, 251-264.