A MIXED-CODE APPROACH TO COMMERCIAL MICROCOMPUTER APPLICATIONS

B.R.Gaines, M.A., Ph.D., C.Eng., F.I.E.E.*

Summary

It has become very attractive to develop office automation systems based on microcomputers. However, currently no single one of the commonly available languages on such machines is suited to the full range of requirements: interactive dialogue; accounting; statistics; word-processing; data-base management; etc. This paper outlines the key requirements, tabulates the strengths and weaknesses of current languages, and describes a system that is now in use in a number of commercial applications which allows different parts of an application to be written in the most suitable language, yet the whole to be simply and uniformly integrated together.

1. Introduction

The advent of low-cost, but computationally very powerful, microprocessors, together with stores, discs and terminals that match them in cost, reliability, and lack of special environmental requirements, has opened up a new market for commercial computer systems. The personal computer in an office, serving an individual or local community, can provide the power of a previous generation of central service machines, but under the direct control of local staff (ref.1).

This potential is now obvious to many computer users and is beginning to be tapped by suppliers to provide a wide range of systems ranging from simple office automation, to full management information systems. However, as often in the past, the lack of available system software catering for this class of applications currently places severe limitations on the full exploitation of the hardware capabilities now available.

This paper outlines some of the software problems and suggests that it is unlikely in the near future that some "universal commercial language" will become available to solve them. Instead techniques are proposed to make best use of currently available languages on microcomputers, paying particular attention to the systems engineering aspects of "modularity", "developmental tools", "quality control", etc., in the context of software.

2 Language Requirements

Much of the discussion in this paper is relevant to a wide range of applications of microprocessors in instrumentations systems, etc. However, I have in mind specifically the context of systems used interactively by non-computer-oriented personnel for a wide range of secretarial, clerical and managerial tasks. From experience of such applications (refs.2 & 3), the key software requirements are:-

(1) Compactness of system and applications software - this remains of major significance despite the decreasing cost of semiconductor storage and the increasing address scope of microprocessors - it will be a key

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

requirement for many years to come that operating systems and language run-time systems are a few kilobytes, not the 100's of kilobytes common on mainframes. The main design constraint that arises out of this requirement is that systems software needs to be tailored to a well-defined range of applications and precisely planned to provide the all the functionality required, but only that actually required.

(2) Interactive software development - the low-cost of the hardware makes software costs dominant, and aids to program development are very significant to overall system costs - the ability to rapidly develop and check out programs interactively, probably at the same terminals at which they will be used, is a major contribution to low system cost.

(3) Self-documenting language - once a system specification has been determined, a substantial part of the cost of software development is documentation and the language used should make this simple and integrated into the program specification itself - users can rarely afford the cost of separate documentation and the system should be designed to make it unecessary.

(4) Modularity through an efficient procedure call with parameter passing and local variables - the ability to apply hardware engineering techniques based on the development of a range of standard modules that can be validated independently of one another and used in a diverse range of applications is vital to software reliability and quality control (ref.4) - in particular it minimizes software costs by allowing previously developed, standard modules to form the core of even highly "customized" systems.

(5) Flexible character-string handling - we have emphasized previously (refs.2,3 & 5) the importance of being able to program interactive dialogue with the user in a direct and natural manner that leaves the programmer free to use, and accept, precisely those formats that the user expects - the programming language itself must in no way restrict, or introduce artefacts into, this dialogue.

(6) Flexible and efficient record-structured file access - the "database" aspect of commercial computing is of major significance and one is more often accessing the backing store than doing actual numerical computation - reasonable record structuring facilities are necessary combined with the capability to make both space-efficient, and time-efficient, use of the backing store - this is particularly significant if it is a "floppy" disc.

(7) High-precision integer arithmetic - accounting calculations need to be exact and any rounding precisely controlled to whatever are the accepted rules - 13 digit data and 19 digit intermediate results are sometimes necessary.

(8) Mathematical functions and libraries - although the primary function of most small commercial systems is not numeric calculations, there is still a call on a range of financial computations such as discounted cash flow, trend-projection etc. - as databases are built-up it becomes increasingly difficult to keep track of them in detail and statistical summaries and projections become increasingly important - it should not be necessary to program these up in detail for each new application and a standard library of such programs should be available.

(9) Speed when required - the sections of commercial programs that are

compute-bound are small but significant - for over 90% of the code it is most effective to trade speed for ease of use, compactness of code, etc. - however, sometimes (e.g. in sorting) speed is significant and there should be means for reversing the trade-offs and speeding up particular routines.

(10) Ease of access to, and control off, a wide range of peripheral devices and terminal facilities - the local office computer may nowadays have as many peripherals attached as a main-frame: multiple visual displays, printers, typewriters, graph plotters, special purpose terminals, etc. - device- independence and flexible configuring are important.

(11) Protection against other tasks in a multi-task environment - many commercial systems, even though locally dedicated, run continuously on-line providing essential services to multiple users - it is costly to be forced to develop and update such systems only outside the normal working day, and it may be impossible to test modifications fully except under normal operation - facilities to restrict the available environment of new, untrusted modules are essential, and generally to be able to set up one tasks guaranteed against failures in others.

(12) Instrumentation of software behaviour and performance - the ability to probe and collect information from a hardware system in order to test and evaluate it is a standard requirement - similar facilities are necessary in software not only at the lowest levels (where various instruments now provide them) but also at the highest levels where only the operating system and language run-times systems have adequate information on the structures involved to provide meaningful data.

3 Available languages

Some parts of the preceeding list are controversial, reflecting engineering pragmatism rather than computer science, whilst other requirements seem so obvious that no reasonable system software should be offered without them. Table 1 gives a brief analysis of the main languages (various assemblers, Fortran and Basic) currently available on microcomputers in terms of the above requirements. I have included CAP's MicroCobol as an example of the new family of business-oriented languages becoming available for minis and micros. I have included another non-standard language, Basys II (ref.7), since this is a major component of the system described in this paper. Currently some such nonstandard language, or a set of nonstandard extensions to Basic or Fortran, is necessary to cover some of the main requirements listed. Other nonstandard languages available on microprocessors may readily be classified in terms of this table.

One feature of the table that is clearly apparent is that no one language uniformly satisfies all the requirements. Conceivably some language might do so - an interpretive version of Algol 68 (ref.8) would come closest to doing so provided the run-time system could be made to satisfy (1). However, the very generality of such a language is self-defeating since the declarations of the data-types and operations required itself becomes complex, and must be done specifically for each of the very diverse types of computation involved. In essence, using a very general language one has to specify strong contexts for particular sub-programs and, effectively, define various sub-languages. This is so little different from actually using different languages that one wonders whether a single, "universal" language would actually win out against a combination of separate languages - provided they can be used together effectively.

Table 1 Comparison of Microcomputer Languages

Language:	Assembler	Fortran	Basic	Basys II	Micro-Cobol
General Remarks	Good macro assemblers commonly available	Full Fortran IV commonly available	Extended versions commonly available	Nonstandard language - not commonly available	Cobol subset on a range of machines
(1) Compact	Good	Run-time system small if properly organized	Can be small	Good	Good
(2) Interact	No	No	Yes	Yes	No
(3) Self-Document	Can be good if standards set & macros used	Poor layout in standard versions	Poor layout in standard versions	Good with multiple commands/line and procedure names	Can be good if standards set
(4) Modular	Can be good if standards followed	Can be reasonable if standards followed	Very poor due to only global names and lack of proper calls	Good due to segment structure and local names	Can be good with proper standards
(5) String-Handling	Needs specialist routines	Poor - language unsuitable	Reasonable in some versions	Excellent - specialist language	Poor - use specialist routines
(6) Record-Structure	Needs specialist package	Needs specialist sub-program	Language generally unsuited	Needs specialist segment	Very good - Cobol file structure
(7) Long Integers	Need specialist routines	Not suited	Not suited	Excellent - language feature	Very good - language feature
(8) Maths Library	Specialist routines	Excellent	Not suited	Not suited	Not suited
(9) Speed	Excellent	Can be very good	Poor	Not suited - but gives easy access other langs.	Poor - can use intermediate language
(10) Peripheral Control	Excellent	Depends on OS interface - often good	Depends on OS interface - often poor	Very good	Very good
(11) Protection	No - unless hardware	No - unless hardware	Good - interpretive	Good - interpretive	Not multi-users/tasks
(12) Instrument	No - unless hardware	Little	Little	Good - by spy-segment	Good - by debug aids

It is interesting to note the role of the lowest-level language, the assembler, in Table 1. It shows up as satisfying most requirements provided standards are set, and adhered to, and specialist modules are written. These last points make obvious sense since the assembler makes available the machine at its most flexible and least structured level - all other languages may be viewed as restrictions on the assembly language programmer, forcing him to obey certain conventions and providing him with additional pre-programmed facilities. If it were not for the problems with (2), (11) and (12), assembly language (as noted in ref.9) would be a very good kernel about which to build. All of these problems stem from the fact that micros currently (and most minis and mainframes !) do not adequately support their own machine code. An adequate protection scheme (e.g.PP250 capability system ref.10) integrated with procedure calling (ref.11) would put an "Excellent" rating under Assembler in rows (1), (11) and (12) of the table. Other architectural enhancements based on a wider range of data types (ref.12) would build into the hardware the "specialist routines" of rows (5) through (8). It can only be a transient manifestation of the inadequacies of current machine designs that the assembler (albeit a "high-level" one) plus some standard procedures does not provide us with adequate facilities.

Currently, however, we have no universal languages on micros and only a combination of available languages can achieve adequate ratings under all the headings of Table 1. The following section describes how 3 languages, Assembler, Fortran and Basys II, have been married together to form a powerful, integrated combination, that has been successfully applied to a range of commercial systems.

4 A Mixed-Code System

The system to be described is one that makes best use of the appropriate software on a particular machine by allowing programs written in different languages to be freely intermixed. This is certainly not a new approach or requirement in itself, but what is innovative (for small systems) is the:

(i) Modularity of the system - sub-programs are not bound together until run-time and may be dynamically loaded and unloaded;

(ii) Integration of the system under an interactive operating system such that run-time errors in essentially non-interactive languages are reported interactively as if they occurred in the interpretive part;

(iii) Inherent sharing of modules between tasks even though the language in which they were written was not designed to allow such sharing.

There is also a common procedure call for modules in any language so that the user of a module does not have to know in what language it was written.

The machine used in this work is the DEC lsi11 which has a good macro assembler and Fortran compiler. However, virtually none of the development is specific to this machine and both the Zilog Z80 and IBM Series 1, for example, can support the same approach. In our previous applications of a variety of machines to commercial systems (ref.2) the interpretive language Basys has been used with assembler extensions for speed or special purposes. Basys (ref.7) is a language similar in syntax to Dartmouth Basic but with integer arithmetic and extensive string-processing facilities, developed specifically to satisfy requirements (5) and (7) in Table 1. Because Basys is table-driven it has been very simple to add to it new commands by linking assembler routines through the appropriate tables

(ref.13). However, this form of static linkage results in several different versions of Basys, whereas one would like to validate and maintain a single version of the interpreter. There have also been problems in writing some numeric routines in the integer arithmetic of Basys and it became desirable to link in Fortran programs also. Again, it would have been possible to extend the language with more data types and operations, but this would have increased the size and complexity of the interpreter for all applications.

The solution adopted in Basys II has been to set up a run-time environment for Basys consisting of a number of independent "segments" which may be dynamically loaded with individual Basys, Assembler or Fortran, programs. These segments may be used as (multiple-entry) sub-programs by Basys programs and control transferred to them through a procedure call which allows parameters to be passed by value or by reference, and is the same for all segments regardless of the language in which they are written. The key features of this approach are described in the following sections on Basys II segment management and procedure calls.

4.1 Segment Management

The Basys II command: Fetch [Size] Segnam.Ext

allocates an area of memory to the segment "Seg" (known by its 3-character name) and, if appropriate, loads it from the file "Segnam.Ext". The extension, "Ext", determines the type of segment:

Ext.	Type	Fetch Action
Bay	Basys program	Size, allocate & load
Job	Basys job	Allocate, length "Size", link into round robin & run Basys program
Rel	Assembler or Fortran prog.	Size, allocate & load fixing relocation
Ter	Terminal handler	Size, allocate and load fixing relocation and plug in interrupts
Sys	RT11 device handler	Size, allocate and load fixing RT11 tables and plug in interrupts
Dat	Common data	Allocate and load initial data
‹none›	Unspecified	Allocate, length "Size"

The Basys program segments (.Bay) are passive, shareable programs whereas the Basys jobs (.Job) are active programs together with workspace and data areas running as separate tasks under a clock-driven, round-robin scheduler.

The RT11 Assembler and Fortran compiler both produce object modules which may be linked with libraries to give relocatable code in a common format, and it is straightforward to load this (.Rel) and fix up addresses where necessary.

The handler segments (.Ter & .Sys) allow a system to be flexibly configured, and dynamically re-configured, for different sets of

peripherals.

The data (.Dat) segments allow different jobs to share a common data area, and the unspecified segment allocation allows system programs to set up special segments for other purposes.

The Basys II command: Free Seg[nam.Ext]

undoes any linking to schedulers and interrupts and releases the area allocated to the segment, "Seg".

4.2 Procedure Calls

Basys II has a similar syntax to Basic in that all program lines have numbers. However, they may also have symbolic labels consisting of up to 3 characters preceeded by an asterisk (to distinguish them from variable names). Lines may be referenced by number or label. Thus the following is a fragment of a Basys II program as it would list:

```
 20*In  Print 'Test:' :Input X Y
 30     If X>Y :Goto *Cal
 40     Print 'First number must be greater'; :Goto *In
100*Cal Let Z=X*A(Y) P=B(Y) :Do /VolTst %Z Y
```

The Basys command, Do ⟨program line⟩, causes the specified line to be executed but does not transfer control to it. In Basys II, this command is extended to be a procedure call with parameter passing. An example is given in line 100 of the program fragment above where the final command specifies that the procedure "Tst" in the segment "Vol" is to be executed, passing to it a reference to the variable Z and the value of the variable Y. "Tst" itself might have the form:

```
100*Tst Begin A B
110     Let A=(2+A)*B+V0(A)
190     Back
```

where the "Begin" indicates that the line to be done heads up a whole procedure, and the "Back" indicates the procedure end and returns control to the calling program. In the example given, A and B are set up as local variables when "Do /VolTst %Z Y" is executed, and become a reference to Z and the value of Y, respectively. The local variables disappear when the "Back" command is executed, although, of course, any value assigned to A has been automatically transferred to Z.

Note that the line numbers associated with "Tst" have no interaction with those of the calling program because the called routine is in a different segment. This, together with the symbolic labels, removes many of the problems of Basic line numbers whilst still preserving their advantages in program editing.

The action taken by the Basys II interpreter in executing the "Do /VolTst" is to look up the segment "Vol", determine it is a Basys program, and evaluate and set up the parameters, %Z and Y, accordingly. If, however, "Vol" is found to be an Assembler or Fortran segment (.Rel), the two parameters are evaluated and set up according to RT11 Fortran conventions. This is totally transparent to the programmer writing the calling routine and he is not aware of the language in which "Vol" is written. Thus, it is not only possible to mix languages but also to write procedures in one language and then transfer them to another later, e.g. to gain speed,

without having to make any changes in the programs that call them.

4.3 Interface to Fortran and Assembler

The "Call" command in RT11 Fortran (ref.14) executes a "JSR PC,START" with R5 pointing to a parameter block of the form:

```
Number of parameters
Address of first parameter
Address of second parameter
   etc.
```

where "START" is the first address of the sub-program. Control is returned by an "RTS PC". Thus Fortran sub-program calls are compiled into simple subroutine calls through the stack with register R5 indicating any parameters passed.

Basys II uses this convention to send to a Fortran program the following information:

```
Number of arguments passed
Label of called routine (in Radix50)
First argument location
First argument size in bytes
Second argument location
Second argument size in bytes
  etc.
```

Thus a Fortran sub-program for the segment "Vol" above might begin:

```
SUBROUTINE  FORSUB (NPAR,SWITCH,PAR1,PAR1L,PAR2,PAR2L)
INTEGER  NPAR,SWITCH,PAR1,PAR1L,PAR2,PAR2L
```

where NPAR is the number of parameters passed, SWITCH is the 'line label' in radix50 and may be tested to determine an entry point, PAR1 is the first parameter passed by reference and PAR1L is its length, etc. The length specifications allow arrays of variable length to be passed to the Fortran program. This is particularly useful since Basys II allows for dynamic variation in array size whereas Fortran IV does not except in arrays passed as parameters. Note that, since Fortran expects all parameters to be passed by reference, the Basys II "Do" command processor passes a reference to temporary workspace for parameters passed by value (e.g. Y in the example of the previous section).

Assembler segments are treated by Basys II exactly as if they were Fortran sub-programs and hence have to start at their first location and accept a parameter block in RT11 Fortran format. Neither requirement is at all restrictive.

Run-time errors in RT11 Fortran result in traps and Basys II picks these up as if they were run-time errors in a Basys program. It decodes the Fortran error information to indicate the source line number in the original Fortran program where the error occurred. Thus one might obtain:

```
>List
 125      Let X=12 :Do /ForSub %X
>Run
Fortran 340 Error 15 at 125
>
```

where the error message indicates that at line 340 of the Fortran program source error type 25 (Fortran manual: log of negative number) occurred and the Basys program calling line was 125. Thus, the diagnostics given are very detailed and, since the fault is most likely to be in the data transferred (the Fortran programs tend to be small and easily debugged), it is convenient that the system returns to Basys II program level where the data actually transmitted may be examined.

The scheduling of Basys II programs is based on examination of the system clock after execution of each line the program. Thus a program may be de-scheduled whilst traversing a Basys II program segment but this does not matter since such segments contain no dynamic data and are intrinsically shareable (all dynamic data is back in the calling (.Job) segment). Entry to a Fortran or Assembler segment is treated as if it were execution of a single line of Basys program so that no de-scheduling can take place until exit. Thus programs in such segments must be designed to take a reasonably short time (less than 0.2 seconds say) to maintain system responsiveness in a multi-user environment. This turns out to be a reasonable constraint since such segments are generally being used for their speed anyway - long calculations can be made to return regularly if necessary.

The Fortran and Assembler segments are also shareable provided they do not leave any data in local workspace in the segment. The calling program can always pass to them some of its own workspace to avoid the need for this. In effect, the Basys II operating system provides 'transaction processing' for Fortran and Assembler programs and 'time sharing' for Basys II programs. Coupled with the interrupt-driven segments that operate independently within this framework, these various modes of operation lead to highly effective and powerful real-time systems on a very small machine.

It is possible to assemble and compile programs at the same time as Basys II is running, so that Fortran and Assembler segments can be developed and modified on an operational system. Lack of memory protection hardware on the lsi11 makes this of dubious value currently, but it will certainly become useful as full memory management is introduced for micros.

Note that only Fortran sub-programs are used, not main programs, so that no input/output system is loaded for Fortran and all peripheral transfers are handled through Basys II (which provides very direct access to RT11 facilities).

4.4 General Remarks

There are a number of small enhancements to the system described that have proved useful in practice.

The Basys II command: System Lab Parameters
is the same as: Do /SysLab Parameters
and calls a system library segment which contains standard routines for date entry and output, overlay calling, dialogue processing (enforcing the rules of ref.5), etc.

The Basys II command: Our Lab Parameters
is the same as: Do /XXXLab Parameters
where XXX is the generic name for the program suite in use, set up when it was started.

In a typical application to a stock exchange Gilt Dealing Records system there are 4 program segments in use in addition to overlays in the job segment:

Name	Language	Function
Sys	Basys II	System library
Gdr	Basys II	Application library handling database access, record structures, etc., for whole program suite.
Gdf	Fortran	Yield calculations
Gds	Assembler	Sort routines for database output

The assembly language sort routines in Gds exemplify the value of the approach since the source is only some 30 lines long and readily checked out, yet it gives speed precisely where it is most needed. The Fortran program for Gdf is also short and readily developed and maintained - it is a standard library module for a range of such financial systems.

Because of the various library segments the actual Basys II program overlays performing various tasks tend to be very short, generally a page or less in length, and just contain the code necessary to call the library routines to execute the desired functions.

One other value of the segment structure is that it provides a very convenient mechanism for program instrumentation and measurement. Various forms of 'spy' segment have been written that allow one segment to monitor the operations and data structures of another. These are useful not only as debugging aids, but also in real-time, multi-task environments, to monitor the overall system operation and integrity.

5 Summary and Conclusions

Basys II has now been in use for some 18 months in a variety of real-time commercial systems and the approach described here has proved its value. The extreme modularity of the segmented program structure has allowed very rapid program development with far less debugging required. This has been achieved without increase in the size of the Basys interpreter which is still only 10 Kbytes with full dynamic segment management and the enhanced procedure call. It is suggested that the techniques outlined here would be of value in a variety of microprocessor-based systems, not only for commercial purposes, but also for industrial and clinical instrumentation, etc. They allow full advantage to be taken of the best software available on a given machine by embedding it in an overall, coherent framework that gives maximum opportunity for direct interaction with the system both in development and use. Our experience suggests that the costs of doing this are small and the benefits very great indeed.

6 Acknowledgements

I have been helped in the development of Basys II by the critical comments of Bruce Anderson, Peter Facey, Dick Pope, John Zuckerman, and many Basys users. I am grateful to Dick Waller of CAP for information on their Micro-Cobol.

7 REFERENCES

1: B.R.Gaines, Minicomputers in business applications in the next decade, in Infotech State of the Art Report on "Minis Versus Mainframes", 1978.

2: B.R.Gaines & P.V.Facey, Some experience in interactive system development and application, Proceedings I.E.E.E. 63(6), June 1975, 894-911.

3: B.R.Gaines, P.V.Facey & J.B.S.Sams, Minicomputers in security dealing, Computer 9(9), Sept. 1976, 6-15.

4: B.R.Gaines, Hardware engineering and software engineering, Euromicro 3(2), Apr. 1977, 16-21.

5: B.R.Gaines, Programming interactive dialogue, in Pragmatic Programming and Sensible Software, Feb. 1978, Online Conferences, Uxbridge, Middlesex, UK, 303-320.

6: CAP, MicroCobol Language and User Manuals, March 1978, CAP Microsoft Ltd., London, UK.

7: B.R.Gaines & P.V.Facey, BASYS - a language for programming interaction, in Proceeedings of Conference on "Computer Systems and Technology", IERE Conference Proceedings 36, March 1977, University of Sussex, Brighton, UK, 251-262.

8: C.H.Lindsey & S.G.van der Meulen, Informal Introduction to Algol 68, 1977 (2nd ed.), North-Holland, Amsterdam, Holland.

9: B.R.Gaines & P.V.Facey, Real-time system design under an emulator embedded in a high-level language, Proceedings of British Computer Society DATAFAIR 73, Apr. 1973, Nottingham, 285-291.

10: D.M.England, Architectural features of System 250, in Infotech State of the Art Report on "Operating Systems", 1972.

11: B.R.Gaines, M.Haynes & D.Hill, Integration of protection and procedures in a high-level minicomputer, in 1974 Computer Systems and Technology Conference, Oct. 1974, I.E.E., London, UK.

12: B.R.Gaines, P.V.Facey, F.K.Williamson & J.A.Maine, Design objectives for a descriptor-organized minicomputer, EUROCOMP 74, May 1974, Online Ltd., London,UK, 29-45.

13: B.R.Gaines, Interpretive kernels for microcomputer software, in Proceedings S.E.R.T. Symposium on "Microcomputers at Work", Sept. 1976, University of Sussex, Brighton, UK, 56-69.

14: RT11/RSTS/E Fortran IV User's Guide, DEC-11-LRRUA-A-D, 1975, Digital Equipment Corporation, Maynard, Mass., USA.