British Computer Society

DATAFAIR 73

Real-Time System Design Under an Emulator Embedded in a High-Level Language

Peter Facey and Brian Gaines

Department of Electrical Engineering Science
University of Essex, Colchester, U.K.

Abstract

The paper discusses the problems of minicomputer software development and describes experience in the development of a small real-time computer system under an interactive emulator embedded in extended BASIC on the PDP10.

Major problems with minicomputer software development are:

a) Even if the minicomputer is capable of supporting adequate software development tools, the configuration for development may not do so (lack of core, backup storage etc.).

b) The operation of the system may not be possible except on-site or in conjunction with specialist peripheral devices.

c) The system development may involve the definition of specialist hardware interfaces and peripherals which will not be available during the software development phase.

These consideraitons suggest that system development under an emulator is attractive particularly one on an interactive time-sharing system where the 'hands-on' advantages of the minicomputer will not be lost.

To emulate at a reasonable speed it is necessary to code an order-code interpreter for the minicomputer in machine language (or preferably as a microprogram), but to maintain maximum flexibility in emulating peripheral and associated systems a high-level language is desirable.

This conflict has been resolved in the Essex system by adding a command 'MINIC' to thelanguage BASYS (Essex Extended BASIC) on the PDP10 which has the same syntactical structure as other BASIC commands but takes as its argument the name of an array to be regarded as the core store of a MINIC minicomputer. When the instruction is executed it takes the first 20 locations of the array as the MINIC hardware registersand as status information controlling execution and data traps, time-quantum for execution, etc., and emulates the MINIC CPU. Control returns back to the BASYS program when either a (emulation-level) time quantum has elapsed, a certain number of instructions has been executed, certain instructions have been executed or certain data accessed, or an input-output or transfer to variable microprogram instruction has been executed. Status information is set up in the array argument to indicate the reason for the return to main program and the internal status of the MINIC CPU.

This project has demonstrated in an actual real-time system development the advantages of working under an interactive emulator embedded in a high-level language. The experiments have also emphasized the importance of other facilities such as a high-speed contextual editor. Summaries of BASYS and a suitable editor are given as appendices to the paper.

REAL-TIME SYSTEM DESIGN UNDER AN EMULATOR EMBEDDED IN A HIGH-LEVEL LANGUAGE

PETER FACEY AND BRIAN GAINES
Department of Electrical Engineering Science
University of Essex, Colchester, U.K.

INTRODUCTION

This paper discusses the problems of minicomputer software development and describes experience in the development of systems software for a minicomputer using an interactive emulator embedded in an extended BASIC-like language on a PDP10 timesharing system. The systems developed for the target minicomputer include a BASIC interpreter, a floating-point arithmetic package and a real-time clinical patient-monitoring system for the minicomputer with drum, cartridge tape, visual display, real-time clock, analog/digital convertor, teletype, fast reader and special-purpose keyboards. The objectives of this paper are to further elucidate some problems of software development for minicomputers and the relative merits of differing approaches, and also to demonstrate the practical utility of the modular BASIC-like language (BASYS or Essex extended BASIC for system programming) which was originally described at DATAFAIR 1971 (Gaines, Gedye, and Facey 1971).

It is perhaps worth emphasising from the outset that the objectives of the work and the conclusions to be drawn from this paper are far more related to the ergonomics of software/system development than to the computer-science underlying the techniques involved. We have been concerned to place the system developers in a position of power to fully and efficiently utilize their own capacities and have not hesitated to be lavish with the computer facilities required to support this objective. The original thesis has been that problems of system development and implementation on computers generally reflect poor man-machine communication This applies not only to the implementation of defined systems but also to the, generally most difficult and unmanageable, phase of <u>system definition</u>: the capability to rapidly develop demonstration systems to enhance man-man communications during system definition seems to us to be one of the most important facilities that should be made available through the use of computers.

MINICOMPUTER SOFTWARE DEVELOPMENT

The availability of small computers of ever-increasing power and ever-decreasing cost has made it attractive to design systems in which the minicomputer is just another component, albeit the most complex and powerful one. It is probable that already more programming effort is being devoted to these minicomputer 'components' than to the larger computer installations. This will certainly be so when the costs fall even further with the exploitation of complex semiconductor production techniques.

Whilst the use of minicomputers in this way has opened up new areas for system implementation on a scale previously impossible, it has also generated new problems (or, at least, regenerated old problems in a new guise). The fabrication of a system by 'programming' rather than by more conventional and concrete engineering techniques gives the system a unique flexibility and capability for modification both during development and in the field. However, the shift of implementation technology requires a concomitant shift in design and development technology: the development of software equivalants to the drawing office, prototype production, exploratory test-beds, etc. This is all the more important because many, perhaps the majority, of minicomputer buyers are 'first-time' users without previous experience of computers and without computer-based support facilities. In time the ready availability of the machines which is causing this flood into new territory will also have the compensatory effect of saturating most potential applications areas - however, for several years at least, the growth of applications is likely to exceed the growth of the necessary backup technology.

It is not trivial to note that systems heavily dependent on computer software development are those most ripe for the application of 'computer aided design'. This may be obvious but the fact that editors, assemblers and compilers are CAD tools for software development itself tends to be overlooked, in that they are seen as an essential part of the computer system, not as tools largely independent of the computer which should be tailored to the requirements of the system designer. It is true that most minicomputer manufacturers supply the design tools for their computer software in such a form that some configurations of their own computer can actually implement them. However, this is technically irrelevant, and introduces some practical confusion in that it is highly unlikely that the minicomputer configuration required for the final system is able to support an adequate software generation system - the two specifications are generally far apart.

Cross-assemblers and compilers enabling the programs for one machine to be created on another have been in use for some time. However they represent only one comparatively minor step in the design process: the translation from program specification to machine-acceptable code. On one side of them are the text creation systems such as interactive editors, and on the other side are the code testing systems such as emulators. Both editors and emulators are systems which cannot be handled well under batch facilities and require a good interactive system to support them. This, in itself, is a positive reason for having design software on a minicomputer: it is easy to provide an interactive editor on a suitable minicomputer configuration, and the machine can generally be programmed as an efficient emulator of itself under a symbolic debug. However, the editor requires backup store, preferably drum or disc, and the debug requires additional mainframe memory.

The logical conclusion is that for minicomputer software development an interactive system, not necessarily related to the target system, should be set up with suitable computer-aided design tools for minicomputer software development: a powerful interactive editor, an assembler (preferably a meta-assembler) and an interactive emulator. This has been our approach to developments on a number of systems, for example clinical timesharing systems on the PDP8 and micro-16, and the effort involved in writing effective editor/assembler/debug systems has been well worth the resultant saving in development time. The software development aids in these cases were written on the actual target machines because in all cases the configuration had drum or disc backup stores and sufficient main memory.

When faced with a situation, however, in which we did not possess a suitable target machine configuration, we went further than this and set up cross-assemblers and cross-emulators for the target system (a Micro-computer systems MINIC) on available open-shop systems (a Digital Equipment PDP9 and PDP10). In particular, the PDP10 MINIC-emulator was written as an additional module for the BASYS (extended and modified BASIC) interpretor we had written for general use on the PDP10. The advantages of doing this and the form of implementation are described in the next section.

A MINIC EMULATOR UNDER BASYS

To write an instruction-set emulator for one machine upon another is generally simple. However, to turn such an emulator into an effective design tool presents problems. Generally an emulator of the computer alone is inadequate for all but pure software development. What is required is a complete configuration emulator including peripherals such as discs, analog/digital convertors, printers and displays, and the timing problems associated with real-time control of physical devices. At the design stage moreover many of these peripheral systems may not be available, their characteristics may be unknown, or it may be part of the design to actually develop them. If the emulator is written as a machine-code program it is comparatively simple to run the target computer mainframe diagnostics on it to check its operation. However, if the ill-defined peripherals are also emulated with machine-language programs the process of system development involves joint writing and debugging of machine code programs on both the design machine and the target machine - a situation fraught with error.

This problem could be largely overcome by writing the emulators in a suitable high-level language. However, in emulating the target computer speed is of great importance in order to provide processing power on a comparable time scale (generally about 10-20 times slower). Since the computer will remain constant whilst peripherals change with configuration, it seems reasonable to combine the advantages of both approaches by embedding a machine-code computer emulator in the high level language with suitable facilities for calling and controlling it.

It so happened that the extended BASIC we had developed for 'natural-language' interactive text-handling also featured ease of machine-code linkage through its table-driven command structure and standard subroutine calls. The language was being successfully used for clinical applications and textual analysis and such additional commands as special-terminal drivers and a speech-synthesiser controller had already been added to it as linked machine-code segments. The emulator embedded in a high-level language presented a further opportunity to test BASYS since the text-handling and variable-radix arithmetic of the language should be able to support effective assemblers and symbolic debugs; the file-handling and input-output control (available down to virtually direct machine-level) should be well suited to peripheral emulation; and the ease of integrating further machine-code modules should make it possible to embed a fast emulator for MINIC itself in a convenient form.

A synopsis of BASYS is given in Appendix 1 - one further command was added to call the MINIC emulator -
MINIC (array name)
where locations 0 through 20 of the array contained status information and the remaining locations were regarded as a MINIC core store. The status information included the state of MINIC registers, the number of MINIC instructions to be emulated, etc. On executing this command the MINIC emulator was called and the specified numbers of instructions emulated with appropriate changes to the array 'core and registers' and status information. As usual command then passed to the next instruction in the program line unless the emulation could not be completed because MINIC input/output or extended microprogram instructions had been encountered - in these cases control was transferred to the following line with appropriate status information set up to indicate the cause of interruption.

It was convenient to pack 4 MINIC 8-bit words into one 36-bit word on the PDP10 and use the residual bits to tag each MINIC word and cause interruption when it was accessed by the emulator; hence break-points were simply realized under the emulator. Running under interrupts was simulated simply by setting the number of instructions to be executed, and real-time interrupts from the PDP10 timesharing teleprinter were generated through a buffer flag test. MINIC peripherals were simulated by BASYS program segments entered after interruption of the 'MINIC' command which adjusted the array 'core and registers' to emulate the appropriate peripheral transfers.

EXPERIENCE WITH THE EMULATOR SYSTEM

An assembler for the normal MINIC assembly language was written in BASYS in about two man-days. A symbolic debug and environment for the MINIC emulator was written in BASYS in about three man-days. This system was then used to generate two packages previously written for MINIC, a floating-point arithmetic package and a BASYS interpreter. At the same time, well before a hardware configuration was available, work commenced on writing a clinical monitoring system for a real-time MINIC with a variety of peripherals. These were not formal experiments and could not be so since the 'control' developments differed in detail and had not been adequately monitored. The results, however, indicate clearly the tremendous potential of an effective interactive system to aid software development - the software packages were developed in about one tenth of the time and took up about sixty percent of the space of comparable previous developments. For the real-time system the back of the software development was broken well before the hardware was ready and the transfer to the real system was fairly painless.

This is an ongoing experiment and the results of this first phase do little more than indicate the value of further research on the optimization of the software development environment. However, the following conclusions are so far worthy of note -

(a) A major benefit accrues solely from the use of a powerful high-speed interactive text editor. In particular the block move capability made it easy to re-arrange programs for ease of reading and to annotate them sensibly.

(b) The majority of program bugs may be detected through a combination of assembler diagnostics and the examination of well laid out and formatted listings. In particular the lack of hardware for the real-time system diverted effort into examination of the program texts and eliminated most faults even before tests on the emulator.

(c) Ease and speed of operation are of paramount importance. The magtape-based editor on the PDP9 was substantially less satisfactory than the disc-based PDP10 and this is strongly reflected in the program quality of the parts of the system produced on the PDP9 editor/emulator system.

(d) An emulator embedded in a high-level language does combine speed with ease of peripheral simulation and debugging support in a powerful combination.

In some respects we are only re-discovering well known (if not well-kept) programming adages. However, as Sime, Green and Guest (1973) have suggested it seems eminently sensible to investigate the psychological/technical foundations for the production of good software, and to use the knowledge gained to design more effective program design and development aids.

REFERENCES

Facey, P.V. ED - A Contextual Editor, Department of Electrical Engineering Science, University of Essex.

Facey, P.V. BASYS User's Manual, Department of Electrical Engineering Science, University of Essex.

Gaines, B.R., Gedye,J.L., Facey,P.V. A Versatile Multi-User Interactive Language System for a Minicomputer, B.C.S. Datafair 71, Nottingham.

Sime,M.E., Green,T.R.G., Guest, D.J. Psychological Evaluation of Two Conditional Constructions Used in Computer Languages, Int. J. Man-Machine Studies 5(1) January 1973.

APPENDIX 1 SUMMARY OF BASYS

BASYS is a systems and interactive applications programming language with a syntax based on Dartmouth College BASIC. It is an ongoing development and versions of the language have been used on various configurations of the PDP8, PDP9, PDP10, PDP11, PDP15, MINIC, MODULAR ONE, and MICRO 16.

The main extensions are -

(1) Improved string-handling modelled on SNOBOL - handles arbitrary-length, dynamically changing strings with full ASCII 7-bit character set - string relational operators allow for anchored and embedded searches, and tests for string equality, inequality or telephone-directory comparisons.

(2) A new 'PUT' command provides keyword-searching, pattern-matching, and string-decomposition facilities. These give simple and natural syntax-analysis capabilities and allow conversational programs to be written which can communicate with untrained users in an approach to natural language.

(3) A new 'CODE' command enables a running BASYS program to modify itself, dynamically generating arrays or new code.

(4) All error conditions may be trapped by a BASYS program enabling supervisory programs to be constructed that give the user access to the language but retain control.

(5) Numerical expressions are accepted wherever a number could occur, and these may contain logical and relational operators. Hence all transfers of control may be computed.

(6) Several commands may follow on one line and any statement may follow a conditional. Many commands have implied conditionals such that execution along the line continues only if the command is successfully executed.

(7) Extensive file-handling facilities allow BASYS programs to create, delete, rename, read and write files, either in ASCII or in binary form. A single LET command allows a list of numeric and string variables to be packed or unpacked to or from an array, and arrays themselves may be written into arbitrary locations in random access files - hence complex record structures are readily set up.

(8) Simple overlaying is possible through commands which load and run a filed BASYS program passing to it the current simple variables and selected strings and arrays.

(9) A variety of priviledged modes allows the construction of system level programs protected from examination or interference by the user. Such system level programs are automatically called when the user logs in or out, or presses certain control keys - hence the system may be trailored in detail to the exact requirements of each class of user.

Expressions in BASYS

A BASYS program consists of a set of numbered lines each containing one or more commands separated by colons, e.g.

```
100 INPUT $5 :UNLESS $5='YES' :PRINT $5'?' :GOTO 200+K
```

A numerical expression (ne) is something that can be evaluated to produce a number. The following operators are allowed:-

PREC	OPERATOR	MEANING
1	-	Unary minus
2	←	a←b: open bit shift of a by b places left if b>0, right if b<0
3	↑	Exponentiation
4	/	Division
5	*	Multiplication
6	-	Subtraction
7	+	Addition
Relational operators		
8	<	Less than
8	=	Equal
8	>	Greater than
String operators		
9	>	Alphabetically greater than
9	=	Identical
9	<	Alphabetically less than
9	↑	a↑b: true if string-a begins with string-b
9	←	a←b: true if string-a contains string-b
10	'or"	Quotes enclosing literal string
10	$	Right-associative operator meaning string-name
Logical operators		
11	&	AND
12	/	Exclusive OR
13	!	Inclusive OR

A string expression (se) is anything that can be evaluated to yield a string. String expressions are built up from one or more of the fields listed below. The value of the expression is formed by concatenating the values of the constituent fields.

FIELD	CORRESPONDING STRING
$(ne)	The string in dollar-line (ne)
'(string)'	(string) which may not contain '
"(string)"	(string) which may not contain "
(ne)	The value of (ne) converted to a string under control of the current output format and radix specification.
;	Carriage-return and linefeed
,	Has no value. May be used as a separator to resolve ambiguity.
%S(ne1) $(ne2)	The (ne1)'th substring of line $(ne2).
%C(ne)	The ASCII character formed by taking the value of (ne) modulo 128.
(ne1) @(ne2)	The value of (ne1) converted to a string in format (ne2).
(ne1) #(ne3)	The value of (ne1) converted to a string using radix (ne3).
(ne1) @(ne2)#(ne3)	Both, as above.

* * * * * *

Synopsis of BASYS Commands

Storage (null effect on execution)

ARRAY (name) (number)

sets up an array in the program line with (number) the highest subscript. The value of the line number is assigned to the simple variable (name) when the RUN command is executed enabling array elements to be referenced symbolically with the usual syntax.

$ (characters)

sets up a character string in the program line to be used as a string variable.

REM (characters)

sets up a character string for comments.

Assignment

LET (name1) = (ne1) (name2) = (ne2) etc.

assigns the value ne1 to the numerical variable name1, etc.

LET (array name) $\lessgtr$ (list of variables)

packs or unpacks the listed variables into or from the array - the list may consist of variable-precision numbers and variable length strings - in conjunction with READ and WRITE commands, this enables random-access mixed record structures to be set up on the backing store.

PUT (se) (search mode) (destination) (look for) (replace with)

generates the string se and decomposes it as specified - the last four fields may be iterated many times and execution of the command only continues as long as the search string (look for) is found - hence there is an implied conditional in the command and action may be taken if the decomposition rule cannot be applied.

Conditionals

IF (ne)

continues execution of the line if the value ne is non-zero.

UNLESS (ne)

continues execution of the line if the value ne is zero.

Many other commands have implied conditionals and continue execution of the line only if they have been performed satisfactorily.

Transfer of Control

DO (ne)

executes program line ne - control returns to line following DO commands.

GOTO (ne)

transfers control to line ne of program.

GOSUB (ne)

stacks the current line number and transfers control.

RETURN (ne)

unstacks the line number stacked by last GOSUB and assigns value to system variable QA - transfers control to next line with number greater than or equal to QA + (ne) - if (ne) is absent a value of 1 is used - if a command follows the GOSUB only the unstacking is performed and the transfer of control is not made. Hence variable returns and non-return transfers may be made from gosubs.

RUN (ne)

deletes simple variables, clears system stacks, gives system variables default values and transfers control to next line with number greater than or equal to (ne).

STOP

stops execution and returns to keyboard edit mode.

EXIT

stops execution and CALLs standard system program.

BYE

stops execution and logs user off system.

Editing and Housekeeping

LIST (ne1) (ne2)

gives formatted listing on channel 1 (default TTY) of program lines ne1 through ne2.

CLEAR (ne1) (ne2)

deletes program lines ne1 through ne2.

CODE (se)

reacts to the string se as if it had been typed in during the keyboard edit phase - enables program to compile additional lines at run-time, for example, dynamic arrays.

X (number) (look for) (replace by)

changes (look for) to (replace by) in programline (number) and prints changed line - primarily for swift editing.

CORE (ne)

adjusts user's core allocation until free space is at least ne characters.

GARB

collects garbage to maximize free space - normally done automatically when free space goes below limit.

Peripheral Transfers

BASYS input/output is largely device independent through numbered 'channels'. To minimize programmer effort standard device assignments are set up when a user logs in, but these may be changed ad lib.

INIT# (channel) (devicename)

> initializes the channel and attaches the named device.

RELEASE # (channel)

> closes the channel and releases the device.

CREATE # (channel) (mode) (filename)

> creates a new file for use in a certain mode and opens on a channel.

OPEN # (channel) (mode) (filename)

> opens an existent file for transfers in given mode on a channel.

DELETE # (channel) (filename)

> deletes a file from the device assigned to a channel.

RENAME # (channel) (newname),(oldname)

> renames a file - newname and oldname are string expressions.

PRINT # (channel) (se)

> writes the string se onto the prescribed channel - default is channel 1 (usually TTY).

INPUT # (channel) ?(ne) (PUT command decomposition syntax)

> if ?(ne) is present, PRINTS line $(ne) as a cue, otherwise PRINTS asterisk as cue - reads a line of text from device and decomposes it as for PUT command. Note that command defaults to simple list of numeric variables or character string input.

TAB # (channel) (ne) (se)

> print the string se repeatedly until the column ne is reached - used to tabulate and produce charts on TTY.

WRITE # (channel) (array name)

> write contents of array in line ne as binary block on channel - used for record structures in random access files.

READ # (channel) (array name)

> reads block of binary data from channel into array.

SAVE # (channel) (filename) (nel) (ne2)

> saves program lines nel through ne2 of current program as a text file.

CALL (start)#(channel) (filename)

> calls the specified program file and starts program at prescribed line - values of variables previously defined are retained.

DUMP# (channel)

> dumps the current program in core-image form.

LOAD (start)#(channel) (nel) (ne2)

> loads a core image created by DUMP and start at prescribed line - values of variables retained - lines nel through ne2 of old program also retained allowing arrays and strings to be passed. This command gives rapid overlaying.

The channel specification may be omitted normally when the standard default channels are used. Additional input/output and supervisory commands will exist for most systems since it has been the policy to interface most operating system facilities directly to BASYS (with suitable protection where necessary).

APPENDIX 2 SUMMARY OF ED

ED is a line-by-line string-oriented editor suitable for implementation on any computer having a block addressable storage device. At any moment during an editing session one line of the user's file is accessible for editing. We may think of a symbolic pointer, the current line pointer, which designates a particular line of the file. Editing consists of moving the pointer to the required line, and then making a change. The pointer may be moved freely up and down the file, and any amount of text may be deleted or inserted at any point. The necessary file handling and buffering are done automatically by ED in a way that is transparent to the user.

File Control Commands

IN filename	Opens an existing file for editing.
OUT filename	Creates a new file. If used inconjunction with IN, defines the name of the edited file.
LIST filename	Opens an existing file for examination only.
CLOSE	Finishes editing and restarts the editor.
EXIT	Finishes editing and returns to the monitor.

File Backup Measures

It is important that the file being edited should be adequately backed up, both to protect the user from his own mistakes, and as a safeguard against system failures and telephone disconnections. In addition to preserving the initial state of the user's file, ED also keeps a second, up-to-date backup copy:

SAVE	Saves the current state of the edited file.
BACK	Cancels all editing done since the last SAVE command.

Editing Commands

Some commands take numeric arguments. If these are omitted a default value of 1 is used. Commands that search for strings or move the current line pointer normally cause the pointer to step towards the bottom of the file. If the command is prefixed with the letter U, the movement will occur in the upwards direction towards the top of the file.

Commands for Printing

P n	PRINT n lines.
PF string	PRINT-FIND. Print down to line beginning with the string.
PL string	PRINT-LOCATE. Print down to line containing the string.
PB	PRINT down to the BOTTOM of the file.

Commands for Positioning the Pointer

T	Move pointer to the TOP of the file.
B	Move pointer to the BOTTOM of the file.
Nn	NEXT. Move pointer down by n lines.
Un	Move pointer UP by n lines.
F string	FIND the next line beginning with the string.
L string	LOCATE the next line containing the string.

Commands for Editing the Current Line

R string	REPLACE the current line with the string.
A string	APPEND the string to the end of the current line.
I string	INSERT the string below the current line.
UI string	Insert the string above the current line.
qC/string1/string2	CHANGE the q'th occurrence of string 1 to string 2 in the current line.
X/string1/string2	Change all occurrences of string1 to string2 in the current line.

Commands that combine Searching and Editing functions

LC/string1/string2	LOCATE next line containing string1 and CHANGE it to string2.
LD string	LOCATE next line containing the string and DELETE the line.
DF string	Delete down to the next line beginning with string.
DL string	Delete down to the next line containing the string.
qM/string1/string2	MACRO. Change all occurrences of string1 to string2 in the next q lines.

Commands for inserting large amounts of text

λ	Enter input mode. In input mode everything that is typed is inserted into the file upto the next ESCAPE character.
Vn	Move pointer up by n lines, then enter input mode.
On	OVERLAY. Delete n lines, then enter input mode.
OF string	Delete down to next line beginning with the string, then enter input mode.
OL string	Delete down to next line containing the string, then enter input mode.
Y filename	YANK. Insert whole of specified file below current line, and advance pointer to bottom of inserted text.

Multi-file Capabilities

ED can support two output streams concurrently, enabling text to be diverted from the main file to an auxiliary file. The user may switch the output from one stream to another at any time, allowing the input file to be segmented in any desired way.

PUT filename	Switch output to auxiliary stream.
OM	Switch output to main stream.

ED can also support two input streams, enabling the output file to be built up from parts of other files.

GET filename	Switch to auxiliary input stream.
IM	Switch to main input stream.

The GET and PUT streams may be used in combination to perform complex block-move operations. This is especially useful when rearranging machine-code programs for computers with limited addressing capabilities.