Which programming languages for minicomputers in process control?

V. HAASE
UNICOMP, Blankenloch, W. Germany

A

Computer involvement in industrial and scientific process control shows a very significant increase in the application of very small computers. This is due to the fact that hardware costs are falling, whereas the cost of designing, installing and maintaining special-purpose systems is not becoming lower. This calls for general-purpose electronics in measurement and control: the minicomputer. This is the reason for the fact that this end of the computer spectrum is not populated so much by small brothers of medium- or large-scale real-time computer systems, but by special systems designed for – and in many cases by – the engineer used to working with special-purpose control hardware. This hardware is now replaced by a 4 to 16 K, 8- to 24-bit machine with storage cycles between 0.5 and 5 μsecs, in most cases capable of, but not necessarily equipped with, all kinds of conventional and process-control capabilities which medium-scale computers have as peripherals. The whole thing costs no more than a special-purpose hard-wired system.

Nevertheless, the lack of relationship between minis and larger systems often causes a software gap between minis and midi- resp. maxi computers. Hardware-mindedness of both designers and users, and the fact that minis are used in environments which are not frequently changed, have led to the development of special-purpose software (instead of the special-purpose hardware that existed before!) – nonmodular code including both operating system and application program functions.

Programming was (and is) done in some kind of assembly language. In the course of time, here and there, FORTRAN and BASIC compilers or interpreters were developed (non-real-time-applicable) and the idea of indirect programming came up (minis were used more in changing tasks, laboratories, test-stands, etc.). As the indirect method is of little use if a larger computer is not at hand (which is often the case if a program has to be changed in the field), it is necessary to think about programming languages and compilers to solve real-time problems with small computers.

B

There is an extensive literature on real-time languages, but little implementation experience. The minicomputer has attracted few papers. We shall therefore try to express the apparent contradiction between the (unpublished) feelings of computer scientists and the experience of users, namely, that both kinds (small and big) of real-time computers should not be programmed in the same way, at least not using the same higher level language.

There exist special types of (real-time) programming languages that are especially suited for compilation and execution on minicomputers. They will be explained and proved as follows.

If we look at *higher level problem-oriented special-purpose* programming languages that could be used on minicomputers in process control, we may distinguish several groups of languages:

1. Macro assemblers.
2. So-called 'low-level' languages (e.g. PL/360).
3. Real-time dialects or subsets of procedural languages (e.g. R-T-FORTRAN).
4. System writing languages (e.g. POLYP).
5. Special real-time-languages (INDAC, PEARL) which may also be based on other languages (e.g. PAS).
6. Problem-oriented languages (e.g. fill-in-the-blanks language).

On the other hand we may lay down criteria (functional requirements) for real-time languages:

a. Easy to use.
b. Machine-independent (at least, 'controllable' machine-dependent).
c. Effective.
d. Timing in 'programmer's hands'.

among others. We add for the mini:

e. Executable.
f. Compilable on a machine with less than 16K working store (no mass store).

Everybody knows that items b. and c. (especially in connection with e. and f.) are contradictory. We therefore have to find a compromise matching the lists of language types and requirements:

1. Macro assemblers are effective (c), pro-

vide timing-control (d) and cope with the mini-requirements (e, f); they are not machine-independent ($\bar{b}$) and not always easy to use ($\bar{a}$).

2. Low-level languages can be designed to fulfil criteria a., c., d., e. and f., but they are explicitly machine-dependent ($\bar{b}$) and therefore, for example, very useful for system writing.

3. Real-time dialects and subsets of FORTRAN and similar languages have shown that they are not able to solve both real-time *and* efficiency problems, i.e. inherent problems of batch processing languages ($\bar{c}$, $\bar{d}$). Perhaps this could be achieved, but not in mini-environments.

4. System-languages are sometimes claimed to be the solution for process control and any real-time problems. They are, in fact, but only in the hands of very clever programmers. However, ease of use ($\bar{a}$) and timing and tasking features are not ideal ($\bar{d}$).

5. Newly designed real-time languages would, of course, be the best solution (if they fulfil the functional requirements). The problem of generality versus efficiency *can* be solved for larger systems, including backing storage (INDAC, PEARL), but for minis this is not possible. The design concept has to be modified so that both assembly type and procedural type languages are combined in a new language. Should this lead to a modular structure of the language itself, implementation costs could also be reduced (PL/1 + PAS/1, and PROCESS-BASIC, to be introduced here).

6. Special problem-oriented languages will not be considered here, as they are no common solution for real-time problems (and in many cases also designed in a descriptive type: no timing = $\bar{d}$).

C

If we combine the efficiency of an assembly language, handy macros for handling real-time functions (timing, interrupt-handling), and the machine-independence and ease-of-use of BASIC or FORTRAN – the whole of which can be compiled and executed on a mini – we obtain the language we are searching for. Of course, these properties cannot be present simultaneously in every language element (under our marginal conditions), but we can try to have them in an additive structure. This seems to be sufficient in the majority of cases, e.g. one frequently needs a special algorithm for data reduction (that is present in a higher level language library), but not the I/O driver for a special device used in some other installation (that has to be programmed very efficiently).

My solution for a programming language for minicomputers in process control is a composite language, the elements of which come from:

1. A procedural language (BASIC) that is easy to learn, use and compile.
2. An assembly language, covering both machine instructions and macro-instructions (if it is sufficient, the normal assembler of your machine).

It is necessary that these two components can be mixed at the statement level – not only at the module (= linkage editor) level – and it is to be provided that user-named items (data and labels) can be defined in one and used in the other statement type.

C1

Since these ideas came up when a special project was considered (the usefulness of a process-oriented programming language to be implemented on the UNICOMP 201) some aspects of this machine will be mentioned.

UNICOMP 201 is a 20-bit machine with working storage expandable up to 32K. Since the limited instruction repertoire, as far as arithmetic is concerned, is balanced by a great variety of operation- and addressing-modes (system-user states; literal, working store, indirect, external store and 'execute' operands), real-time programming can be done quite nicely.

A convenient macro-assembler makes use of 'supervisor-call' instructions, causing the execution of a subroutine package present at runtime that can be implemented as a 'firmware' module (a fast read-only-memory). These macro instructions include both fixed point (multiple precision, too) and floating point arithmetic, and interrupt and I/O handling instructions. For the design of a language with the desired features, the algorithmic part must be replaced by a common procedural language. We have chosen BASIC for reasons of easy learnability and compilability.

To improve efficiency, BASIC had to be expanded by a data-type INTEGER (INDEX) that is used for counting and field addressing operations. As labels are represented by line numbers in BASIC also, the assembly type statements of Process-Basic are allowed to have and to refer to integer labels. The name-syntax of BASIC is not changed (for compatibility reasons), so that all names in assembly type statements that do not consist only of one letter and one number are unique for the assembly part of the program.

The three levels of Process-Basic statements comprise:

Level 1 (BASIC):

INPUT, READ, PRINT, DATA, RESTORE	Input and Output
GOTO, IF, FOR, NEXT, GOSUB, RETURN, STOP, END	Control
DEFFN, DIM, INDEX	Definitions
LET	Assignment

Level 12 (Macros):

OUT,OTN,INP,INN,OSM,ISM,OLS,ILS,KTL,PLA,SAZ:

I/O Operations done in parallel or sequentially, with or without formatting, using symbolic or direct addressing.

ISS,ISL,RIT:

Interrupt answer definition, enable, disable.

Multiple precision arithmetic, block transfers, text editing and test instructions are also handled on macro level.

Level 13 (machine instructions):

Operations			Modifications
BRI	(load)		
UMS	(store)		
ADD			literal
ADU			working store address
KON	(and)		indirect address
EOR		X	I/O address
ERH	(+1)		execute
VER	(-1)		supervisor call
SPR	(jump)		
UPS	(subrout.)		
SAN	jmp =0		
SUN	jmp ≠ 0		
SKU	jmp car =0		

All three levels of statements can be mixed line-wise as shown in the following example:

```
      .EQUAL. TIMER='006'      device definition control statements
      .EQUAL. ADC1 ='5F1'      on assembler level
100   FOR I=1 TO 10
110   BRI 30 ADC1              loop coded in BASIC intermixed with
120   UMS 22 I                 machine code
122   NEXT I
      .PSW. '0100'             address of an interrupt routine
200   BRI 30 TIMER
210   DIV 00 333               macro instruction, time converted to
220   UMS 20 T1                secs
      :
      :
```

C2

The actual software (as planned) will be based on the existing UNICOMP software; it will consist of operating system, compiler, linking loader, test and text-editing routines.

The operating system (2 to 4K) is modular in the sense that a list-driven event (interrupt)-handling routine, the device driving package and supervisor-call-interpretation macros can be tailored by the user according to his special purposes. The compile-time-OS can be different from the execution-time-OS (the former including more string handling functions, the latter real-time macros).

As for the compiler implementation for small machines, two philosophies are applicable. First, for very small core size two pass systems are to be preferred; the object code will be punched. Certain on-line text-editing is possible, but no incremental compilation. The object code may be both relocatable and linkable. Secondly, if one can afford to hold both dictionary and object code in core, one pass systems which allow incremental compilation can be preferred. The object code can be either executable at once or be used as input into a linkage editor.

The compilers will be very similar, as the structure is defined by the general syntax parser. Directory- and code-generating routines are used as subroutines by the general routine. Certain optimisation (in the field of space versus time-efficiency) can be performed in that the code generator has two possibilities which can be chosen by the programmer: more in-line or more subroutine-like code (e.g. loop-heads, if-statements). The advantages of BASIC can be seen in the general expression definition, the name syntax allowing a directly addressable directory and the very clean statement type recognition.

The type of language described here is thus a good solution, both for the implementer (good price/performance ratio) and the user of a small machine (handyness and flexibility).

1. KEMENY, KURTZ, 'BASIC'.
2. FRÖHLICH, 'SAMMI, assembly language for UNICOMP 201'.

Discussion

Q. Why must the compiler run on the 8K machine?

A. Frequently the computer is out in the field at some plant, and the users will wish to be able to do their own program development. Also they may not have access to bigger machines.

Q. Will not the advent of satellite computers alter this?

A. It will still be awkward if you have to go to another computer which is remote.

Q. Line by line interleaving of assembly language and high level language statements implies that the compiler cannot attempt inter-statement optimisation. Do you accept this reduction in efficiency?

A. Yes, we are much happier that the programmer is aware of this and has to deal with inter-statement optimisation explicitly (e.g. by use of INDEX) than that he leave this to the compiler. We allow a testing phase using an interpreter to give debugging facilities, with only tested parts of the program compiled.