Experiences with languages for real-time programming

D.R. FROST
General Electric Co, Phoenix, Arizona, USA

At General Electric's Process Computer Operation in Phoenix our product consists of real-time process data acquisition and control systems, with applications including the automation of power plants, power distribution, refineries, chemical plants; such processes as discrete parts manufacturing, steel making; production of cement, non-ferrous metals, paper, textiles, and others. For these automation systems we provide the computer and process input/output. We frequently provide analysis and programming as well. Since economics dictate that we will get no more money for this service than it is worth to our customers, profitability depends upon our skill in keeping programming costs down. This paper is about our attempts to lower costs through the use of higher level programming languages and application packages. I will discuss:

1. What we expect from higher level languages and application packages.
2. Our specific experiences with these various types of languages and application packages and how successful they have been.
3. Where we are headed.
4. Lessons we have learned.

Expectations

The whole rationale for using higher level languages and application packages is to save system costs through lowering programming costs. There have been many successful tools for various other computer applications: FORTRAN, ALGOL, COBOL, LISP, various report generators, sort packages, and simulation languages are just a few. All of these tools have a common objective: to produce machine-independent source programs.

Machine independence is often equated with portability between computers. Yet the truly important facet of machine independence is not so much portability; of more importance, it allows the user to implement his solution without all the computer-related clutter that gets in his way. To a problem solver this clutter is overhead, with all of that word's connotation of cost – and he is right. He wants independence from the computer.

A good programming tool:

1. Gives the programmer a vocabulary that matches his application.
2. Takes care of hardware
   - storage allocation
   - controlling peripherals.
3. Provides
   - standard algorithms
   - data formats
   - character codes.
4. Promotes good design, and allows good design to be maintained as the system gradually evolves.
5. Has self-contained debugging aids, which
   - make mistakes less likely
   - make error correction more reliable.
6. Gets rid of 'patches' [machine-oriented program corrections] and the dependence on knowledge of the computer which 'patching' requires.
7. Aids maintenance of a design as the application changes.
8. Aids maintenance of code as errors are found and corrected.

Good design is probably the biggest cost saver in software production, and good tools encourage good design. An excellent example is the successful language COBOL, which forces the programmer to define and document his data base. (I suspect that this is actually more a key to COBOL's success than is its natural English-language-like syntax.)

Successful languages and application packages such as the ones mentioned exist in the comparatively orderly world of scientific and business data processing. They fit their applications very well. FORTRAN is a good language for mathematics and logic. The vocabulary of COBOL consists of business data processing terms in common use long before computers. The environment of a sort generator package is orderly files of data. But the term 'real-time programming' causes many of us to think simplistically. It makes us believe we will find a single, ultimate tool for 'real-time programming'. But real-time applications are just as diverse as those for which we use LISP, FORTRAN, and report

generators. As a result, we are finding and will continue to find a multiplicity of 'best' tools for real-time systems, depending on the particular applications.

Good tools for real-time programming must provide the same advantages as do existing successful tools for other applications:

1. Independence from the computer.
2. A problem-oriented vocabulary.
3. Encouragement of good software design.

Let us now look at some of the tools that have been used by employees and customers of General Electric's Process Computer Operation.

Our experiences

FORTRAN – a semi–success

Soon after FORTRAN swept to supremacy as the earliest and most used mathematical programming language in the United States our customers wanted to use it to program their process applications. In the early days there was considerable naïveté about the usefulness of FORTRAN for real-time programming. It 'seemed natural' that its acceptance for mathematical applications would be duplicated immediately in process control applications. Yet even now, more than ten years later, its use is light compared to assembler language. Without considerable work it would have been a total failure (and was for some time).

First, let us look at its shortcomings:

1. Real-time – FORTRAN has no capability of recognising the passage of time or of responding to real-time data arriving at unpredictable intervals and in varying volumes.
2. Data types – it does not handle large numbers of logical variables efficiently.
3. Data communication – COMMON storage (as used in FORTRAN) is monolithic: it provides total communication or none, which works against good program design. Besides, data bases are seldom small enough to fit into one COMMON pool in main memory.
4. Program organisation – FORTRAN's subroutine method does not provide control linkage or data communication between segments operating under a real-time operating system.
5. Input/Output – Formats fit neither the needs of process input nor communications with operator consoles and other specialised devices.

In spite of these shortcomings, FORTRAN is in use today. In some companies (and some application areas) it is used a great deal. This is possible because of various attempts which have been made to remedy FORTRAN's shortcomings. These various attempts have resulted in a proliferation of FORTRAN dialects and extensions. We have two ourselves, one predating the FORTRAN ANSI standard, the other based on it.

The real-time problem is solved through providing a real-time operating system outside of FORTRAN. System functions or subroutines are made available to the programmer for interrogating time, which is kept by the operating system.

Data types are augmented by allowing various forms of bit arrays and providing operations to work on them.

Data communication problems are also handled through various extensions to the language. In one case we changed the meaning of labelled COMMON to allow more than one common area. Named system variables are another useful technique.

Program organisation is dependent on the available operating system. A program with its own subroutines is usually (but not always) considered a program segment to the operating system. Special system subroutine calls can be used to schedule other program segments.

Input/Output is a very troublesome area. FORTRAN I/O is based on such a rigid approach that many improvements have been attempted, but most have proven unsatisfactory. For example, the idea of sharing a peripheral is foreign to FORTRAN. This causes a problem when very large operating logs must be collected and stored, then put out on a typer without being interrupted by other output requests to the same typer.

There is a hopeful development: as a result of the Workshops on Standardisation of Industrial Computer Languages held twice yearly at Purdue University, a standard for FORTRAN extensions is being worked out. Although the semantics of some of these extensions are operating-system-dependent and cannot be standardised, the syntax can. It is predicted that this effort will be successful.

In our own shop the use of FORTRAN is mostly limited to power plant performance calculations. I believe the problem here is that trading off savings in program production against an increase in program size and hardware cost is difficult to 'sell'. Hardware costs are easy to calculate, but software costs are nebulous; it is not always easy to demonstrate the advantages to financial management.

Therefore, I have to consider FORTRAN as being relatively unsuccessful.

PERCON and SCANCON — application package failures

PERCON (peripheral control) and SCANCON (process input control) were two application packages intended for our first process control computer with core memory, the GE-412. PERCON was really no more than a typical peripheral I/O package. SCANCON was a table driven process scanner much like that now in SEER.

These two packages never got off the ground. After considerable design work they were killed before coding began; thus we will never know their merits. They failed because they were conceived and designed by a group that was not actually involved in the application programs; and this is fatal. Analysis and design work must be done by those who will use the package. There are two reasons:

1. They probably know better what to do.
2. Even when they do not know, you have a nearly impossible political job in selling the package to them.

COOL — a language that failed

Even in the early days it was obvious to us that FORTRAN was not the right language for process computer programming, so we tried our hand at inventing one. COOL (Control Oriented Language) was a dialect of WIZOR, a little known (but excellent) language for system programming available on the GE-225 general-purpose computer. Although it looked like other computer languages, control of almost everything remained in the hands of the programmer. Its strong point was that it gave the programmer a good syntax for operations on scaled fixed point and logical variables. The compiler translated COOL into assembly language for input to the assembler.

The language was greeted with enthusiasm by our application programmers (a study had been made of the language before the translator was implemented, both for technical and political reasons), and it was immediately used. But it fell into disuse quite rapidly. What had gone wrong? Several things:

1. The compiler existed on our general-purpose computers only, which made language patching in the field a necessity.
2. With assembly language accessible to the programmer, re-assembly without recompilation was tempting, and this quickly made the COOL statements obsolete, indeed misleading.
3. Since COOL statements quickly became obsolete, they saved time only in the initial coding of programs, which is certainly a small part of the programming job. (Our study reported positively on the language mostly because of savings in coding time; we just did not then realise what a small part of the total work coding really is.)
4. We failed to do a good job of training the programmers in the intelligent use of the language.

Our lesson here is that coding aids are not enough, and training is essential. A macro assembler failed later for exactly the same reasons.

MACRO assemblers — abortive attempts

Macro assemblers have been frequently proposed, and one was implemented. It was written by an application programmer for application programmers, which is an approach that should lead to success. Yet it was little used. It failed for the same reasons that COOL failed. It did not do enough for the programmer, and reassemblies frequently bypassed the macro processor, and there was no training of any consequence.

SEER — a big success

As power consumption in the United States has mushroomed we have been providing a great many process computers for power plant monitoring and control. An application package for power plant monitoring, SEER, has been our most successful tool for saving programming costs. It is a system of functional modules for data scanning, logging, alarming, and console input and display. It also includes support routines for performance calculations. It provides a base system which can easily be expanded to include further plant monitoring as well as plant control.

Several factors contribute to its success:

1. All steam power plants are pretty much alike; thus we had an application narrow enough to fit an application package to.
2. Programming and hardware both are nearly always purchased by power companies from a limited number of traditional instrumentation suppliers; thus programming has not been fragmented among many teams of programmers.
3. SEER evolved slowly, starting as individual programming jobs where we carried program design and code from one job to the next. We thus learned to understand the application very well.
4. It was invented by the same group which was going to use it.
5. The programmers using it knew the software as well as the application, and could

thus use it intelligently.

6. Because of our heavy participation in the market, our approach has had good acceptance from major engineering firms that design plants for the utilities.
7. We waited until we had a *de facto* near standard before we standardised the application package.
8. A team of programmers with power plant experience was assigned to write the standard, guaranteeing political acceptability as well as continuity of knowledge, experience and technology.

Although our exact costs cannot be divulged, it can be said that, working in ratios, our first plant monitoring system took about 18 units. Before the final standardisation effort, the time was down to 3 units. We can now, with our present standard application package, produce one in a single unit of effort. (In all cases this does not include special unique application software.) We are now developing a SEER for the GE-PAC 3010 based on experience with three previous process computers, this time confidently without the trial-and-error approach.

TASC – an unsung hero

Although monitoring power plant operation is quite standard, control, especially in sequential operations such as start up and shut down of the boiler and turbine, is less so.

Early in our history one of our programmers invented a sequential control language called TASC (for Tabular Sequence and Control). TASC is a typical interpretive system with

1. A sequential control language.
2. A translator of TASC statements (one-to-one) to a compact code.
3. A simple interpreter which, based on TASC statements, selects subroutines for execution.
4. A library of subroutines which actually effect the control actions specified in the language.

The language and translator have remained much the same functionally through the ten years of their existence and through several computers. The control subroutines usually require some rewriting from job to job.

This is called an unsung here because it has never had much publicity to our customers or even our own management. It continues to be one of those valuable tools that no one much knows about except the programmers using it from day to day.

Direct Digital Control – success or failure?

When discussing 'direct digital control' one immediately runs into a semantics problem. The phrase has two meanings:

1. Digital (as opposed to analog) control of a process element in any kind of application.
2. An application package for control of continuous processes only, in which analog controllers are replaced with digital controllers.

The latter definition is used, because it is the application for which our ddc packages have been written.

Direct digital control was originally conceived as a money saver. It seemed that considerable savings in hardware cost could be realised through replacing analog controllers with digital controllers, moving control calculations into the computer. Since this has turned out not always to be true, ddc has not become the control method that its advocates had desired and expected. Indeed, interest appears to be waning.

In addition to a good many special ddc systems, we have produced two generalised ddc packages; one a large application package for general use by customers who, unlike electric utilities, usually do their own application programming. The other is a special package for a specific customer who has us to do his programming for him and has given us considerable repeat business.

The former has not been particularly successful: either deficiencies in our package or lack of success of the concept (large ddc systems) might be the reason. I suspect the latter.

The "special" ddc package has just recently evolved to a state where it is becoming standard. We have realised the same kind of cost savings on recent systems as we did with SEER just before the final standardising effort.

Other Utility Application Packages – evolution

Just as SEER gradually standardised and the special ddc package is on its way to becoming standardised, other utility packages (for monitoring power transmission networks to provide system security and for economic dispatch and power generation control) are also evolving towards a standard.

For power transmission network monitoring (called SCADA) we have moved so far that we are now 'standardising' the package. Economic dispatch and power generation control packages will probably reach the same stage within one to two years. We are already realising savings through re-use of design and code. We are confident that true standardisation will lower our costs per system even further.

BICEPS – a successful supervisory control system

BICEPS is a large system for supervisory control

[where process variable setpoints are sent to analog controllers (or to a ddc system)] of continuous processes. It provides the engineer with forms which he fills out to describe measured variables and control loops; it also provides a simple language for programming his control algorithms. Its files are available to programs for further optimising and control calculations. (These programs are usually written in FORTRAN.) It was designed in cooperation with the same company that had previously been instrumental in the design of IBM's PROSPRO system for the same application. We have delivered it to several other customers as well as to the original customer.

In my opinion, its success may be credited to the fact that supervisory control still remains the main reason for using a computer on a continuous process, and that this application package fits supervisory control well. (Note, however, that one can expect application packages like this to increase system overhead. The laws of hardware/software trade-off apply.)

What next?

From the evidence of our own successes and failures it appears that application packages have saved us considerable money, while new languages have done little for us.

Actually, the dividing line between languages and application packages is hazy at best. Thus the elusive problem-oriented 'languages' that the industry has been looking for are in truth taking the form of application packages like those we have been discussing. Consider the important characteristics of both:

1. Machine independence.
2. A syntax for describing the problem solution that
   - fits the problem
   - the problem solver can use with ease.

Although the syntax used with an application package does not look like a language (i.e. it is not algorithmic), I maintain that the differences are only syntactical and that we should not worry about what form our successful 'language' takes. It is the benefit that counts.

Therefore, my conclusion is that for applications where standardisation is feasible, we at General Electric will continue to define application packages to lower programming costs.

Unfortunately, not all applications can be standardised, usually because the number of similar applications is too small to make it economical. For these we must continue to search for an algorithmic language better than FORTRAN. FORTRAN is merely a stopgap until a really good algorithmic language comes along to take its place – and there are candidates coming from all directions. At present we are looking to the various system implementation languages that are being invented.

Earlier the necessary attributes for success were stated:

1. Machine independence (as it was defined then).
2. A problem-oriented vocabulary.
3. Encouragement of good software design.

Good implementation languages allow one machine independence when one wants it, but let one get as close to the hardware as one wishes when one needs to. Although an implementation language cannot give 'the best' problem-oriented vocabulary (since it will be applied to diverse problems), it can at least give a better algorithmic vocabulary than FORTRAN, and a good implementation language will encourage good program design.

We are working on one now.

Lessons we have learned

To sum up, our nearly fifteen years' experience in searching for better programming tools has taught us the following:

1. An application package can be a real money saver where there is a large enough population of jobs for the package.
2. Application packages are best developed by the people experienced and actually involved in the application. Yet members of this group must be set aside and be dedicated to final standardisation independent of any one specific application.
3. Do not expect standards to be final. They are subject to evolution.
4. The ideas for any languages designed for use by application programmers should come from them. As a poor second choice, there must be a careful, early study conducted by the system programmers to discover the application programmers' real needs. This study serves two purposes – it increases the probability of technical success and it helps to 'sell' the tool. And make no mistake, the acceptance of a tool by those destined to use it is one of the most critical requirements for success!
5. The programmers who will use a tool must be thoroughly trained in both the application and the software itself.
6. FORTRAN with appropriate extensions can help if one is willing to make the necessary hardware/software tradeoffs. Yet it is not really the right algorithmic language.
7. None of the candidates (and there are many) for the 'right' algorithmic language has yet proved successful (that is, none has become a widespread standard).
8. Tools that help merely in the coding stage of programming are not worth the trouble.
9. Any language should have support so that its translation can occur at the application site. Otherwise much of the expected savings fail to

materialise.

10. Good software design vs. poor software design will make more difference in programming cost than the language selected (except where the language fits the application particularly well).

11. An application package must be modular; considerable effort should be made to separate the relatively stable elements from those that will likely change from job to job.

For FORTRAN see:

- American Standard FORTRAN
- American Standard Basic FORTRAN
  both available from:
  United States of America Standards Institute,
  10 East 40th Street,
  New York, N.Y. 10016,
  USA.
- 'Clarification of FORTRAN standards – initial progress', ACM Communications, V12(5), (May 1969).
- 'Clarification of FORTRAN standards – second report', ACM Communications, V14(12), (October 1971).

For the Purdue proposed extensions see:

- Minutes of the Fourth Workshop on Standardization of Industrial Computer Languages (pp. 71 ff) available from
  Purdue Laboratory for Applied Industrial Control,
  Purdue University,
  Lafayette, Indiana 47907,
  USA.
  There have been minor amendments which have not been published. This work will result in an Instrument Society of America Standard.
- Also obsolete in some of the details but adequate to give you a feel for the style of the extensions is KELLY, E.A., 'FORTRAN in process control: standardizing extensions', Instrumentation Technology (May 1970).

For information on some of our application packages see:

- SEER System (GEA 8465)
- BICEPS Summary Manual (GET 3559)
- GE-DDC Summary Manual (GET 3558)
  also available from General Electric in Phoenix.

For discussions on the functional requirements for Industrial Computer Systems see:

- Minutes of the Fifth Workshop on Standardization of Industrial Computer Languages, also available from the Purdue Laboratory for Applied Industrial Control (see above). This work has been summarized in CURTIS, R.L., 'Functional requirements for industrial computer systems', Instrumentation Technology (November 1971).

For further discussions on the applicability of FORTRAN for process control see:

- FROST, D.R., 'Fortran for process control', Instrumentation Technology (April 1969).

For a good survey of process-oriented languages see:

- PIKE, H.E., Jr., 'Process control software', January 1970 Proceedings of the IEEE, a special issue on computers in industrial process control. This same issue contains many good articles, including SCHOEFFLER, J.D., and TEMPLE, R.H., 'Real-time language for industrial process control' – (a proposed algorithmic language).

For General Electric's FORTRAN see:

- GE-PAC 4000 Process FORTRAN Language Manual (PCP 122)
- GE-PAC 4000 FORTRAN IV Language Manual (GET 6051)
  both available from:
  General Electric Company,
  Utility and Process Automation Products Dept.,
  Technical Publications,
  2255 W. Desert Cove Road,
  Phoenix, Ariz. 85029,
  USA.