Choosing a standard high-level language for real-time programming

M.G. SCHOMBERG
AERE, Harwell, England
and
P. WARD
Plessey Radar, The Plessey Co, England

Introduction

This paper is concerned with the problems of providing high-level language facilities for a real-time project. It covers particularly those features necessary for a real-time applications programmer and does not consider the rather more special problems for the systems software programmer.

A large number of real-time projects are programmed entirely in assembly language for a variety of reasons. These range from necessity, due to space and time requirements, to simple prejudice. This paper assumes that the use of a high-level language is justified for the project.

Section 2 of this paper considers the alternative approaches of using existing languages or writing special ones. Section 3 identifies the special features necessary for a real-time language and how they can be achieved and in Section 4 a method of selecting the most appropriate language for a hypothetical project is demonstrated.

General-purpose or special-purpose languages

There seem to be two main schools of thought concerning the selection of a suitable high-level language for real-time projects. The first of these, which comes from the more academic circles, implies the development of a special language for almost every individual problem. The alternative view, supported by this paper, favours the use of existing high-level languages whenever possible.

The 'academic' point of view is based on the use of the programming language not only as the means of communicating with the computer but also the framework for thinking about the solution of the problem. On this basis they argue that an apt language will lead to a good solution, while an inappropriate one may lead to a poor solution. In addition, the special language for the particular situation includes within itself standard solutions to some of the fundamental operations required to formulate the solutions. Thus the programming language points the way to the solutions by already having solved some problems and laying the foundations for the solution of the rest.

The 'practitioner' point of view rejects the conclusions of this argument, not by denying the facts but by pointing out their characteristic of only marginal improvements in comparison with more general-purpose languages, and considering other problems and constraints of greater weight. The time scale for development of a new language is long (from initial ideas, constructing the compiler, testing and assessing the value of the new features). A new language also implies re-training of programmers and the writing of user manuals. Consequently it is not usually practicable to undertake to do it within an already tight time scale for a development project. Furthermore, the solutions to the problems involved will be discovered by people working in the project with their individual prejudices, experience, capabilities and potentials exerting a much stronger influence than the programming language. A man using a new programming language is most likely to continue with design habits acquired during his previous experience with other languages: thus the potential benefits of a special-purpose language are weakened, and the effort which would have to be expended to develop one is not judged to be justified.

Real-time features

This section identifies the special features associated with real-time programming and then attempts to show how they can be realised in more general-purpose languages.
These are divided into three categories.

1. **Time-dependent features**: those features which control and monitor the dynamic system state. They include:
   - I/O control
   - resource allocation
   - storage control
   - interrupt handling
   - re-entrancy
   - parallel processing
   - non-sequential operations

2. **Time-independent features**: those features which are mostly found in the solution of real-time problems but which are not strictly time-constrained. They include:
   - byte handling
   - list processing
   - in-line insertion of code
   - bit addressing
   - string handling
   - in-line insertion of code

3. **General features**: those features which are not only important in the real-time field but also in other fields of computing. They include:
   - modularity
   - ease of use
   - macros
   - ease of program maintenance
   - well established

The last three, although not strictly language 'features', are included to bring out the importance that is attached to these aspects of a language.

In the above it is only the time-dependent features which require communication with the operating system. It is therefore assumed that the operating system provides the necessary 'hooks' to perform such operations as the scheduling and suspension of tasks and resource allocation as requested by the 'user' programs.

**How these special features can be provided**

Having established what special features are necessary for programming a typical real-time application it is possible to examine the means by which they can be provided.

Consider the problem faced by the programming manager about to undertake the implementation of a real-time project. He has a computer, a general-purpose operating system and one or more general-purpose high-level language compilers.

He must first identify precisely which real-time features are necessary for the project. Those which only the operating system can provide will entail either modification to the existing operating system to provide them or writing a new operating system. Those features which are independent of the operating system are discussed later.

A means must then be found of making the additional features of the operating system available to the programmer. There are three possibilities:

1. To provide the facility as an assembly code subroutine which can be called from the user program.
2. To provide a facility in the compiler whereby in-line assembly code instructions can be used. This facility can be given a high-level flavour by the use of macros.
3. To modify the compiler to provide a set of new language statements which ensure the necessary linking to either the operating system or to a library subroutine that performs the required function.

The first alternative of using assembly code subroutines is probably the easiest to implement but the subroutine calling sequence will result in some loss of object code efficiency. The other alternatives avoid this by generating in-line code. If the compiler permits the insertion of in-line code then this can be achieved by a macro pre-processor. The chief penalties here are a reduction in the general legibility of the source program and a loss of checking by the compiler. The desirability of enhancing an existing compiler to include new language statements can only be judged in each individual case.

**Time-independent features**

The above considerations lead to an unexpected conclusion. Those language features normally associated with real-time programming are not, in fact, the most essential features when assessing a language for a real-time project. Such features can usually be provided by other means without undue difficulty.

However, the language feature provided by a general-purpose language that is least likely to match the requirements of a real-time application is its data structures. The ideal data structure will probably be different from one real-time application to another and requiring any combination of list structures, multidimensional arrays, Coral TABLE structures or the COBOL hierarchical record structure, while, at the same time, being influenced by data retrieval requirements. PL/1 is the only language that approaches this degree of flexibility. However, in spite of this, real-time projects do get implemented and such problems with data structures do get overcome as they arise; apparently without too much difficulty. It is possible to describe data structures in a long-hand manner if suitable language facilities are not available.

To conclude this section, it can be seen that all the special real-time features, except perhaps data structures, can be provided without the need for special language development. The lack of suitable data structuring facilities makes the programmer's job more difficult but not usually...
impossible. Some of the currently available general-purpose languages provide adequate facilities which enable enhancement to the level necessary for most real-time projects. It is therefore suggested that the use of such languages provides the most cost effective solution to this problem in many instances.

Graded assessment of high-level languages

This section examines six commonly used languages for their suitability for a real-time project. They have been selected on the basis that they include the most widely used languages and represent a wide spectrum of applications. Others, e.g. JOVIAL, should perhaps be included for completeness, although the purpose of the exercise is to illustrate a method of assessing languages and not to do an assessment.

Method of assessment

In order to carry out an assessment of programming languages for any project it is first necessary to list all language features and other factors which have any effect on the application. This list of assessment criteria is then grouped together according to their relative importance. A typical grouping might be as follows: essential features, highly desirable features, desirable features and possibly useful features. Each group is then assigned a grade range. For example, the most important group is given a grade of 100, the next group the range 50-99, and so on. Within this grade range each of the assessment criteria is given a particular grade to show its relative importance compared with other criteria in the same group. Each language is then taken in turn and for each of the criteria is given a mark up to the maximum allowed for that criterion. This mark must reflect the capability of the language to meet the criterion compared with the other languages being assessed. Once the assessment table is complete it is possible to carry out a simple elimination exercise.

Any language which does not meet all the essential criteria can immediately be eliminated. By totalling all the marks for each language the comparison between them can be made. Hopefully, the language showing the highest over-all total is the best for the job.

Although the initial marking of features for a particular project must necessarily be subjective and based largely on experience, once this has been done the actual assessment process is probably fairly reliable.

Other factors, however, influence the final choice of language. Among these, and perhaps even more important than the language itself, is the availability of a suitable compiler for the selected computer. To have a professionally implemented compiler which has stood the test of time contributes more towards the successful achievement of a project than any number of features specially designed for the project. This is particularly so when such features can be provided, perhaps less elegantly, by other means. Full error checking and unambiguous error reporting, comprehensive de-bugging facilities and a good user’s manual are of prime importance in the use of a language in any project. These features can, of course, be included in the language assessment criteria.

Table 1 illustrates this technique for a hypothetical project. The main points that the table brings out are as follows

1. The relative merits of each feature to each other feature, e.g. it is more important that the language is easy to learn and use than it should have I/O statements.
2. How each language performs for each feature, e.g. FORTRAN and COBOL are easy to learn, whereas PL/1 is the most difficult.
3. Neither COBOL nor ALGOL can be considered for this project.
4. PL/1 is the most suitable language for the project.

This method is both simple to use and understand and can be designed to any level of detail required. It is fairly objective in its approach and enables the value and effect of new information on the choice of language to be readily assessed and leaves irrelevant arguments to be discarded immediately.

Conclusions

This paper has sought to show that in many real-time projects it is more appropriate to use existing languages than to develop special ones. It is suggested that most time-dependent features can be realised without too much difficulty and that it is more important to consider such aspects as a good implementation of an existing language which has the necessary data structures. The final section illustrates one way of selecting that most suitable language for a real-time project.

Discussion

C. I applaud your courage in presenting specific figures.

C. The results have little meaning, because the weights can be changed to manipulate the figures.

Q. What do you mean by dynamic storage in this context?
### TABLE 1  Language assessment for a real-time project

<table>
<thead>
<tr>
<th>Essential features</th>
<th>Max. points</th>
<th>Fortran</th>
<th>Cobol</th>
<th>PL/1</th>
<th>Algol 60</th>
<th>Algol 68</th>
<th>Coral 66</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modularity</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Bit /byte addressing</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Sub total</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>200</td>
<td>0</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Highly desirable features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data structures</td>
<td>85</td>
<td>0</td>
<td>85</td>
<td>85</td>
<td>0</td>
<td>85</td>
<td>50</td>
</tr>
<tr>
<td>Macro facility</td>
<td>85</td>
<td>0</td>
<td>0</td>
<td>85</td>
<td>0</td>
<td>0</td>
<td>85</td>
</tr>
<tr>
<td>Easy to learn and use</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>30</td>
<td>50</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Good documentation and maintainance</td>
<td>60</td>
<td>60</td>
<td>30</td>
<td>60</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Parallel processing</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>In-line code insertion</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>Usage</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Sub total</td>
<td>460</td>
<td>170</td>
<td>225</td>
<td>350</td>
<td>110</td>
<td>215</td>
<td>275</td>
</tr>
<tr>
<td>Desirable features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic storage</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>I/O statements</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>String handling</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>List processing</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>Floating point</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Sub total</td>
<td>170</td>
<td>60</td>
<td>100</td>
<td>170</td>
<td>60</td>
<td>130</td>
<td>60</td>
</tr>
<tr>
<td>Possibly useful features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free format</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Machine independence</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Recursion</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sub total</td>
<td>15</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>TOTAL</td>
<td>845</td>
<td>430</td>
<td>330</td>
<td>733</td>
<td>183</td>
<td>559</td>
<td>548</td>
</tr>
</tbody>
</table>

| Implementation features       |             |         |       |      |         |         |         |
| Compiler available            | 90          |         |       |      |         |         |         |
| Compiler usage                | 90          |         |       |      |         |         |         |

A. The program's facility of influencing storage allocation.

Q. How would you take into account the quality of the programmer?

A. We had this in mind in 'easy to learn and use'.

C. You should also take into account
   a) the experience of the programmer
   b) the size of the computer.

C. Learnability should be taken into account in language design.