A HIERARCHY OF HIGH ORDER LANGUAGES FOR SYSTEMS PROGRAMMING

Don M. Lyle
Burroughs Corporation, City of Industry, California

I. Introduction

The programming of systems software in higher level languages has been a subject of much interest and debate. At Burroughs the debatability of the issue has long since ceased to exist since both the operating system and the compilers for the B5500 were successfully implemented in variants of ALGOL 60. The low manpower requirements and the ease of maintenance and modification have caused all concerned with the project to accept this approach without question. This technique has the added benefit of producing more reliable software due to the greatly reduced number of lines of code required.

When the decision to produce a successor to the B5500 was made, there was no controversy over the use of high level languages but considerable debate over how high a level the languages should be. B5500 ESPOL (Executive System Problem Oriented Language), the operating system implementation language, contained many "unsafe" constructs, for example, the ability to directly address memory through a subscripted reference to an array known, strangely enough, as MEMORY. This allowed coding errors to produce undesirable side effects on the system. Examination of the algorithms used in most operating systems indicated that surprisingly few routines actually require such "unsafe" tools leading to the conclusion that the proper approach to the coding of the overall systems software might be a hierarchical set of systems programming languages with varying degrees of "safety."

The hardware facilities of the B6700[5] were designed with this fact in mind. For example, array bounds protection is provided in the hardware which the "user" programmer cannot circumvent and the "systems" programmer must consciously work to circumvent.

II. B6700 Extended ALGOL[1] as a Systems Programming Language

The object of B6700 Extended ALGOL is to allow the programmer access to all machine and operating system facilities within the constraint that his program may not adversely effect other programs not created by the current execution of his program. The flexibility of the language is illustrated by the fact that all B6700 compilers, including the Extended ALGOL compiler itself, are written in Extended ALGOL. The Extended ALGOL compiler compiles large programs at a rate in excess of 4000 card images per minute thus proving that it is possible for a compiler implemented in a high level language to be both reasonably fast and to generate reasonably efficient code. The claim that the code generated is efficient is supported by noting that the compiler, which runs at reasonable speed (and is therefore reasonably efficient), is an ALGOL program.

Certain extensions have been made to the ALGOL 60 language to facilitate its use as a systems programming language which, by the way, makes it a more powerful user language. These include, among many others:

1. Bit manipulation facilities

```
IF X.[5:2] = 3 THEN X.[9:1] := 1;
```

The above statement tests the two bits beginning at bit number five of the current value stored in the variable X against the value 3. If equal, then bit 9 of the value of X is turned on.

```
X := Z & Y.[31:20:15];
```

The value of X is replaced by the bit pattern at Z which is modified beginning at the 31st bit by the 15 bits beginning at bit 20 of the variable Y.

2. Recursion

The ALGOL language allows full recursion. If efficiently implemented, this provides a very powerful tool for the systems programmer. For example, in B6700 FORTRAN the following subscripted reference to the array X is allowed:

```
A = B + X(B + SQRT(X(I)))
```

This (possibly useful) extension to the FORTRAN language occurred due to laziness on the part of the implementor of the FORTRAN compiler. Subscripts are encountered frequently in the compilation of arithmetic expressions. Since the implementor was

coding the expression routine which must, of course, compile generalized arithmetic expressions, when he reached the point in the algorithm where he must call the subscript evaluation routine, he elected to call himself recursively rather than go to the trouble of coding still another specialized expression routine.

3. High Level Macros (Defines)

The define capability is provided primarily for documentation and maintainability.

```
DEFINE STATUS = [5:2]#,
       TESTPASSED = [9:1]#;
```

A define indicates to the compiler that when the defined word is encountered in a subsequent statement, a direct textual substitution of the text appearing between the equal sign (in the DEFINE statement) and the pound sign should be made for the tokens parsed by the compiler. With the above DEFINE the following statement produces identical code to that produced by the first example under "Bit manipulation facilities"

```
IF X.STATUS = 3 THEN X.TESTPASSED := 1;
```

Similarly, a DEFINE may be "passed" parameters at compile time. If the following DEFINE, as well as the two above, has been encountered during the compilation process,

```
DEFINE TEST (Z) = IF Z.STATUS = 3 THEN Z.TESTPASSED := 1#;
```

then the subsequent appearance of the statement TEST(X); will again produce identical code to that of the first example.

4. Inter-Program Communication

While the ALGOL 60 language specification does not allow separately compiled procedures, the EXTERNAL declaration for procedures in B6700 Extended ALGOL allows this feature. The system allows procedures to be explicitly "bound" to a host program prior to execution, implicitly bound at execution time, or even to be non-existant assuming that the procedure is not to be called during a given execution. While itself an important (and necessary) addition to the language as far as effective systems programming is concerned, EXTERNAL procedures provide an even more useful extension to the language when coupled with the multi-tasking facilities described below.

Possibly the most useful extension made to ALGOL from the standpoint of its use as a systems programming language is the incorporation of Inter-Program Communication (IPC) facilities. Software EVENTs are included which may be CAUSEd, WAITed upon, or tested to see if they have HAPPENED. Both hardware and software INTERRUPTs (similar to PL/I on-conditions) are allowed which may be ENABLEd and DISABLEd. Any given procedure may be invoked via a normal procedure call, as a coroutine partner, as an asynchronous process, or any combination of the three in any given program. Asynchronous processes and coroutines are assigned TASK identifiers by which they may be monitored and controlled through interrogation and assignment of their task attributes.

Task attributes include as a subset those familiar in the PL/I language. Some of the additional attributes include:

COREESTIMATE	Estimated core storage requirement of the task (used for system scheduling purposes).
MAXPROCTIME	Maximum processor time to be allowed before the task is automatically terminated by the system.
STATUS	Current status of the task: Scheduled, active, temporarily suspended, terminated.
HISTORY	Reflects cause of termination of a task: Operator terminated, arithmetic fault, etc.
EXCEPTIONTASK	Identifier of the task to be notified of changes of status of this task.
NAME	By assignment to the NAME attribute of an inactive task and the subsequent activation of that task a program may refer to

any program or procedure known to the system (compilers, user programs, etc.).

By appropriate manipulation of task attributes the programmer may activate as a dependent, asynchronous process any procedure known to him as well as any program known to the system (subject to security constraints) including compilers, utilities, and other user programs.

A measure of the relative conciseness of B6700 Extended ALGOL is reflected in the size of some of the B6700 compilers. The Extended ALGOL compiler is roughly 15,000 source statements in length; FORTRAN IV, level H requires 12,000 card images while a full CODASYL 68 COBOL is implemented in 25,000 statements.

III. B6700 DCALGOL[2] as a Systems Programming Language

B6700 DCALGOL (Data Communications ALGOL) was originally intended as a slightly less safe superset of B6700 Extended ALGOL for the purpose of writing that portion of the systems software dealing with the data-communications interface to remote terminals and computers. The language contains many constructs dealing with handling of remote stations such as polling frequency, automatic dial-up, etc. It is beyond the scope of this paper to deal with this aspect of the language, therefore, we will concentrate on those features with which the programmer may exercise control over the central system.

It was previously stated that one of the constraints upon the facilities allowed in B6700 Extended ALGOL was that an ALGOL program was not permitted to detrimentally affect other concurrently executing programs not created by itself. The design constraint placed upon DCALGOL was that a DCALGOL program is not permitted to detrimentally affect the operation of the operating system (alias Master Control Program or MCP).

As mentioned above, the original conception of DCALGOL was that this language was a vehicle to control a data-communications subsystem. It has since been realized that the facilities embodied within the language provide a powerful, safe and convenient tool for more general systems programming. The basic differences between DCALGOL and Extended ALGOL, other than those intended specifically for data-communications, include:

1. Messages

   Messages may be thought of as a variable length string of characters whose meaning lies in the eyes of the beholder. Messages passed to the Data Communications Processor (an outboard processor of the B6700 CPU) have a fixed format in the first six words (36 characters), but messages passed between asynchronous processes may have any format whatever. An example of a message declaration and assignment is:

```
MESSAGE MESSAGETOCONTROLLER;
    .
    .
REPLACE MESSAGETOCONTROLLER BY "COMPILE MY/JOB USING COBOL";
```

   Messages may reside in save (non-overlayable) memory if realtime response is required. The ALLOCATE statement assigns a fixed number of words of save memory for a given message. The statement

```
ALLOCATE(MESSAGETOCONTROLLER, 8);
```

   reserves eight words for the message to be subsequently placed in MESSAGETOCONTROLLER.

   Messages form the atomic elements of Queues.

2. Queues

   Queues contain messages. A message may be placed in a queue at either the queue head or queue tail. Since queues may be passed as parameters to procedures, coroutines or asynchronous processes, such routines may communicate via messages. A queue may be thought of as a non-symmetrical, two-dimensional array in which each row might be of a different length. Messages may be added to a queue head or tail via the INSERT statement. One form of the INSERT statement is:

```
INSERT(MESSAGETOCONTROLLER, CONTROLQ, BOOL);
```

In the above statement MESSAGETOCONTROLLER is linked into the queue, CONTROLQ, at either the head or tail depending upon the value of the boolean expression, BOOL, true or false respectively.

The number of messages contained in a queue grows when an INSERT statement is executed and shrinks when a REMOVE statement is executed.

3. Remove Function

The simplest form of the REMOVE function is:

```
SIZE := REMOVE(MSGID,QID);
```

in which the message, MSGID, is removed from the queue, QID. The message size is returned as the result. A possibly more useful form of the REMOVE construct is:

```
SIZE := REMOVE(ARRAYID,QID);
```

in which the top message in the queue, QID, is removed and placed in the array row, ARRAYID, and the message size is returned as the result.

4. Queues may be interrogated as to their depth, concatenated to form a larger queue, split to form smaller queues, etc.

5. Operator Communication Facilities

The DCALGOL language contains two "known" queues. The first is a queue of messages by which a DCALGOL program may communicate with the operating system in exactly the same way as the machine operator may communicate with the system. This is known to the compiler as OPERATORQ. The second is the queue by which the operating system communicates with the operator. The depth of this queue is an installation option and is specified by either the maximum number of messages retained or the maximum duration of retained messages. This queue is known to the system as MCPQ.

Using these two queues, a DCALGOL program may monitor activities taking place within the system (such as jobs initiated independent of itself), perform scheduling, priority assignment and even abort jobs in the system of which it does not approve. In this way a DCALGOL program may assume complete control of the system, with the same "power" as the human operator, subservient to the control of the B6700 MCP.

The capability of DCALGOL is illustrated by the current implementation of the B6700 Remote Job Entry system. Written entirely in DCALGOL and providing remote sites with the full capabilities of the central site, the RJE software consists of under 2300 lines of DCALGOL source code.

IV. B6700 ESPOL[3] as a Systems Programming Language

Within the current concept of the B6700 operating system, B6700 ESPOL is the high level language utilized for the implementation of those machine dependent aspects of the B6700 MCP. Some of the facilities of the language and their uses have been adequately covered by Cleary[4]. The ESPOL language contains constructs which enable one to massage the physical resources of the machine and is, therefore, somewhat machine dependent. Another way of stating the above is that ESPOL contains constructs which are unsafe. Coding errors in ESPOL routines can be disastrous to the overall system.

Due to the inherent machine dependency of B6700 ESPOL, this language is of less interest to the subject of this paper. Even though a radical departure from generally accepted "state-of-the-art" practices, ESPOL is still another variant of B6700 ALGOL and, in general, quite readable by ALGOL programmers. The language contains constructs which allow it to directly address memory via a subscripted array reference, scan out control words to the B6700 multiplexor, interrogate the state of the physical hardware devices, etc. In theory, only those portions of the MCP actually concerned with manipulation of the physical resources need be coded in ESPOL with the remainder coded in DCALGOL or Extended ALGOL. Unnecessarily, under the current implementation, most of the operating system is coded entirely in ESPOL amounting to over 40,000 lines of source code. This excessive amount of ESPOL code will be gradually reduced as the inevitable refinements and rewrites take place. It should be noted, however, that the operating system implemented is a multi-programming, multi-processing, dynamic storage allocation system. Considering this, the "excessive" size looks somewhat more reasonable when compared with contemporary systems.

V. Conclusion

We have presented here some aspects of a currently implemented and successfully operational hierarchy of languages for systems programming. A large part of the success of the implementation is due to the fact that the B6700 systems architecture was designed with this high level language approach in mind. It is, nevertheless, felt that such an approach to systems programming would be successful regardless (or in spite of) the hardware architecture.

Many of the ideas presented in this paper will undoubtedly be obsolete at the time of presentation due to the impressively fast evolution of the system implemented using these tools. Although large systems programming has been done in high level languages at Burroughs for over 10 years, the power and flexibility of our current hierarchical approach are making themselves felt in the extreme flexibility in the things that may be done utilizing the system, resulting in high rate of development and improvement of the system. The value of this approach to software systems implementation is unquestioned by any of those concerned with the current system.

REFERENCES

1. "Extended ALGOL Reference Manual." Form 5000128. (7/1971). Burroughs Corporation, Detroit, Michigan.

2. "B6700 Data Communications ALGOL Language." Form 5000052. (11/1970). Burroughs Corporation, Detroit, Michigan.

3. "B6700 ESPOL Reference Manual." Form 5000094. (11/1970). Burroughs Corporation, Detroit, Michigan.

4. Cleary, J. C. (1969). "Process Handling on Burroughs B6500." Proceedings of Fourth Australian Computer Conference, 1969 The Griffin Press, Adelaide, South Australia.

5. "B6700 Processor." Form 1040326. (8/1971). Burroughs Corporation, Detroit, Michigan.