MLS—the Titan mixed language system

By J. Larmouth and C. Whitby-Strevens*

This paper describes a method of job-step control designed to operate within a multi-programming environment with a general purpose filing system. MLS combines a command language with the well-known techniques of relocatable binary and routine structure to produce a flexible programming system.

(Received March 1968)

1. Introduction and system outline

The Titan, installed in the Mathematical Laboratory at Cambridge, is a prototype Atlas 2 computer running under the control of a supervisor; this provides full multi-access facilities in parallel with normal batch-processing operation. The system is here described from the viewpoint of the off-line user, but is equally available from an on-line console.

A job is presented to the system as a series of documents. These are usually paper tapes or cards containing program, data, etc. Once within the system, however, a program reads from and writes to streams. Streams are known by number and are of two types: input and output. A special document directed to the supervisor (and known as the Job Description) gives various details about the job. In particular it defines the association between documents and input stream numbers, and specifies the destinations of output streams. Streams are stored in a standard internal code format, and the supervisor performs the necessary code translations at the peripherals. This enables programs using input/output statements to be independent of the source or destination of the streams involved. The first line of the Job Description denotes the programming system to be used. Thus, if the Job Description begins RUN MLS, the MLS command program is entered. This then proceeds to read and interpret commands. A typical command is:

(Fortran, 4, 2)

This command causes the FORTRAN compiler to be entered to compile, in order, input streams 4 and 2. These streams may have been set up in the Job Description as coming from paper tape, or could be the result of editing other streams. Thus the above command might be preceded by:

(Edit, 5, 6, §4)

This enters the editor and causes it to edit the document on stream 5, using the editing commands to be found on stream 6, and to leave the edited document as input stream 4.

The Titan supervisor provides a general purpose filing system with files held on fixed magnetic discs (Barron, Fraser, Hartley, Landy and Needham, 1967). A file may be read by specifying it as the source document of an input stream, and may be written to by specifying it as the destination of an output stream. Creating a stream to or from file may be performed within a command as well as in the Job Description. The user simply writes a file title instead of a stream number. Thus the command:

(Edit, /PROG, 6, §4)

will cause the editor to edit the file /PROG.

Input for a command may also be provided by small follows documents, thus:

(Edit, /PROG, F*, §4)

〈editing commands〉

* The asterisk terminator is an arbitrary symbol chosen by the user so as not to match any line in the editing commands. Note that many FORTRAN systems allow nothing but follows documents.

Files are stored in blocks of 512 words. Block transfers may be made to and from core and when a file is being used in this way it is usually referred to as a blockfile. Blockfiles form a convenient method of storing compilers and other systems programs (which will subsequently be referred to as processors) in absolute binary ready for use. Such processors appear as ordinary object programs to the supervisor, except that the associated blockfiles are owned by 'LIBRARY'. Hence, when a Job Description is headed RUN ALGOL, the supervisor simply finds the requested amount of store, loads the first block of the file LIBRARY/ALGOL and enters it. A call to the supervisor from an object program is available which hands back the currently occupied store and enters a processor in an exactly similar manner. This is termed a phase change and, within the multi-programming environment, is the standard method of proceeding from processor to processor. All entries to processors from MLS are performed by changing phase.

2. Statement of the problem

Early use of the Titan system demonstrated that it was convenient to enter, say, an editor from the Job Des-
cription and then change phase (by use of an ‘editing’ command) into a compiler, assembler, etc. This practice elaborated and it became clear that the typical run consisted of a series of job-steps; for example: edit, macrogenerate, compile and run. The use of job-steps led directly to the following two problems:

(a) How to specify which job-step to proceed to, having completed the current one.
(b) How to specify which documents are relevant to which job-step, remembering that output from one job-step may well be required as input for a subsequent job-step.

Two possible solutions exist to these problems:

(i) Each processor has its own conventions about which stream to expect its data on. Further, it either has a fixed successor or possesses directives for changing phase. In syntax, at least, these directives are likely to differ from processor to processor. This solution has the merit of making efficient use of the machine, but can become inconvenient to use; the user may have difficulty in ensuring that the right documents are used by the right processor. Sufficient flexibility must also be designed into each processor to permit a phase change into any processor which may be subsequently added to the system.

(ii) An overall command program is used, with each job-step returning to it. Each command specifies the processor to enter, and which streams or files that processor is to use. This also gives the opportunity for stream and file handling (creating, deleting, renumbering etc.) to be performed outside the individual processors.

There are two further points which require attention in the design of any system. These are:

(c) It must be possible to load a selection of pre-compiled library routines without explicitly asking for each individual routine.

(d) The form of the run-time system should not be dependent on any one particular language. Every routine loaded into store should be present on an equal status. This allows new languages to be added to the system with no change to existing programs.

3. The design of MLS

The problems outlined in the last section, together with the advent of a new FORTRAN compiler which would need a small driving program, led directly to the design and implementation of the MLS system.

The system was designed in the manner outlined in (ii) above with special emphasis on the job-steps needed in running an elaborate FORTRAN job. It was felt that this would provide greater flexibility than a system in which programs and data always followed the commands on the same document. The need to be able to add other high-level languages to the system was constantly in mind, and this gives the system its name. The main features of the system may be grouped under three headings:

(i) Runtime

The runtime system is routine oriented. Any number of routines may be placed together to form a program, and any routine may be produced by any compiler or assembler in the system. Routines may also come from public or private libraries. A library containing a wide variety of general purpose and diagnostic routines is always available, and other more extensive libraries are available on request (see the LIBRARY command in §3 (iii)). These libraries will, of course, have been produced by some compiler or assembler at an earlier date. A standard calling sequence is used between routines, and all compilers in the system must follow the associated conventions. Within the calling sequence parameters are specified by reference. Thus, if a parameter takes the form of an expression, then the value of that expression must be allocated a word of storage within the calling routine. This permits the conventional techniques of calling by reference or calling by value to be used between routines produced by different compilers. Calling by name is a manner for further conventions which, while perfectly permissible between routines produced by the same compiler, are outside the scope of MLS. Assembly language users are obliged to set up the calling sequences for themselves. (They may, of course, macrogenerate these.)

The runtime system contains a simple diagnostic system which is largely independent of language. The current position within each routine is specified by the contents of three special registers, and the compiled code must contain orders to set these appropriately. Typically this requires several orders at the head and tail of each routine, and one order for each executable line of source program. On an error at runtime these are decoded to produce a message of the form:

DIVISION OVERFLOW IN LINE 5 AFTER LABEL 4 OF SUBROUTINE EVALUATE CALLED FROM LINE 8 AFTER LABEL 22 OF SUBROUTINE INTEGRATE CALLED FROM LINE 7 OF MAIN PROGRAM.

After printing the position of an error, the diagnostic routine does not return directly to the command program but chains back using special error links. This enables language dependent postmortem printing to be produced by each routine involved in the error. The runtime system will always return to the command program after execution to obey any further commands. If the run produced a diagnostic, then an error flag is set on return. The user may include commands which are obeyed only if this flag is set (see §3(iii)).
(ii) Relocatable binary

All program destined for the runtime system is compiled into a standard form of relocatable binary. A core image may be regarded as a sequence of internal code characters laid end to end. It is possible for any number of characters to be read from an input stream or to be added to an output stream. It is also possible to convert an output stream into an input stream, thus using it as a 'first in first out' buffer. Storing small pieces of core image in this manner, interspersed with relocation directives, gives a primitive but easily manipulated form of relocatable binary. Public and private libraries may be stored in the filing system or on magnetic tape in this way. The loader used is a developed version of IAL (Internal Assembly Language) (Richards and Whitby-Strevens, 1967) designed originally as the loading phase of a CPL compiler.

The routine structure of the runtime system is present within the relocatable binary, and each routine is known by one or more unique names. These names are treated as parameters and each one takes the value of an entry point of the associated routine. Linking is performed quite simply by referring to these names in other routines. There are two modes of loading: LIBRARY and LOAD. In LOAD mode IAL loads all routines presented to it; in LIBRARY mode IAL will not load a routine if there are no outstanding forward references to any of the name parameters specifying entry points to the routine. IAL possesses a useful stream nesting facility. An IAL directive causes the loader to switch reading from the current stream to a second stream and then to continue loading in the normal manner. Another directive in the second stream causes loading to be resumed from the original stream (without specifying it explicitly). This will always take place if the end of the second stream is reached. When scanning a stream in LIBRARY mode, IAL will perform such a resumption if, at the end of any routine, no forward references remain. Thus a library stream need only be scanned as far as is needed to load the requested routines.

(iii) The command program

While IAL is the heart of the system at the machine level, the command program (Larmouth, 1967) is the skeleton and provides the user interface.

There is a small and well defined interface between the command program and processors within the system, enabling stream numbers, compilation modes and error flags to be handed over. Streams are of course global to commands, and no job-step will disturb a stream unless that stream is mentioned in the command. In addition to commands which control the selection of the next job-step, a large number of commands exist giving facilities for stream and file handling. A further purpose of the command program is to organise the relocatable binary that a job-step might produce. The command:

(Fortran, 4, 2, /PROG)

will cause streams 4 and 2 and the file /PROG to be compiled, and the resulting relocatable binary is left on a stream of whose existence the user is not normally aware. The command:

(Enter, /DATA, $START)

causes all such relocatable binary to be loaded and the program entered at the routine $START with the file /DATA set up as the main data stream. If, however, the user enters a compiler by writing, say:

(Fortran, 4, 3, $/BINARY)

then the relocatable binary is instead preserved in his file $/BINARY. By use of the stream handling commands the user can arrange both to preserve the binary and use it.

If the binary has been written to a file in an earlier run (or by an earlier command in the job), then the commands:

(Load, /BIN1, /BIN2)
/Library, /BIN3

will set up IAL commands on the 'hidden' streams so that on an ENTER command the files /BIN1 and /BIN2 will be scanned by IAL in LOAD mode and the file /BIN3 will be scanned in LIBRARY mode.

The command:

(Iferrors, F*)

<sequence of MLS commands>

will cause the supplied commands to be obeyed only if the previous command has given rise to an error condition. This may be used to obtain extra listings to aid the debugging of faulty programs or data.

The general form of a command is:

(<command name>, <arg 1>, . . ., <arg n>, $<arg>)

The value of n is completely variable, and may be zero. The arguments arg 1, . . ., arg n denote the input to the job-step about to be performed. The $ arg may be omitted, (in which case various assumptions are made), but otherwise it specifies where the results of the operation are to appear. The command program lists the arguments provided (on a special stream) and hands this list to the processor involved. This forms the major part of the interface between the command program and the processors in the system.

The command name is either a known word, such as LOAD, or else an arbitrary file name. In the latter case, the command program assumes that this file contains the absolute binary of some program, and requests the supervisor to load it as the next phase. Thus new processors can be added to the system without change to the command program. The input arguments 'arg 1' to 'arg n' may take one of three forms. If they are numerical then they are usually taken as input stream numbers. Otherwise they either have the form F$ (where $ is any character), indicating the presence of a follows document, or are treated as file names. In
either of these cases MLS will set up appropriate ‘hidden’ streams, and hand the numbers of these streams to the next job-step in the normal manner. Similarly the \$ arg will specify a stream or a file. In many cases the \$ argument specifies the input stream on which the result is to be set up (for example, after macrogenerating or editing), but occasionally it represents an output stream, as in the case of FORTRAN or COPY. (COPY concatenates its input streams onto the output given but does not destroy the input streams, so that it can be used to duplicate results.)

4. Conclusions

A number of points have emerged during the last twelve months, when MLS has become extensively used within the Laboratory. The system has proved convenient in use, and is unfortunately used from time to time when older methods with less machine overheads would do the work—albeit with a more difficult steering mechanism. The following points might be noted:

4.1 The ease with which streams and files can be handled within the system encourages the user to work with several documents instead of one. This introduces overheads in supervisor administration. A user who wanted two copies of his results one printed and the other punched, would, in a conventional system, use two WRITE statements in his FORTRAN program. Under MLS, he will probably use the commands REINPUT and COPY after his FORTRAN program has run and produced a single output. In central processor time, this is more efficient, for binary/decimal conversion and FORMAT interpretation are done only once. However, the procedure involves the supervisor in extra stream-handling. Streams are held mainly on the disc, and the resulting jam on disc channel time can more than balance the saving in central processor time when overall machine efficiency is considered.

4.2. Some problems have arisen with assumed output limits. In order to be able to schedule the use of the disc, the supervisor requires to be told the length of an output stream before the stream is used. This means that either the user must specify stream lengths whenever new ones are introduced, or that assumed values are taken. Initially the assumed values were made too high, and although the user found this very convenient, it defeated the supervisor scheduling and led to space jams on the disc. Reduction of the limits helped to some extent, but there are difficulties in estimating the lengths of the ‘hidden streams’ which the user never explicitly asks for.

4.3. There is one noticeable inadequacy in the system. The command structure allows multiple input parameters to a job-step, but only one output parameter. (This does not of course prevent a processor from outputting on a fixed stream.) This is particularly annoying in the case of COPY, when only a single copy can be produced at each application of the command.

4.4. Finally, it is worth noting that the already existing Titan supervisor was sufficiently flexible and powerful for the system to be developed with no supervisor alterations. However, the use of the system has changed the statistical load on the machine. For example, the average number of streams in use at any one time has increased considerably. This has not been absorbed without difficulty.

5. Acknowledgements

The authors would like to acknowledge the financial help of the S.R.C., and wish to thank the many members of the University Mathematical Laboratory, Cambridge who have contributed useful comments and advice during the development of the system.

References


