C

Abstract:

The development of the Algol W compiler and run-time system are discussed, with particular reference to input/output facilities and external subroutine linkages.

A Technical Report prepared for the 4th MTS Workshop held at Newcastle University in July 1978.

Algol W Development at Newcastle

By

J.A. Hunter and M.M. Hindmarsh

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 124
October, 1978

bibliographical details

HUNTER, James Alan.

Algol W Development at Newcastle. [By] J.A. Hunter and M.M. Hindmarsh.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 124.)

12"

Added entries HINDMARSH, Margaret Mary.

Suggested classmarks (primary classmark underlined)

Library of congress:		
Dewey (17th)	001.6424	001.6425
U. D. C.	681.322.06	

Suggested keywords

ALGOL W
COMPILERS
PROGRAMMING LANGUAGES

Abstract

The development of the Algol W compiler and run-time system are discussed, with particular reference to input/output facilities and external subroutine linkages.

About the author

Both of the authors are currently Programming Advisors for the Northumbrian Universities Multiple Access Computers organisation (NUMAC).

Table of Contents

1 Introduction

This year we will release the new version. We expect to have the first release system ready in December, and it will be known as Release 6.0. Documentation will follow by February '79, at which point we will distribute it. During the long development time, design aims have changed somewhat, but we trust that the final result will be acceptable to those MTS sites most interested. We assume, from the correspondence we have received, that these sites are primarily the University of Alberta, the University of Michigan and the University of British Columbia.

Our reason for producing this Workshop paper in the format of a Technical Report is to ensure a wider readership, particularly among the Laboratory staff at Newcastle. It is possible that certain members of the Computing Science Department may see this document as a 'U' trailer for an 'X' movie. We can offer some reassurance on this score. This report is essentially a final specification of the coming Algol W release, and nothing which is not described, or at least hinted at, in these pages will be in the released version 6.0.

Implementation is not yet complete in some areas of the system, and this means that there may be slight changes before release. For instance, if any better names for new pre-declared identifiers are suggested to us before release, we would certainly consider them. This paragraph then is a disclaimer: the user system will be described in a new edition of the User Reference Manual, and this report should not be taken as user documentation.

2 Design Aims

Initially we intended to do only remedial work to keep our own users happy. We had available a bolt-on multi-streaming I/O interface (our *STRAW), a modified compiler interface (our *AW) and an expressed wish to inprove the linkage to non-Algol W routines. The idea originally was to merge these three with a few minor modifications, and the figure of six man-months was quoted (Algol W working party - Newcastle July 1975).

By the time one of us (JAH) actually started to work on the project in November 1975, several other sources of feedback had appeared. A user survey brought replies from most departments at Newcastle, showing a diversity of requirements but almost all insisting on a vastly improved I/O system. The second major source of information was the 2nd MTS Workshop held at the University of Alberta in July 1975. This brought two major points to our attention. First, the University of Michigan (and most other sites) would like a standardized compiler system. Second, the University of Alberta would only be interested in an enhanced Newcastle Algol W if their modifications (originally from Manitoba) were incorporated. This seemed quite reasonable from their point of view as they had many users with programs dependent on their extensions.

As a result of this combined evidence, our own Algol W and Alberta's were both examined to see if a multi-streaming system could be built in which would be acceptable to both sites. At this point we would like to thank Kathryn Ward who sent us the University of Alberta system and Tony Marsland who visited Newcastle and put Alberta's point of view.

Our I/O extensions in *STRAW worked with READ(ON) and WRITE(ON) by providing a switching procedure to vary the reader and writer between the available I/O streams. Alberta provided new standard procedures GET(ON) and PUT(ON) which specify through their parameters the I/O stream and formats to be used in that operation. In effect, the Alberta system provides a Fortran-type I/O interface to Algol W.

When the coding and logic of the Alberta system were examined, the form of their implementation proved to be unacceptable to us. The additional code for the formatted I/O system highlighted a criticism of our own library in that it is not re-entrant. Algol W's library is not a library in the normal sense of the word where modules are selectively loaded as required. A program to add two numbers together will load the whole 'library' (30+ kbytes of code) regardless of the fact that it does not need complex ABS, or COS, or whatever. A further criticism of the Alberta system is the manner in which I/O information is buffered. It has two separate systems, with separate buffers, for READ/WRITE and GET/PUT, and this means that a user program writing to the same stream with both WRITE(ON)s and PUT(ON)s could have data appearing on separate lines in a possibly surprising order, depending on which buffer gets flushed first.

Having accepted in principle the desirability of the Alberta I/O entries, our own stream switching approach, and above all re-entrancy, a start was made on the design of a new I/O system. Since the compiler phases had been coded re-entrantly from inception, we have been able to produce a system which consists entirely of shared code. Only the user's object program need sit in his own address space. We consider this aspect of the improvements to be the most important one; with the trend towards re-entrant language processors we could not see Algol W surviving for practical applications without this improvement.

The principle of having a single compiler interface, as recommended by the 2nd MTS Workshop, was accepted. This takes the form of a single public file, *ALW, which determines by control cards and the run parameter field the mode in which it is to run. Many control cards are available but none are essential - all can be left to default.

Late in the project (later than we like to admit), we decided that any enhanced Algol W had to be able to run unchanged under other operating systems, notably IBM's MVS. There has been considerable discussion here during the last year about future operating systems for the NUMAC computers. Part of the discussion centred on what we believed to be the most important part of MTS at the user level. My own (JAH) feeling, as a programmer, was that the most likeable part of MTS was the clean subroutine interface. A module of about 25 routines has been written to implement this, with the calling sequences slightly altered from the MTS ones to allow easier implementation under other operating systems.

Algol W sits on this interface and the main system makes no direct calls to MTS.

As a consequence of this work a modified version of the Elementary Function Library has been produced for Algol W, the main changes being in the error handling, and a keyword/expression scanner more suited to Algol W's needs has been written. This latter subroutine, KXSCAN, is completely independent of other routines and has therefore been seized by our MVT/MVS crew who are unaccustomed to such luxury!

The remaining sections of this report are in greater detail and some familiarity with Algol W at the implementation level is assumed.

3 Re-entrancy and System Organisation

The new system is a development of the old, and will be distributed as the old compiler, a set of update decks, and new library and system control routines. To enable reference to be made to the system components we will give the names now:

DRIVER	Compiler control routines
SERVICES	Compiler I/O etc. service routines
AWA	Compiler scan and parse phase
AWB	Compiler code generation phase
LOADER	CLG mode loader
RENTALIB	Library initialisation routine
IOPACK	Run-time I/O system entries.
SYSPACK	I/O system control routines
FMTPACK	I/O system formatting routines
FDPACK	I/O system format string decoder
FUNRTNS	Analytic and implicit function entries
SHORTFNS	Modified type 'E' EFL routines
LONGFNS	Modified type 'D' EFL routines
RECORDS	Record allocation
BADGUYS	Run-time error processor
WOLFPACK	Independent routines e.g. KXSCAN
MTSSIM	MTS system interface module

All of the above modules reside in shared address space. Routines AWA, AWB and RECORDS are coded in PL360. All of the remainder are in Assembly Language.

We intend to have only one public file here for this Algol W system. It is called *ALW, and it links to the entry point of the shared system which is in DRIVER. This common entry point is used for object deck generation, compile load and go (CLG) and student monitor modes. The decision on which mode is in force is taken after inspection of the leading control cards in the source deck (if any), and the *ALW run parameter field (if any). By default an object deck will be generated.

Entry to a user program requires that the run-time system be initialised. In the original system this required one of the library modules to be the entry point for a user program. It picked up the genuine user program module addresses (AWXSC001 for the first code segment, and AWXRCTBL for the record description table), and branched to the library initialisation routine, which then called the first code segment to start execution. This is not a very clean way to go about things. To circumvent this linkage problem, the compiler code generation has been modified to generate an extra starter segment which is made the program entry point. The sole function of this control section, AWXSTART, is to branch to the library initialisation routine taking with it the address of a vector of user program segment addresses. The library initialisation routine is in RENTALIB.

By suitably structuring the address vector in AWXSTART, it has been possible to reduce dependence on non standard loader records to a minimum. The library entries referenced by user programs for I/O etc. are all contained in a low core symbol table AWSYSLCS: this is also in RENTALIB. Referencing this by a V-constant in AWXSTART eliminates the need for a RIP (Reference If Present) loader card, so that only:

```
     LCS           LCSYMBOL
     LCS           AWSYSLCS
```

are required to link in an Algol W user object program in MTS. AWSYSLCS is a low core symbol table containing all entry names in Algol W which are required in the user interface. This includes all library routine symbols and the compiler entry point name AWDRIVER. AWSYSLCS is the only symbol required in the MTS system low core symbol table LCSYMBOL. The public file *ALW needs to contain only:

```
     RIP  AWSYSLCS
     LCS           LCSYMBOL
     RIP  AWDRIVER
     LCS           AWSYSLCS
     LDT           AWDRIVER
```

to allow the shared code compiler system to be invoked.

In load and go mode the necessary data structures are built by LOADER, and the RENTALIB initialisation routine is called by the control routines in DRIVER. LOADER is a new re-entrant loader, with most of the previous restrictions removed. For example, under test it has loaded a 220 segment, 144 kbyte program. Design of the loader takes into account the provision of a linkage editing option to generate single control section object decks; this will, however, not be provided initially.

Run time input/output entries are accessed via IOPACK. they are implemented by a set of routines in SYSPACK and FMTPACK. The internal linkage for these routines is implemented by a set of stack manipulation macros which solve many of the problems of structuring a complex I/O system.

Standard functions of analysis (SIN, COS etc.), and implicitly called functions (powers, complex arithmetic) are accessed via FUNRTNS. All elementary function calls are immediately passed on via transfer vector entries to the relevant routines in SHORTFNS and LONGFNS. These are the Michigan EFL library routines modified so as to be suitable for Algol W.

Record allocation is unchanged from previous versions of Algol W. The routine is still in PL360 and the only conversion work done has been to achieve re-entrancy.

The run-time error processor (BADGUYS) is still on the stocks. We have a very primitive version available at the moment, basically to prevent the system collapsing about our ears whilst testing. It looks as if this will be the last module coded. The basic error messages will not be quite so basic when we do get around to implementing them. Using the Elementary Function Library is one factor in this, since it does not forget the starting values of function arguments. Similarly a fair amount of information is available if an error is detected somewhere in the I/O system. Post mortem dumps and more sophisticated facilities will not be available in release 6.0 - see later discussion.

The last two modules to be mentioned are WOLFPACK and MTSSIM.

WOLFPACK contains subroutines which are logically independent of the Algol W system although called by it. An example of this is the KXSCAN (keyword-and-expression-scanner) subroutine. This is a re-implementation of the University of Michigan's KWSCAN routine with many of the restrictions removed: in particular it can process free standing right hand side expressions. It can therefore be used for the implementation of both the Newcastle and Alberta style input routines, with the exception of the Alberta "A" and "Z" formats which do not fit into KXSCAN's conception of an expression.

MTSSIM is the only routine in the system to call MTS subroutines directly. It provides a subroutine interface by which the other modules communicate with the operating system. The rest of the system is independent of MTS. System independence is discussed in the next section.

Distribution of the system should be fairly clean. In answer to expressed anxieties at the 2nd MTS Workshop, Algol W will be distributed in a form which only requires the G Assembler and the Linkage Editor to be available. A PL360 (fairly old and simple) and an UPDATE (a by-product of my 2H-CS Assembler teaching load) will be included, plus all source decks required to generate and test the system. All development here is centred on a special user identifier ALWB. It would be an advantage to any implementor to have an ID of this name available. All macros used in system assemblies are supplied in a single library. This is maintained as an IEBUPDTE source deck to facilitate use under MVS/MVT. Last but not least, Algol W only uses the System/360 Model 65 subset of System/370 machine instructions.

All entry point names within Algol W which are required for Algol W are 8 characters in length, the first two being AW. The requirement for the MTS shared system is about 120 kbytes of store and one symbol (AWSYSLCS) in LCSYMBOL.

4 Operating System Independence

Newcastle would like to be in a position to have the same version of Algol W available for both MTS and MVS. The reasons for this are two-fold. First, we will ourselves be running both operating systems on the 370/168 from December 1978. Also we have received several requests from OS installations for a better version of Algol W. We would like to be able to satisfy these requests, probably as a cost covering operation rather than as a commercial enterprise.

With this in mind, the new system has been designed with all of the MTS dependent parts built into one module, MTSSIM. In theory, Algol W could be run under other systems simply by replacing this module by MVSSIM, CMSSIM or whatever. In practice, some tinkering might still be required. The two points which come to mind are line-number/record-key formats, and the generation of MTS-specific LCS loader records.

MTSSIM provides a subroutine interface which is intended to be well defined. We do not believe that it will be until we have implemented the system under at least one other operating system.

Three kinds of subroutines are provided. They are storage allocation, initialisation, and service routines.

The storage allocation routines are AWGTMAIN and AWFRMAIN. Both are register call routines and neither require register 13 to point to a save area. These two routines may be called at any time whether or not the system interface is in an initialised state.

All other routines in the system interface have standard OS Type I convention calls with a fixed number of parameters. Many of the subroutines will be instantly recognisable to members of the MTS installations as variations on the basic set of MTS I/O subroutines. The strategy developed is to make the first parameter of each routine the address of a one hundred double-word work area. This allows an implementation using this interface to achieve re-entrancy without hanging storage addresses on the operating system.

The initialisation routines are AWSIMBGN (which must be called before any routine other than AWGTMAIN/AWFRMAIN is presumed available) and AWSIMEND (after which call the routines other than AWGTMAIN/AWFRMAIN are no longer available). Within these AWSIMBGN/AWSIMEND brackets any of the other routines may be called. Note, however, that the 100D work area must be the same region for all system interface routine calls, and that it must not be changed by the caller. It is assumed that AWSIMBGN is called only once at the start, and AWSIMEND only once at the end. This affects the implementation of the Algol W library in that, in load-and-go mode, the library does not call the initialisation routines. These are called by DRIVER once only, and the 100D work area is passed to the library together with an indication that it is being called in load-and-go mode. When running an object deck or decks from a file, the library will call the SIM initialisation routines, again only once each.

The service routines are the raison d'etre of the whole system interface. A set of about twenty routines is provided to do input/output operations and one or two specials. For instance, there is an accounting entry which is called from several points in the system. This will be supplied as a null routine, but an installation requiring such facilities would be free to implement it without modifying the main Algol W system.

As an example of a routine lifted from MTS consider AWRDRECD. As its name suggests it is used to read in records from an input file-or-device into memory, and as such is equivalent to the MTS READ system subroutine. In MTS a record would be read by:

```
        CALL  READ,(REGION,LENGTH,MODS,LNUM,UNIT)
```

In the Algol W system the equivalent call would be:

```
        CALL  AWRDRECD,(WORK,REGION,LENGTH,MODS,LNUM,UNIT)
```

where WORK is the 100D region previously mentioned, and the other parameters are as defined in the MTS READ subroutine specification. The similarity between the two calls should be readily apparent. We have yet to decide which modifiers will be supported in our implementations in operating systems other than MTS. Indexed, sequential, notify and errrtn are at present used by the main Algol W system and would have to be included.

In order to free Algol W from dependence on the limited set of MTS logical device names, the unit parameter is derived by the Algol W system from an interrogation of the system interface via a subroutine AWDDNAME. When supplied with an 8-character name AWDDNAME returns a 1 to 8 character key for use with AWRDRECD in the UNIT field. A set of internal names is defined which is the only one seen by an Algol W user within a program. To invoke that program in a particular system, he or she has to be aware of the mapping of internal and external names in that operating system, for instance:

INTERNAL NAME	MTS LOGICAL DEVICE NAME	MVS DATA DEFINITION NAME
INPUT	SCARDS	SYSIN
PRINT	SPRINT	SYSPRINT
PUNCH	SPUNCH	SYSPUNCH
ECHO	SERCOM	SYSLOG
USER	GUSER	SYSUSER
0 ... 19 (as strings or integers)	0 ... 19	SYSUT0 ... SYSUT19

An Algol W user might code in his program:

```
PUT("PUNCH", "X,A3,2X,A66,I8", CODE, DATA, ISEQ);
```

Algol W internal tables are aware of PUNCH - in fact a hash look up is done. The unit field in the internal control block for PUNCH will have been filled in at I/O initialisation by AWDDNAME. For MTS it will be CL8'SPUNCH'. It need not be, as Algol W does not inspect the unit field, but merely uses it in calls to the system interface. It is convenient in MTS to supply CL8'SPUNCH' because this cuts down the work that the system interface has to do. In other systems a 4-byte table address might be more appropriate.

The Algol W system links into the system interface routines directly so they are loaded together. However, for other purposes two low core symbol tables are provided, AWSIMCS and SIMLCS. The first contains all the routines, and SIMLCS contains the general use subset of the routines with 6-character long names formed by stripping off the leading AW, e.g. RDRECD for AWRDRECD. These entries are therefore Fortran callable. They should provide us with a means of accessing an MTS-like subroutine environment in both the MTS and MVS systems.

5 Compiler Control Cards

As previously mentioned, none of the control cards are essential. If a file containing only Algol W source statements is presented to *ALW via (in MTS) SCARDS, it is compiled to an object deck. The file -AWLOAD is obtained and emptied for this purpose. When invoked conversationally, by default no listing is produced. In batch a listing is written to (again in MTS) SPRINT. Should a terminal user wish to have a listing he can either assign SPRINT explicitly or code PAR=SOURCE on the $RUN command. In the latter case, with SPRINT not assigned, the file -AWLIST is obtained and emptied for the listing. If SPUNCH has been explicitly assigned then this is used for the object program rather than -AWLOAD.

To cause a program to be loaded immediately and executed after compilation, assuming no data is required from SCARDS, all the user needs to do is code PAR=GO on the RUN command. Action proceeds as for the default deck generation option, except that the object program is written to a virtual file in the user's address space, -AWLOAD not being used. This object program is then loaded and executed. The compiler main data area, which is now free, is used for this purpose. Storage remaining in this region after loading the program provides space for the run time stack. The virtual file storage is freed before execution of the user's object program.

Using control cards, the basic load and go job is :

```
/ALGOLW   <compilation parameters>                --+
... ... ...                                         |
... ... ...                                         | Repeat
<algol w source statements>                         | zero
... ... ...                                         | or more
... ... ...                                         | times
/EXECUTE  <run time parameters> --+                 |
... ... ...                       | Repeat          |
... ... ...                       | zero            |
<data for the above program>      | or more         |
... ... ...                     --+ times         --+
```

Note that a program, once compiled, may be executed many times by repeating the EXECUTE card. This apparent miracle is achieved by copying the record description table on entry to the library, which was the only obstacle to re-usability (and, in fact, re-enterability) of an Algol W object program.

Many other control cards are supported, and a brief description follows. The recognition of control card "verbs" requires at least four characters to appear (including the slash prefix), and any abbreviation from the full name down to that limit is acceptable. All control cards start with a slash, including former directives such as /TITLE, /LIST and /NOLIST. To enable old decks to be run with the new compiler, a keyword parameter is supplied to allow definition of an alternative starting prefix character.

The original intention was to satisfy the appeal made by Gary Pirkola at the Second MTS Workshop for a standardized set of control cards for MTS language processors. We feel we have achieved most of the requirements; however local conditions and structure of the language processor system have meant a couple of compromises had to be made along the way. For instance, /COMPILE would cause difficulties here. The method by which NUMAC's Batch Monitor student systems work is for the MTS and OS HASPs to recognise an <S8> card followed by a language control card, and queue the job for the corresponding monitor task. At present we have three batch monitors: Algol W (control card $ALGOL), WATFIV ($COMPILE) and PL/C ($PLC). Hence /COMPILE would cause user confusion with the WATFIV system. For any installation requiring /COMPILE a one line modification to a keyword table is all that is necessary: however, the NUMAC distribution version will use /ALGOLW instead.

Because we are committed to supporting the present production Algol W system for two years after the new one goes in, we feel that certain changes can be brought forward in the new version as conversion of users is bound to be slow. Control card recognition is one of these areas. Control cards will now begin with a slash (/) symbol only. The distinction between control cards and compiler directives has tended to confuse some of our users, so the directives have been reorganised as follows. Directives which control the action of the compiler scanner (i.e. those which have some instantaneous effect) have been made into control cards. Hence @LIST becomes /LIST, etc. Directives whose action was a constant of the compilation (i.e. those which do not have any effect until parse or code generate time) have been made compiler run parameters. For example @NOCHECK becomes NOCHECK as a parameter.

Compiler parameters may appear either in the run parameter field of *ALW, or on a control card indicating that a new Algol W program follows. At this point we shall discuss how *ALW differentiates between modes when control cards are present.

There are three modes in which *ALW can run. It can generate an object deck: we shall call that one Deckgen. It can compile, load and execute a program: that we shall call CLG mode (G for GO). It can support a student monitor system. This mode, Monitor, is a special case of CLG mode where certain facilities are denied to a user to protect the integrity of the system (for instance, access to Assembler coded subroutines of unknown ferocity).

When *ALW is invoked, its first action after the necessary initialisation has been carried out is to read in the first record from its INPUT stream, which, in MTS, will be SCARDS. Invoking the input routines in SERVICES to request a single record will cause records to be read from SCARDS until either a program start control card or a non-control card has been read (end-of-file, would of course, cause termination). Program start cards are /ALGOLW or /OPTIONS. Other control records are processed immediately (for instance /COMMENT which is a null operation) or they generate a control record for the compiler. An example of this latter type is /LIST, which, by forming a data record for the compiler scanner, transforms itself into a non control record and therefore stops the reading process.

As a result of this action, *ALW's control routines in DRIVER are either in possession of a /ALGOLW or /OPTIONS record which they must process, or have a control or source record for the compiler (which has not yet been invoked).

/ALGOLW and /OPTIONS are fundamentally the same pseudo command. Both take the same parameters, which are the same set valid in the *ALW $RUN command parameter field. However /ALGOLW has an implicit GO parameter attached (it invokes CLG mode) whilst /OPTIONS has associated with it an implicit NOGO (stay in default Deckgen mode). GO or NOGO parameters on the cards could, of course, change the mode again if the user so desired.

Because the set of compiler parameters is now very extensive, a continuation mechanism was felt necessary and this is provided by allowing cards with a plus sign in column one to be continuations of /ALGOLW or /OPTIONS cards (note that plus cannot start a valid Algol W program). *ALW continues by processing the parameters on the control cards held and then looping to read and process continuations until a non-control card (or compiler control record) has been encountered. From this moment the mode is cast in concrete and may not be changed during this run. If a /MONITOR record is encountered before the first /ALGOLW record, Monitor mode is set and remains in force until the run terminates.

At this stage the compiler is invoked. Its first read will supply the pending source or control record which must by now be held by the DRIVER routines. Reading proceeds with the interface routines permanently on the lookout for further control cards. This may sound a rather complex way of organising it but it implements Gary Pirkola's main recommendation without restricting the users' use of the interface, although most users will probably limit themselves to a set of simple recipes. A description of the various control cards is given in Appendix One, and a list of parameters which may appear on /ALGOLW or /OPTIONS cards, or in the parameter field, is given in Appendix Two.

Batch compilation of object decks is now provided. If an Algol W source program is followed by a /OPTIONS card then the compiler decides to compile another program, or, more usefully, procedure. This action may be repeated as often as desired, allowing a library of routines to be compiled in one run of *ALW. The *ALW run parameter field is applied to each compilation after the /OPTIONS parameters, if any, have been processed.

For convenience, another control card is provided to allow the overall compilation options to be supplied with the source deck in one place. This is /GLOBAL. Parameters given on this card are applied for each compilation before the /OPTIONS card parameters are processed. This removes the need to repeat common parameters on each /OPTIONS card, but allows the global ones to be over-ridden in a particular case if so required. Note that the *ALW run parameter field is applied to each compilation after the control cards have been processed.

Monitor mode is worth further clarification. This coding is the result of several discussions with Peter Whillance of NUMAC's Basic Systems Group. The aim of this mode is to provide an idealised Batch Monitor environment.

The major problem with Batch Monitor system implementation is in deciding when all output from a job has been produced. In most implementations of Algol W, for example, a program requiring no data will be compiled and run when the next user's $ALGOL card forces this action. This introduces a fair amount of unwelcome complexity into HASP Batch Monitor interfaces. They have to be aware of the control card constructions used by a particular language processor and cannot treat the process as a convenient black box. To help with this problem a /FLUSH card is provided. Processing of this card causes the *ALW system to complete the current compilation and execute the program. If a program is executing an end-of-file will be supplied to it, should it attempt to read further through the source deck, until the program stops. No further /EXECUTE requests are then accepted and the Monitor will not process another job until a valid /ALGOLW card comes along. Any HASP interface has then only to block the jobs together supplying a /FLUSH card between each. There should be little problem in deciding which output goes with which job.

As part of its Batch Monitor role, *ALW contains coding to produce a header banner for each job processed, and also calls a verification and logging entry in the system interface module. Any installation-dependent identifier checks and job logging can therefore be done by local mods in MTSSIM. The Monitor will not process a job unless an ID is given on each /ALGOLW card, and this ID is passed OK by the verification routine.

Provision for input from other files is provided via a /COPY control card. In MTS, of course, $CONTINUE WITH is available. However, not all operating systems are so fortunate (or so cursed?), and with the Batch Monitor it is essential that errors in a copy file do not stop the Monitor. After a /COPY command has been obeyed, and end-of-file signalled from the copy file, processing continues with the original input stream. At the present time, chaining of /COPY commands is not allowed but future versions will have this facility. In Monitor mode, /ALGOLW cards in a copy file are treated as /FLUSH to prevent several jobs being batched together on the sly. All other control cards are available.

All input by the compiler system allows, by default, records of any length. Long records, if they are being sent to the compiler, are split into 72 byte chunks and sent as requested. Options are provided to check for short records only. If such a check fails, or indeed any error is detected during a compiler read operation, an in-line error message is produced as part of the compiler listing.

6 Run-time Input/Output

Our intention here is to provide both the Newcastle and Alberta routines in such a way that they behave as the user would expect them to do. (We are refering here to the problem of the separate buffers mentioned earlier.) We also want a multi-streaming I/O package which can be accessed via the Newcastle entries. A secondary aim is to provide some kind of indexed input/output facility with database projects particularly in mind. First, here are some definitions.

By Newcastle entries we mean the original standard I/O routines READ, READON, WRITE, WRITEON etc. The Newcastle extension to these routines was the 1971-72 addition of an output formatting system based on pre-declared variables such as R_FORMAT, R_W etc. Use of these routines in the past has always implied that Algol W possesses only a single input stream and a single output stream and for many years this has formed the basis of most serious criticisms of Algol W.

By Alberta entries we mean the set of I/O routines which specify in their calls both a source/destination unit and a format string. These routines originated at the University of Manitoba and were merged with the 1972 Newcastle release by staff at the University of Alberta to form an MTS version. Most of the present Newcastle strategy for an enchanced Algol W comes from a wish to provide these facilities without the overhead of the larger library - hence our re-entrancy goal. Alberta's entries effectively provide Fortran formats in Algol W: their names are GET, GETON, PUT and PUTON.

To implement both series of I/O entries, a basic set of multi-streaming I/O routines has been written. The Newcastle and Alberta entries sit above these basic routines. The system is set up so that each input/output unit in use has its own control block, with I/O buffers chained on to it. We define a stream name for each unit to be a string of from 1 to 30 characters formed from the set A - Z, 0 - 9. These stream names fall into one of two categories.

Predefined stream names are provided to access system I/O units as described in the section on system independence. Hence INPUT in MTS accesses SCARDS and PRINT accesses SPRINT. The I/O unit numbers can be accessed either as a string ("0") or as an integer.

User defined streams provide dynamic allocation of files and devices. A new standard procedure, ASSIGN, both defines a stream name to Algol W and assigns a file or device name to it. When work on this unit is finished another standard procedure, RELEASE, may be called to free the file or device and disable the stream name. For example:

```
ASSIGN("PRINT","-OUTPUT(LAST+1)");
```

re-assigns the PRINT stream. In MTS a call to SETLIO in MTSSIM would result in LDN SPRINT being re-assigned. Dynamic allocation of streams is performed in the next example:

```
ASSIGN("DATA", "CL38:DATAONE");
GET("DATA", "3X,I5", NDITEMS);
FOR I:=1 UNTIL NDITEMS DO
   GET("DATA", "3(F12.5,5X)", A(I), B(I), C(I));
RELEASE("DATA");
```

This example dynamically creates an input stream called DATA, reads in some values from it, and then closes down the stream again. Because DATA is user defined, the low level calls would be to GETFD and FREEFD in MTS.

To provide multi-streaming support for the Newcastle entries, we have introduced the concept of the current Reader stream and the current Writer stream. Initially the current reader stream is INPUT (in MTS - SCARDS) and the current writer stream is OUTPUT (MTS - SPRINT). Two more standard procedures, READER and WRITER, are provided to switch streams. For example:

```
READER(2);
FOR I := 1 UNTIL 30 DO READON(DATA(I));
```

The READON statement here will be reading from unit 2. One of the functions of READER will be to ensure that reading from unit 2 via READON fetches a new record, regardless of the state of stream 2 or of the previous current reader stream.

Another example is

```
     WRITER("SPUNCH");
     FOR I:=1 UNTIL CARDS DO WRITECARD(DECK(I));
     WRITER("PRINT");
```

This example causes the writer stream to be temporarily re-assigned to PUNCH (in MTS - SPUNCH) to allow the program to dump a string array on this unit. WRITECARD is the output parallel of READCARD, and has been added to the system both for completeness and for compatibility with the Alberta implementation which also has it.

One major difference between the Newcastle style output procedures (WRITE etc.) and the Alberta ones (PUT etc.) is that the latter never generate carriage control characters automatically. If required, they must be supplied explicitly via the output list or the format string.

The Newcastle WRITE and related entries do generate control characters automatically. This action may be disabled either by specifying PAR=NOCC on the run command, or by setting a new pre-declared logical variable WRITE_CC to FALSE.

A series of standard procedures has been added to the system to provide basic services such as rewinding and emptying files attached to I/O units. A complete list appears in Appendix Three.

A set of additional entries for string input/output has been provided which includes support for simple indexed operations. Because Algol W normally buffers its I/O, indexed operations cannot be provided with the usual I/O entries which deal in data items, because the actual physical reads and writes occur at unpredictable times determined by the state of the I/O unit buffers. However, all the string I/O entries cause immediate reads and writes and can be indexed if required. The option taken was to provide separate entries for the indexed operations. All these entries follow Alberta's style in that they specify the I/O stream to be used. One example is

```
     GETCARD(9, STRI);
```

which reads a string from unit 9 into STRI. If the record read is longer than the declared length of STRI then it will be truncated; otherwise it will be padded with trailing blanks. Another example is

```
     PUTCARD("PUNCH", CARD);
```

which, similarly, writes a card image to PUNCH. The indexed entries all specify a line number in internal form as the second parameter of the call. Implementation in other operating systems requires that the index or key is made to appear to be an integer. However, the key could be allowed to take a different form with very little modification on the Algol W side of the system interface. Examples are:

```
     XGETCARD("MF", INDEX, INBUFF);
     XPUTCARD("DF", INDEX, OUTPUT);
     XDELETE("UF", INDEX);
```

In all three of these entries the second parameter, INDEX, is an index to the file on the specified unit. The last entry, XDELETE, deletes records. This is provided both for clarity and because Algol W does not have zero length strings for a null write, as MTS has. MF, DF and UF are assumed to be user defined stream names created using the ASSIGN procedure, and attached to files which permit indexed operations.

All these string entries accept strings of length 1 to 256 bytes. The restriction on READCARD (which insisted on an 80-byte destination string) has also been lifted, and the complementary routine WRITECARD has been added with the capability of writing strings of any length up to 256 bytes.

The two string input procedures previously mentioned (GETCARD and XGETCARD) bypass the normal end-of-file handling (using the ENDFILE reference to the pre-declared EXCEPTION record) providing instead a single pre-declared logical variable FILEMARK. This should, however, be inspected after each attempted read operation. Such a simplification is particularly necessary in the case of the indexed entry XGETCARD, where the absence of a record is synonymous with end-of-file.

Many of the requirements for input items have been relaxed, both for READ/READON and GET/GETON.

Strings can now appear in an input field either within quotes (") or primes ('). If there are no blanks in the string then the quotes or primes surrounding the string are optional. With bit strings the hash symbol (#) is now also optional.

Logical values may be read in as TRUE or FALSE (or any abbreviation down to T or F), or as the integers 1 or 0.

Restrictions on floating-point number formats have also been relaxed. The exponent separator which is currently the prime (') may now also be the characters "E" or "D", thereby allowing the output from Fortran programs to be read into Algol W without prior editing. Also, in the present system, a trailing "L" is required for long real values to retain their accuracy. We feel this to be nonsensical and will, in the new system, decode all numbers as type D accuracy until they are stored. The presence of a trailing "L" will be failed in the new system. This change is not upward compatable, but we feel it is logical to introduce it at this time.

Complex numbers present a problem in that the present system cannot re-input complex values which it has previously output. This is because the input format is A+BI where there are no embedded blanks. To cure this we intend to allow complex values to be input as either A+BI or as (A,B). The second format may contain embedded blanks, and this will be used to output complex values. The brackets may seem a trifle strange but it will allow us to both retain neat columnar output and allow later re-input of the items.

The Alberta Fortran-like formats fall into three groups. There are control entries (H, T and X), there are specials (A and Z) which do not obey the rules for expressions under the Newcastle system, and there are the remaining formats (D, E, F, I and L) for which expressions may easily be defined. This last set will be processed by the same KXSCAN routines used for the Newcastle formats just described. Each defines a field width: some define a decimal field width, but this will be ignored by the input routines. Within each field examined, one and only one expression must appear, of the correct type, or an error will be recognised. Certain Fortran attributes such as recognising blanks as zeros, implying the position of a decimal point, and allowing spaces between the exponent separator and the exponent will not be implemented, as they are considered contrary to the spirit of the Algol W implementation. A numerical value will, after input, be exactly as it appears to be, without suprises for the unwary.

The two special formats (A and Z) present no particular problems. Defaulting the field width on an A or Z format specification will imply a width which is the length of the variable in A format and twice that in Z format. Embedded blanks will be allowed to enable tidy input of long hexadecimal strings to be produced.

For output formats we do not intend any major changes other than in the complex number format mentioned earlier. However, to enable Algol W to output floating point values that can be read in by Fortran, the exponent separator character has been made a variable, R_EXPCHAR. It still defaults to the prime ('), but can be set by a user to "E" or "D" as desired. Also in the floating point arena, we intend to tidy up the Newcastle freepoint format but as yet the form this will take has not been finally decided.

Alberta provide, in their system, the ability to do I/O conversion both into and out of string variables within the program. They do this by allowing the string to appear in the unit field of a GET(ON) or a PUT(ON) procedure call. By our more general stream name definition we have prevented this construction so, clearly, it must be re-introduced in another form. We have done this by providing two extra standard procedures:

```
GETSTRING(string, format, variable-list);
PUTSTRING(string, format, variable-list);
```

User controlled error recovery will be provided for these routines.

Lastly, in order to provide some degree of interchangeability between the Newcastle and Alberta I/O procedures, the following extensions are defined. A format string may be the reference value NULL in which case the Newcastle formatting system is used. Similarly, Newcastle formatting is also invoked if a format string is exhausted (rather than causing a format rescan).

This allows a certain mix:

```
     REAL A, B;    STRING(24) S;
     ... ...
     ... ...
     GET ("INPUT", "A24", S, A, B);
```

would read the first 24 bytes of the record into the string S and then locate two real numbers by free format scanning.

To allow the reader and writer streams to be used in GETs and PUTs without the programmer having to remember their assignmemts, two pre-declared variables are provided to reference them. Examples of these are:

```
     GETCARD(RDR, LINE);
```

reads a line from the current reader stream, and

```
     REWIND(WTR);
```

rewinds the current writer stream, presumably for re-input of the data.

7 External Subroutine Linkages

Apart from the I/O system, one of the major criticisms of Algol W over the years has been the inadequacy of its external linkages. Two means of communication with external routines were provided, one for precompiled Algol W procedures and another to implement the IBM OS Type I "S" type linkage (to Fortran and Assembler for instance).

In general, the linkage for Algol W procedures has weathered the years well. The mechanism in the source code involves replacing a procedure body by the sequence: ALGOL <string>, where <string> is a string literal defining the external procedure first code segment ESDname. The implementation of the linkage, in which the compiler omits the generation of the procedure code segment, leaving the gap to be filled by the precompiled code, is attractive in its simplicity and means that calling an external Algol W procedure is no more expensive than calling an internally defined one. With the single exception that unbound global identifiers may not be referenced in the procedure header of the precompiled external routine, all of the facilities of the Algol W procedure call mechanism are available. This restriction, however, does mean that reference parameters may not be passed (as they need to refer to a record class which cannot be defined when the header is built). It is possible to get round this by a fiddle but we cannot see any way in which this could be made cleaner.

One curious feature of the ALGOL linkage will change. Users have always had to apply a peculiar algorithm to determine the entry point name from their procedure name. In the new system these two names will be the same, with a supplied ALGOL <string> name being truncated to eight characters if necessary. The original scheme came about because of the way in which Algol W picks its segment names. Basically a five character module root is produced by either truncating the name to five characters, or, in the case of a short name, by padding it out to that length with hash (#) symbols. Segment names are then produced by appending 001, 002, 003 etc., and the user has to supply the name of the first segment. Hence arose the peculiar requirement for <rootchars>001 . The compiler has now been modified so that this scheme is followed only for the second and subsequent segments. For the first, the name of the procedure itself is used, truncated to eight characters if necessary. Users need only be careful, in a many procedure environment, to ensure that the first five characters are unique.

For linkage to routines coded to the OS Type I standard, the situation is not so satisfactory. The mechanism requires that a programmer should code FORTRAN <string> as a procedure body. This generates a code segment which sets up a call to the external routine whose ESDname is <string>. Whilst the overhead in calling an Algol W procedure is normally small, it does mean that each call to an external routine involves a double call within the Algol W program.

Other restrictions of this mechanism are more serious. Because a rigid type mask is imposed by the procedure header, in order to call the same external routine with either a different number of parameters or a parameter of a different type, the procedure linkage must be declared again. This is time consuming and wasteful in the amount of code that is generated. Furthermore, most languages which are called by this linkage do not have the rigid restrictions on variable types found within Algol W. This provides an excellent discipline for a programmer within the Algol W language, but it is quite wrong to attempt to impose it on the outside world, and it has become clear that if Algol W is to survive as a research tool then a better linkage must be provided.

Originally we intended to change the language definition slightly to accomodate a new external linkage. However, this met with considerable opposition from the academic side of the laboratory. Fortunately, it was found possible to provide all the requirements utilising yet another variation of the standard procedure mechanism.

Standard procedures were originally provided to support only the input/output system via READ, WRITE, etc. However, the main properties of standard procedures make them nearly ideal for providing an external subroutine calling mechanism. They accept a variable number of parameters, according to the whim of the programmer, and these parameters may be of any simple type; that is, any scalar quantity may appear as a parameter - literal, identifier, field id, or simple expression.

The main standard procedure we have provided is named CALL. The first parameter to this procedure will normally be a literal string, in which case it is taken to be the entry point ESDname of the called routine. This routine name, being supplied as a literal, does not need declaration. This provides effectively the equivalent of the Fortran CALL statement. Subsequent parameters to the standard procedure form the parameters of the called routine. Some examples of CALL are

```
     CALL("MTS");
     CALL("READ", REGION, LEN, MOD, LNUM, UNIT);
```

The first example sets up a call to subroutine MTS without supplying any parameters. The second sets up a call to the READ subroutine with five parameters.

In the spirit of the OS Type I linkage, the variable length convention is obeyed. If there are N parameters, bit zero is set on the N'th parameter address. If N is zero then general register one, which would normally point to the parameter list, will be zeroed.

Passing of arrays as parameters presents no problems even though only simple types are allowed. A user simply supplies the element at which he deems his array to start, followed by the size of that array. Algol W calculates the address of that first element, and passes that address on as the parameter. Hence:

```
     REAL ARRAY DATA(N::M);
     ... ...
     ... ...
     CALL ("SUM", DATA(N), M-N+1);
```

This is, in fact, a clearer solution than that of the current FORTRAN linkage. Arrays passed as parameters via the present construction result in a calculated address of the element whose subcript (or subscripts) is (or are) one. This tends to confuse users; we hope that the CALL scheme will make them think more deeply about array organisation. Fortunately Algol W and Fortran both store their arrays in the same manner.

When an external routine is called via the new CALL mechanism, the program mask is zeroed before the call, and restored to its previous value on return. This is essential because Algol W, by default, runs with fixed point overflow trapping enabled, and this is under the control of the user via EXCEPTION record assignments to pre-declared error control reference variables. The MTS system, on the other hand, initialises fixed point overflow trapping off, and many programs, albeit lazily, do take advantage of this. The University of Michigan Integrated Graphics package is a persistent offender, and recently a case occurred where the MTS READ routine failed for this reason.

With run time checking enabled, a flag is set to enable the error processor to distinguish between program interrupts within Algol W, and those outside. The latter variety then gets a more appropriate form of error processing.

Several extensions to the CALL mechanism have been implemented. These allow a simple register call facility, and also allow subroutine addresses to be passed as integers. Before discussing these it is necessary for us to mention some extended storage control standard procedures which have been provided.

Basically, these routines provide the facilities of the Move Characters (MVC), Load Address (LA) and Translate (TR) instructions of System 370 architecture. They are provided so that data manipulation operations which transcend normal definitions of Algol W types or which access data areas outside Algol W may be achieved.

MOVE provides a byte by byte copy operation from one Algol W variable to another. No type conversion is done. FETCH moves data from an address contained in an integer variable to a second Algol W variable. STORE is the complement of FETCH, sending data from an Algol W variable to a specified address. LOCATE returns the address of an Algol W variable. TRANSLATE translates an Algol W storage region using a supplied table.

MOVE, FETCH, STORE and TRANSLATE take two or three parameters. The third parameter, if supplied, is taken to be an explicit length. If no length is given, the implied length of the variables is used. For MOVE, this will be the minimum of the two possible lengths. Example:

```
STRING(256) DATA;   INTEGER NUM;
... ...
... ...
MOVE(DATA(12|4), NUM);
```

This copies 4 bytes from 12 bytes offset within DATA to the integer NUM. This kind of transfer achieves for Algol W the effect of EQUIVALENCE in Fortran. Direct implementation of EQUIVALENCE in Algol W would be very difficult, if not impossible. LOCATE will be familiar to Fortran users who have used the MTS ADROF (address-of-Fortran-variable) subroutine.

As well as providing the ability to find the address of an internal Algol W variable, a standard function called EXTERNAL has been added to return the address of an external symbol. The address is returned as an integer, and the external symbol name is defined as a literal string in the same manner as for the CALL standard procedure.

Using EXTERNAL, FETCH and STORE together allows a careful user to retrieve and manipulate data in Fortran Named COMMON or BLOCK DATA sections. For some special applications, such as supplying parse tables to a compiler written in Algol W, it would be more practical to generate these tables outside Algol W and access them in this way. The usual course of defining large tables as literals tends, unfortunately, to burst the compiler at the seams, and there is no simple solution to this problem. Algol W was simply not designed with bulk data initialisation in mind.

Returning to CALL, as well as providing for a literal string entry name, the subroutine can be designated by an address held in an integer or bits variable. Suitable manipulation of data via EXTERNAL and FETCH allows the use of subroutine address transfer vectors. Entirely unintentionally, the use of LOCATE on an internal variable allows machine instructions to be issued from within Algol W provided that those instructions (in a bits array, say) form a subroutine. We shan't mention this again as no doubt it will appall some members of staff...

To implement a register call mechanism, RCALL is provided. This is like CALL, but takes no parameters. Before transferring control, it loads general registers zero and one from pre-declared integer variables R0 and R1. These are adjacent in storage and may also be referenced as a pre-declared eight-byte string variable R01.

Clearly we needed to provide a method of recovering the values returned after a function call. To achieve this, after every return from a subroutine invoked by either CALL or RCALL, general registers fifteen, zero and one and floating point register zero are saved. General register fifteen appears in the variable R_CODE as usual, the general register save being achieved by a single store multiple as R_CODE is immediately before R01 in storage. Floating point register zero appears in a new long real variable called R_FLOAT.

For those interested, an example of the code generated by an invocation of the CALL standard procedure is given in Appendix Seven.

Everything so far described in this section has already been implemented and tested. It remains to discuss the question of call back of procedures, and the invocation of Algol W procedures from other languages.

Calling a Fortran subroutine and passing the address of another Fortran callable subroutine as a parameter to the call is already possible by the use of the EXTERNAL standard function. For example:

```
     CALL("FSUB", A, B, EXTERNAL ("QXYZ"), C, D);
```

calls a Fortran subroutine called FSUB for which the third parameter is the address of another Fortran subroutine, QXYZ, which may then be called from FSUB.

The seemingly simple extension to call back Algol W procedures in the same way is in fact several orders of magnitude harder. The compiler modifications to support this extension are already in but some careful work needs to be done in the library before it can work. Call back is provided for a user by adding yet another standard function called LINK. This takes a literal string again, but this time it contains the name of a main code procedure. For safety reasons, LINK is valid only when nested within a call parameter list.

For example:

```
     BEGIN
        REAL PROCEDURE ZFUNCTION(REAL VALUE A,B);
        BEGIN
           ...
           ...
        END ZFUNCTION;
        ...
        ...
        ...
        CALL("FSUB", A, B, LINK("ZFUNCTION"), C, D);
        ...
     END OF PROGRAM.
```

In this example the LINK function call specifies that procedure ZFUNCTION is callable, by an OS Type I sequence, from the Fortran subroutine FSUB. Since the Algol W and Fortran linkage conventions are entirely different, a change of environment needs to be performed. This is achieved by the LINK function call invoking a library routine which sets up an OS Type I callable routine in the Algol W code segment local data stack. The address of this local stack routine is returned as the value of the function, and this is then passed as a parameter to Fortran.

When the Fortran subroutine FSUB calls back the routine which it had been lead to believe is ZFUNCTION, this local stack routine is executed first. It must save FSUB's registers in FSUB's save area. It must then copy the Algol W OS save area from main run time dummy section (DSECT) addressed storage to a safe location in the local stack. Next it calls a second library subroutine passing to it the addresses of the main Algol W DSECT, the first code segment of the Algol W procedure to be called, and of course the FSUB supplied parameter list pointer. The first two addresses have been left in the local stack by the first library routine (the one invoked by LINK). This second library routine then behaves as a section of main code Algol W program, setting up an Algol W call to the procedure. On return the Algol W save area must be restored, any function result values loaded, FSUB's registers restored and control passed back to FSUB.

There are undoubtedly a great number of pitfalls possible in this implementation. The copy of Algol W's save area is required because the re-called Algol W procedure may call another external subroutine, thus overwriting the main save area (which would be used by FSUB on return). Similarly, because of this external call possibility the actual build-up of the parameter list for the call to FSUB needs to be modified. Normally, CALL uses a region set aside for it in the DSECT. Again, this might get re-used so there is an additional rule that if one of the parameters in a CALL is a LINK call, then the parameter list is built in the local stack instead. If the parameter list was always built in the stack, "DATA AREA OVERFLOW" would regularly result so the decision is necessary. Another point is that Algol W needs its program mask back again (another little job for the library routine as it will be in the saved general register three - see Appendix Seven). How the error processor is to handle an error trace back from a procedure called back from Fortran is still giving nightmares.

We are committed to providing this facility as we foolishly promised it to our users some time ago. Basically two non-trivial library routines need to be coded. We will not guarantee that this feature will be in the first new release of the system, but it should follow soon after.

We have had requests from Alberta (via Kathryn Ward) for the ability to call Algol W procedures from Fortran (or from hotter climates...). We do not dismiss this out of hand but we will not be doing it immediately. Our thoughts are that such a facility would require three OS type I routines to be called from Fortran; one to initialize Algol W (provide a stack and I/O system), one to call the procedure (similar to the second library routine required by LINK), and a last routine to shut down Algol W. In this system Algol W would be initialised once, called by procedure invocation as many times as required, and shut down once. These are just thoughts at the moment but Alberta's patience may eventually be rewarded.

8 Compiler System Modifications

Most of the work on the new system concerns the production of new re-entrant Assembler modules forming the run-time system and compiler support system. However, the compiler itself is based on the Stanford original, as developed at Newcastle during 1971-1972 by James Eve and Edwin Satterthwaite.

The compiler consists of two large PL360 modules known as AWA (Phase A - the scan and parse passes) and AWB (Phase B - the code generation pass). Both of these modules were coded re-entrantly in the original version and have been maintained in that happy state ever since. Had they not been so, it is doubtful whether the present work would have been attempted as Newcastle does not have the resources to support a compiler recoding project.

To give an idea of the extent of the updates here are a few figures. Both phases of the compiler are about five thousand lines of PL360 code. Update files tend to appear unjustifyably large to non-combatants, but the cognoscenti will appreciate the extent of the edits involved by the fact that there are three thousand lines of updates. About 60% of these updates are in Phase B.

Over the compilers as a whole several changes have been made. Where possible, coding has been cleaned up and commented when changes were made, in the hope of making the compiler easier to maintain in the future. The compiler communicates with the outside world, in this case the DRIVER and SERVICES modules, via a transfer vector. This contains subroutine addresses and option flags. Reflecting the tendency to move control card functions out to the DRIVER and SERVICES routines, this transfer vector has been extended to 120 bytes to carry extra information.

In all phases of the compiler the error message prologue

```
ERROR xxxx NEAR COORDINATE yyyy
```

has been shortened by the deletion of the word COORDINATE. This should not cause confusion to users and guarantees that most error messages fit on to the screen of the more primitive variety of glass teletype. Also, when running at a terminal, error messages are prefixed by the source for the offending coordinate. This is achieved by the scanner sending each source listing line to a routine in the SERVICES module which then writes a compressed version to a virtual file. When an error message print occurs, the error processor that has gained control requests source lines from this service routine. The lines are reconstituted and passed back for printing. This process is independent of any normal production of a compiler listing in a file, and source is therefore always available for echoing. If the coordinate in the error message is apparently not available, then source for the immediately previous line which is available will be echoed. Tests have shown this facility to be remarkably cheap, and for moderate programs the virtual memory required to store the program listing is not excessive. In a large scale test for a 5000 line program, the compiler took 0.6 seconds to write the compressed listing to VM, and 64 pages of storage were required. The equivalent for a file was 4 seconds (D3.2 MTS - IBM 370/168). When running conversationally a listing file is not generated (by default), and this facility provides a cheap way of pinpointing errors.

There are modifications throughout the compiler phases to allow automatic generation of long real precision object code. This is invoked by the LONG compiler option. When this is specified all declarations of real quantities are treated as if they had been declared long real. Similarly, complex become long complex. This is carried through into the code generation phase where calls to short precision analytic functions such as SQRT generate calls to the routine which implements LONGSQRT. To achieve this, the Name Table entries for pre-declared functions are patched to the required types at Pass One initialisation. When the code for the routine call is being generated, short real functions indicate (by alias entries in a new standard function table) which routine should actually be called. Floating point literal values are automatically assembled to 8-byte precision.

In order to loosen some of the restrictions imposed by the compiler some alteration of the table structure has been incorporated. The segment number for a block or procedure has been moved out of the Name Table to a new dynamically allocated Segment Table. Since only 8 bits were available in the Name Table for this quantity, a maximum segment number of 255 was imposed. This was becoming a serious constraint. Segment numbers can now theoretically be up to 999, although other constraints limit the useful maximum to about 500 - still twice the present value.

Related to segmentation problems was a restriction on the maximum number of external subroutine names which could be referenced by one compilation. This restriction applied to both the ALGOL <string> and FORTRAN <string> constructions and limited the number of different values of <string> to 32. Re-arrangement of the compiler common area has allowed this to be increased to 256. The CALL standard procedure mechanism uses a different table for ESDname storage. Processing of this CALL procedure takes place in Pass Three (code generation). Processing of the ALGOL and FORTRAN strings takes place in Pass Two (the parser) and hence the limit there is for the whole program, as segmentation bounds are not fully decided until the parser completes its task.

Another restriction which prevented the compilation of very large programs was the fixed size Block List Table. The Block List is a table with one entry per block, procedure or FOR statement, and this originally had a maximum size of 256 entries. Alberta raised this figure to 448, and we at Newcastle also accepted this modification. However, it became clear that a more resilient solution was required, since the table was still fixed in size, but had eaten away the last part of the directly addressable compiler common area. Accordingly, modifications were applied to allocate this table dynamically. The Block List is used in conjunction with the Name Table to resolve decisions on identifier scope. As such the table is hammered fairly heavily throughout compilation and this gave rise to concern that a significant speed degradation might result from the extra level of indirection in the addressing. However, much to our relief, tests failed to show any change in timing, and the alteration can be judged a success.

Quite considerable changes of detail have been made to the scanner (Pass One). These changes do not, of course, alter the basic method of conversion of the user's program into a tokenised string. Rather, they implement a series of individually small changes which together, it is hoped, will make the compiler more convenient to use.

As previously mentioned, all control card scanning has been moved out of Phase A into the compiler service routines. Control cards which need to communicate with

Phase A do so by setting up a control record and sending this as data to the scanner, which then deals with it in the normal way. Hence, /TITLE will send a control record containing the address of the new title string. The scanner routine will then reset the listing header buffer to reflect the change.

Error conditions detected whilst reading source records also cause transmission of information to the scanner. In this case it is an error message that is being transmitted. This message is printed in-line in the listing, bracketted between lines of minuses, and is flagged as a Pass One error. Only the error message is written to the virtual listing file (not the lines of minuses), so when the Pass One error is printed, the compiler service routine error message forms the source listing echo.

Two changes to the error processing are important. Firstly, detection of a serious (non-warning) error in Pass One now stops compilation before Pass Two is invoked. Because error messages tend to snowball, the useful information in the Pass Two checking is unlikely to be clearly displayed if the lexical scan has found inconsistencies already. Secondly, where more than one non-warning error message occurs for the same coordinate, only the first is printed. Warning messages are always printed. This technique may seem somewhat arbitrary in its selection of the messages to be displayed, but in practice it has been found to work well, particularly for terminal users. Most programmers require only to be directed to the erroneous coordinate, and so the source code echo is generally more useful than the error message except in the more clear cut cases (such as an ID being undeclared or multiply declared). Usually the error message tends to comment on the effect of the error rather than its cause.

The most noticeable change to the scanner is in the size of the pre-declared identifier tables. For instance, there were 45 standard procedures and functions; now there are 110 of them. This increase in size caused some addressability problems in the Pass One read only data segment and some table re-ordering will be apparent for that reason.

A fast commenting facility has been added. This uses the per cent sign as delimiter. "%" starts such a comment and either the next "%" or the next semi-colon terminates it. The reason for allowing the semi-colon as a terminating delimiter is to prevent an erroneously terminated or unterminated fast comment from swallowing the rest of the program.

Two synonyms of common symbols have been introduced. The double slash (//) is now allowed in place of the vertical bar (|) as a separator in substring designators. The additional reserved word NOT has been added and is allowed in place of the negate symbol (¬). Note that this allows such combinations as "NOT =" for "¬=". For anyone who has ever been confused by the device support routines for typical ASCII terminals, the motives behind this extension will be obvious. Another character which gives trouble at such devices is the underbar (_) but unfortunately there is no other symbol which performs its designated function so well, since it is provided for use as a word break separator within identifiers.

A cross reference listing of identifiers used in the program may now be requested. This facility is implemented through coding in the compiler service routines. If the option is enabled, each identifier look up in the scanner causes the service routine to be called and the identifier and coordinate passed to it. When Pass One is complete, the routine is called again to sort, collate and print the entries. The actual printing is done via a routine in the scanner in order to keep track of the listing pages. The format of the printed output allows two columns of identifiers with up to eight coordinate numbers per half line. As such it is fairly compact. No attempt is made to distinguish different declarations of the same identifier; the information required for this distinction is not available when the collation is done. If printed, the cross reference will appear after the source listing and before any error messages are printed.

Modifications to the parser are few. Mainly they are concerned with making the Block List and Segment Tables dynamic. We should mention here, for the record, that it was necessary to be fairly devious when implementing the modifications to allow segment numbers greater than 255. The parser decides segmentation, forcing a new segment when a procedure or non-trivial block comes along, or when it decides that it is liable to lose track of itself (as with some block expressions). The output from the parser is the tree, or collection of trees, used by the code generator. Because the parser is processing a linear tokenised program to produce the trees, one per segment, it has to suspend and resume processing of particular segments while it deals with more deeply nested ones. When doing this, it pushes data relevant to the segment on to the tree top, and pops this information again when required. The segment number is one of these items, and the original routines provided only eight bits for this purpose. Rather than attempt the fearful task of amending the tree format, four bits were found in the header (previously spare), and the two together provide the required twelve bit segment number.

Standard functions are intercepted on their otherwise smooth passage through the parser to check for the occurence of the LINK standard function. It will be remembered that this is to be used for the future call back of main code Algol W routines from Fortran. The parameter to this routine is a literal string which contains the name of the procedure which would be called back had we implemented the rest of the coding to enable this. Only in the parser may the scope of a variable be checked so here the procedure identifier is looked up. If it is found and passes the checks then the Name Table offset of the procedure is patched into the tree in place of the Literal Table pointer for the string. The code generator can then acquire the information it needs from the Name and Segment tables.

The majority of the modifications to the code generation phase concern changes and additions to the standard function and procedure repertoire. It will be appreciated that they are quite extensive. The standard function case table has been broadened in scope to incorporate check codes for those procedures which require them, and segment numbers for those procedures which invoke library routines (some of these generate in line code). All processing for CALL and related procedures, and for the extended storage control procedures, is contained in this pass.

In the pass three error processor, changes have been made to allow the compiler to continue to generate code if an error is found in a standard procedure. This is sensible if a single erroneous GET call, for instance, is not to cause the entire program to fail. When the erroneous statement is assembled, a call to the run time error processor is built into the object program output to prevent execution of the error path. This action applies only to the newly added pass three errors; the original set remain fatal to the compilation.

References to library routines now have more meaningful names than the anonymous AWXSLnnn external symbol names of the present system. All ESDnames in the Algol W system now begin with AW and are followed by 6 alphabetic characters. This should avoid clashes with any system routine names.

The starter segment AWXSTART, which has already been discussed, is generated at the start of this pass before the Record Table is output.

9 User Documentation

This always seems to be a thorny subject with other installations. Our user manual has also come in for some criticism, probably justified, from our own users, chiefly because it is oriented towards Computer Scientists. Our new approach is based on having two manuals available.

The first will be an introductory document adapted from our programmed introductory text, which was originally written by John Lloyd and Hazel Fells several years ago. Its format is based on the programmed introductory texts for other languages, e.g. Fortran, and it is a no-nonsense introduction to the language designed for engineering and science students. It has proved very popular since its publication. Hazel and John are re-writing this document to provide a general introductory manual to Algol W, with more detailed information on procedures and string handling, so that it becomes a useful first introduction for all users. It is worth noting that while it was originally written for engineers and scientists, most of our Computer Science academic staff prefer their students to use it instead of the Bauer primer in the present User Manual.

The second manual will be adapted from the new University of Michigan MTS Volume 16 ("Algol W in MTS"). UM have kindly allowed us to modify this for our use, and it will become our new "Algol W Reference Manual". As part of the modification it will be converted from TEXT/360, which we do not use here, to TEXTFORM on which we are about to standardize. This new manual is written in clear readable English, and has met with approval from most staff who have so far seen the draft copy in our possession.

The main modification we plan to the UM manual is to add additional chapters describing our extensions to the I/O system, the external subroutine calling linkages and the compiler interface. The I/O extensions will be an extended version of the relevant section of the University of Alberta Computing Science manual, TR15-75, which we also have permission to adapt.

The large amount of documentation made available by Michigan and Alberta will greatly simplify our task of describing the new system for our users. Our thanks go to both of these MTS installations for their generosity.

10 Future Plans

When Release 6.0 is out we would like Algol W to enter a period of stability. This wish is at odds with the requirement for a new release of Algol W to replace completely the current one, because there is no debugging system in the new release. We also wish to provide an MVS Algol W, and we have still to implement facilities for call back of main code Algol W procedures from Fortran.

This last item, the call back facility, is almost certain to be coded soon after Release 6.0 comes out. The lack of this facility is a serious drawback to the use of the English *NAG (Numerical Algorithms Group Ltd.) mathematical subroutine library. Since we recommend this library to our users, and do not wish the overhead of maintaining separate Fortran and Algol W versions, this extension has fairly high priority.

Production of an MVS interface should not be an extensive exercise. Because we have taken care to keep the Algol W system clear of complex routines in MTS, such as the elementary functions and the keyword scanner, the system interface module routines provide basically only an I/O and storage allocation interface. We do not think that these will be difficult to interface to MVS. IBM's Virtual Storage Access Method (VSAM) dataset organisation should make this particularly easy, as it effectively provides MTS line files in MVS. Newcastle would like to produce a version of Algol W for IBM's standard large machine operating system, and although we, its probable implementors, have not yet received the go ahead for this extension, it seems likely that we shall eventually do so. NUMAC will be running an overnight MVS service on our 370/168 from December 1978 or thereabouts.

We have left the most difficult discussion, that of the debug system, to the last. We should emphasise at this point that not only has no decision been made on this part of the work, but as yet no discussion has taken place here regarding it. These paragraphs represent our own thoughts and are in no way a specification of work to be done.

There are two separate questions regarding any debug system developed. One is to provide a post mortem dump option to display variables in the event of an error. An extension of this would be to provide a user callable procedure interface to this giving a snapshot dump and continue facility. However, the second requirement, for serious debugging, would be a fully interactive system similar in nature to the MTS symbolic debugging system (SDS).

The University of Michigan Computing Center have indicated to us they would like to see Algol W debuggable by the use of SDS itself. This entails work to enable the compiler to generate symbol table records (SYM cards). SDS uses these after loading to locate variables for display and statements for the setting of break points. Whilst we agree in principle with the idea of providing an SDS-like debugging system for Algol W, we are against the use of SDS itself to provide this for two reasons.

First, we have gone to a great deal of trouble to provide extensive facilities within the language, at the same time reducing the MTS dependence to a minimum. Use of SDS would re-introduce a high level of MTS specificity into the language system. Since the benefits of this work would then only be available in one of our operating systems, we are opposed to this.

The second reason concerns aesthetic aspects of the use of SDS. SDS was originally developed to debug Assembly language programs and as such it is excellent. Its extension to Fortran provides a good debugging system for that language because of the basically similar structure to (modular) Assembler. However, in our opinion, SDS's extension to block structured languages fails to provide a reliable system on several grounds. For instance, SDS would have difficulty keeping track of identifier scope in its display of (possibly) unavailable variables; display of variables would present an Algol W user with data types very alien to that user, more so than in Fortran; and we feel that display of a program statement should echo the Algol W source for that statement, and not (as SDS would do) display the nearest machine instruction to hand.

To add some constructive criticism to our rejection of SDS, we propose to modify the compiler so that the main tables are dumped as a data control section. The compiler already contains code to output the segment Name Table entries and a Coordinate Table with each program segment. A single extra V-constant address per program segment could associate the correct compiler data segment table with each program segment. As this data would contain both the identifier tables and the tokenised program, most of the facilities of SDS and some it cannot provide would become available by the addition of extra modules to the run-time system. Not all of the details are clear to us yet, but it appears that suitable construction of segment tables, using weak externals for Algol W procedures, could link in the tables for other procedure compilations. Thus we would provide the same debugging facilities in both load and go and object deck generation modes. Also, and we see this as of paramount importance, such a system would be immediately available in any other operating system for which a working system interface module existed.

APPENDIX 1 : Compiler Control Cards

/ALGOLW

The /ALGOLW control card indicates to *ALW that Algol W source text follows and that, unless otherwise overridden, the program is to be executed when (and if) it has compiled. Parameters which may appear on this card are the same set that are valid in the $RUN parameter field, and on /GLOBAL or /OPTIONS control cards. These parameters are described in Appendix Two.

/COMMENT

Source decks can be self-documenting by inserting null operation /COMMENT cards.

/COPY

/COPY is provided to allow source code or data inclusion from files, particularly from Student Batch systems. The single parameter is a file-or-device name. All errors generated by reference to this fdname are fielded, and do not stop the compiler driver run.

/EOF

A "soft" end-of-file indication either to the compiler or (CLG) the run-time system is provided by /EOF. If supplied to the compiler it completes compilation of the current program or procedure source. If CLG or Monitor mode is in force, a succesfully compiled program is loaded, but not executed. Execution can then be started by a /EXECUTE control card. The /EOF card could be used if a data file were to be edited before execution.

/EXECUTE

Execution of a successfully compiled and loaded user program is initiated by /EXECUTE. Any data read by the program via the INPUT stream in CLG mode, or any stream in Monitor mode, should follow this card. Note, however, that any of the control cards are valid in-line and are obeyed if encountered. The /EXECUTE card may appear many times for a particular program, allowing many runs without re-compiling. Parameters on this card are treated as run-time parameters and are processed by the library initialisation routine.

/FIXED

The input mode will be set to FIXED. See the description of this parameter in Appendix Two.

/FLUSH

All processing for the system is brought up to date by /FLUSH so that a HASP Student Monitor system may be sure that all output for a job has appeared. It therefore forces compilation, loading and execution of any pending source text. Any program that is executing is prevented from reading further data from the source stream. After this card has executed, further /EXECUTE cards cause an error message until a new job has been started by a /ALGOLW card.

/FREE

The input mode will be set to FREE. See the description of this parameter in Appendix Two.

/GLOBAL

/GLOBAL takes the same parameters as /ALGOLW and /OPTIONS, but they are not immediately processed. Instead they are processed for each subsequent compilation immediately prior to the /ALGOLW or /OPTIONS card parameters, and hence provide a means of changing the compiler defaults over a series of batched compilations. This control card is not available in Monitor mode.

/INDENT

The compiler listing indentation can be changed during the course of a compilation using /INDENT. By default this is set to 3 spaces per block level, but it may be given any value between 0 and 5. The single parameter on this card is an unsigned integer. If no parameter is found, 3 is assumed.

/LIST

This is the old @LIST compiler directive, and production of compiler listing output is turned on for subsequent source lines. However, if an overriding NOLIST parameter has been given as a compiler option parameter (see appendix 2), this card is ignored.

/MESSAGE

/MESSAGE is similar to /COMMENT in that it does not affect compilation or running of the system, but any characters following the pseudo command will be printed on the ERROR unit. In MTS this is SERCOM. If no characters are present a blank line is printed.

/MONITOR

If this card is encountered in the initial control card group and before a /ALGOLW card has been processed, Monitor mode is set. This is the only way to set Monitor mode on and once in effect it cannot be disabled. It has certain implications for users in that it is designed to implement a fail-safe student system. Some control cards, such as /GLOBAL, /OBEY and /STOP, are disabled. External procedures may not be linked in, and all I/O is routed via the INPUT and OUTPUT streams.

/NEWPAGE

This card causes the compiler to start a new page on the compiler source listing. The title string remains unchanged.

/NOINDENT

Automatic indentation in the source listing is turned off with /NOINDENT. It is equivalent to /INDENT,0.

/NOLIST

This is the old @NOLIST compiler directive: it turns off production of compiler listing output for subsequent source lines. However, if an overriding LIST parameter has been given as a compiler option parameter (see Appendix two), this card will be ignored.

/OBEY

Any characters appearing on this card are presumed to compose a system command and are passed on the operating system in use, if that system supports command processing. In MTS this would be an MTS command, thereby allowing a user access to the Editor and other file system utility commands. /OBEY is not allowed in Monitor mode.

/OPTIONS

The /OPTIONS control card indicates to *ALW that Algol W source text follows, but, by default, an object deck should be generated. Otherwise it is like /ALGOLW. Parameters which may appear on this card are the same set as those that are valid in the $RUN parameter field, and on /ALGOLW or /GLOBAL control cards. These parameters are described in Appendix Two.

/SEQUENCED

The input mode will be set to SEQUENCED. See the description of this parameter in Appendix Two.

/SPACE

/SPACE is similar in action to the Assembler SPACE operation, but also includes a conditional page skip facility. It takes zero, one or two parameters. If given, any parameters must be unsigned integers. The first parameter, which defaults to 1, is the number of lines to skip on the listing. The second parameter, if given, is the number of lines which should be remaining on the current listing page after the spacing operation. If there are fewer lines than this left, a new page is forced instead.

/STOP

/STOP causes the Algol W system to shut down. Any program which is pending is compiled, and in CLG mode it is executed. The card is treated as /FLUSH in Monitor mode to prevent users shutting down the Monitor. Only a genuine end-of-file will stop the Monitor.

/TITLE

This is the old @TITLE directive. Its purpose is to place a string of from 1 to 35 characters in the listing header line. If no parameter is given the title string is cleared. /NEWPAGE can be used to force a new listing page without deleting the title string.

APPENDIX 2 : Compiler Parameters

AUTOSKIP; NOSKIP

If AUTOSKIP is in effect, one blank line is left on the listing before any procedure header line is printed, thus automatically separating procedures. NOSKIP disables this action. AUTOSKIP is the default.

CHECK; NOCHECK

These parameters provide the facility formerly available via the @NOCHECK directive. NOCHECK suppresses the generation of code to check array subscripting, case selection indexing, substring indexing and reference binding at run time. CHECK is the default.

CODE

If specified, a pseudo assembly listing is produced for each code segment generated. It was formerly @DUMP*,1

CONCHAR=character

The control card start character can be changed from the standard "/". CONCHAR=@ would allow a program containing @LIST, @NOLIST or @TITLE to be compiled.

ECHO; NOECHO

ECHO causes the compiler scanner to store a compact version of the listing for use in error diagnostic printing, should any errors be detected. ECHO is the conversational default, in batch it is NOECHO.

FIXED

Card image source follows - see FREE description.

FREE; FIXED; SEQUENCED

The compiler interface, by default, accepts program source input lines of any length - this is the action of the FREE parameter. Lines longer than 72 bytes are split into 72-byte chunks and fed to the compiler scanner. FIXED and SEQUENCED are provided to allow checking of card image source records. FIXED assumes a maximum visible length of 72 bytes on a card and will flag longer records as an error. SEQUENCED is similar, but assumes an 80 byte maximum, and that bytes 73-80 contain a card sequence identifier.

GENERATE

Undoes the effect of the SYNTAX parameter. (See below.)

GO; NOGO

As laid down by the 2nd MTS Workshop, a program for immediate execution must not require control cards. GO marks a program for execution if successfully compiled. The default NOGO parameter causes an object deck to be generated.

ID=1-to-8-characters

This keyword expression is used to supply a user identifier on a /ALGOLW card. Monitor mode insists on this parameter being given, and will call a system interface routine to verify the ID. Only alphanumerics are allowed.

INDENT; INDENT=integer; NOINDENT

These parameters control source listing indentation. Indenting is enabled, by default, to three spaces per block level, but it may be varied between zero and five spaces per block level. If the calculated indent for any source listing line exceeds 32, then it is overridden to 32. INDENT is equivalent to INDENT=3. Similarly, NOINDENT is equivalent to INDENT=0.

LIBRARY=filename(s)

The names of any run time object files or library can be stored with the source program. An INC loader record is generated as part of the object code.

LINESPERPAGE=integer, LPP=integer

The lines-per-page on the source listing may be varied by the use of this keyword parameter between 25 and 100. The default is 60.

LIST, L

Production of a compiler listing is forced by LIST and any /NOLIST cards in the deck will be ignored. The parameter is related to SOURCE, PRINT and NOLIST, and it is never the default.

LISTING=filename

When run at a terminal, *ALW will produce a source listing in a file if SOURCE or LIST is given. The default filename is -AWLIST. This keyword parameter can change the filename. Note that the file is emptied before use, and that if the PRINT stream has been assigned (SPRINT in MTS) then the assigned file-or-device will be used in preference. Specifying this parameter implies SOURCE.

LONG; SHORT

*ALW has the ability to convert all 32-bit floating point entities to 64-bit precision, and it is done by specifying LONG. The default is SHORT, which does no conversion.

MAP

Causes the system loader to print a module address map after loading the successfully compiled program in CLG or Monitor mode.

NOCHECK

No run time checking - see CHECK.

NOECHO

No error source listing echo - see ECHO.

NOGO

Set Deck Generation mode - see GO.

NOINDENT

No indenting of compiler listing - see INDENT.

NOLIST

This option will suppress any compiler listing, ignoring any /LIST cards it may encounter. It is the default when running conversationally if the PRINT stream (MTS - SPRINT) is not assigned explicitly.

See also the LIST, SOURCE and PRINT parameters.

NONUMBER

No line numbers on the listing - see NUMBER.

NOSKIP

Does not separate procedures on the listing - see AUTOSKIP.

NOTEST

No interactive debug table generation - see TEST.

NOXREF

Suppresses any cross reference listing - see XREF.

NUMBER; NONUMBER

These parameters control the display of the source line number on the listing. NUMBER is the default, except in Monitor mode where NONUMBER is enforced.

OBJECT=filename

If *ALW is generating an object deck, it will be produced on the PUNCH stream (MTS - SPUNCH), if it is assigned. If it is not, then, by default, the file -AWLOAD is used. This keyword parameter allows the default filename to be changed. Note that the file is emptied before use.

PAGES=integer, P=integer

Set the execution printed pages limit.

PRINT

A compiler listing is produced as if the SOURCE parameter had been given, but the assignment of the PRINT stream (SPRINT in MTS) is not checked. The listing is produced on PRINT even if it has not been explicitly assigned.

RUNPARS=(run-time-parameter-string)

The RUNPARS keyword parameter allows a set of run-time parameters to be specified at compile time. They are compiled into the starter segment AWXSTART, and hence this parameter only has meaning for main programs. At run-time, the string is processed before any run parameters, so the effect is to change the defaults. It should be particularly useful to users who have programs creating large arrays. A single run parameter may follow the equals sign; if several are given they should be specified within parentheses.

SEQUENCED

See the description of FREE.

SHORT

See the description of LONG.

SIZE=integer(K|P)

Specifies the compiler working storage area size.

SOURCE, S

The SOURCE parameter causes the compiler to produce a listing, but /LIST and /NOLIST control cards along the way will be obeyed. If the PRINT stream (MTS - SPRINT) has not been assigned, it will be defaulted to either -AWLIST or the name given with the LISTING parameter.

STACK

This causes the compiler to print a stack dump after a syntax error has been detected.

SYNTAX; GENERATE

SYNTAX is the parameter which has the same action as the old @SYNTAX directive - check the program only, do not generate object code. GENERATE negates this, and is the default.

TABLES

A listing of the main compiler internal tables is produced. It is the old @DUMP*,3 directive.

TEST; NOTEST

Produces interactive debug tables. Actually it will not do so yet, but the thought is there. When and if we implement a debug system, this parameter will invoke it.

TIME=integer(S|M), T=integer(S|M)

execution time limit

TRACE

Produces code, tables and tree listing - this is the old @DUMP*,7 directive.

XREF, X; NOXREF

XREF produces a cross reference listing, in two column format, after the source listing and before any error messages are printed.

APPENDIX 3 : Standard Procedures

This Appendix provides basic information for all the standard procedures which will be available in Release 6.0 of Algol W. Standard procedures were originally provided to implement the input/output system via READ(ON) and WRITE(ON). As such they allow parameters to be any simple type identifier or expression - that is, any scalar quantity. It has been possible to implement all of the extensions to the I/O system and provide an extended external call linkage using variations of the standard procedure linkage. No change to the language itself was required.

Some standard procedures generate calls to library routines while others generate in-line code. This difference is not apparent to a user calling them.

In the following descriptions of standard procedures the character preceding the name indicates the status of the identifier:

*	newly added in Release 6.0
+	significantly changed or extended
-	essentially unchanged from previous releases
$	deleted in this release

* ASSIGN(stream, file-or-device-name)

ASSIGN is used to allocate, or re-allocate, an I/O stream. The given file-or-device-name is assigned to the stream, and it will be opened when the stream is next used for I/O. Note that any file-or-device-name valid in the Operating System in use may be specified: in MTS this means that modifiers, line number ranges and explicit concatenation are all allowed.

* ATTNTRAP(logical-value)

Simple attention interrupt processing is provided through the use of this standard procedure. ATTNTRAP(TRUE) enables the trap and ATTNTRAP(FALSE) disables it again: by default the trap is not enabled. When trapping is on, attention interrupts do not halt execution of the program, and this allows sensitive parts of a program to be executed without risk of interruption. A program may find out if an interrupt has occurred by inspecting the pre-declared logical variable ATTNMARK. This is initially false, but is set to true when an attention interrupt is fielded.

* CALL(ESDname, optional-subrtn-parameters)

The standard procedure mechanism is used to implement a new IBM OS Type I subroutine linkage. The ESDname parameter must be either a literal string giving the subroutine name, or an integer or bits variable containing the address to which control should be transferred. If parameters are given, they may be of any type that is allowed for standard procedures. Hence they can be literals, simple variable IDs or simple expressions. The procedure stores the return values of the contents of general registers fifteen (in R_CODE), zero (in R0), one (in R1) and of floating point register zero (in long real R_FLOAT). Note that the integers R0 and R1 are available as an eight byte string R01.

* CONTROL(stream, control-parameter-strings)

CONTROL gives a programmer access to the operating system file-or-device control facilities. For instance, in MTS, if the stream were assigned to a magnetic tape then CONTROL could be used to set the block size, write tape marks, position the tape, etc.

* EMPTY(stream)

If the specified stream is attached to a file then Algol W will attempt to empty it, that is, the file is left in a state such that all information previously held becomes inaccessable. Attempts to read an empty file produce an end-of-file indication, and the next write would place a new first record in the file. In MTS, this can also be achieved via a CONTROL operation, but a separate standard procedure is provided to aid user program readability.

* FETCH(source-address, destination, optional-length)

This is one of the the extended storage control routines. Its purpose is to fetch bytes of data from a source address, given as the value of an integer or bits parameter, into an Algol W variable of any simple type. If no length parameter is given, then the implied length of the destination parameter is used.

* FLUSH(stream)

FLUSH causes the contents of the output buffer for the stream, if any, to be written out. It has the same effect on the specified stream as IOCONTROL("NEWLINE") would have on the current writer stream.

* GET(stream, format, input-list)

GET and GETON are the main Alberta formatted input procedures. Data is read into the items specified in the input list from the given stream under control of the supplied format string. The format strings are Fortran-like in appearance, with certain changes and extensions for their Algol W implementation. GET will fetch a new input record. GETON continues to decode the current input record for the stream, continuing from where the last operation left off. The presence of a slash (/) in the format string will cause a new input record to be fetched.

* GETCARD(stream, input-string-list)

GETCARD provides a procedure similar to READCARD, but in which the first parameter specifies the input stream to be used in the operation. One important difference is in the handling of end-of-file conditions. GETCARD bypasses the mechanism using the pre-declared ENDFILE reference; instead it sets a logical variable FILEMARK after each operation.

* GETON(stream, format, input-list)

See the description of GET.

* GETSTRING(source-string, format, input-list)

This procedure provides the means to perform input conversion operations from data records held in an Algol W string. Its operation is similar to that of GET, but the first parameter gives the source string holding the data rather than an input stream name.

+ IOCONTROL(integer-or-string-control-key)

IOCONTROL has been extended in operation to allow both integer and string codes to be given, and more options are provided. These options are:

1	NEXTCARD	Next read fetches new input record
2	NEWLINE	Next write starts a new record
3	NEWPAGE	Next write starts a new page
4	NORMAL	Generate ANSI cc chars

5	FULLPAGE	Generate "9" and ";" cc chars
6	DOUBLESKIP	Next write skips two lines
7	TRIPLESKIP	Next write skips three lines
8	OVERPRINT	Next write overprints line

If the key is given as a string, unambiguous abbreviation down to a minimum of three characters is allowed.

* LOCATE(item, address-destination)

A user program may obtain the address of an Algol W identifier or field identifier using this procedure. The address of the first parameter is placed in the second parameter which should be an integer or bits variable.

* MOVE(source, destination, optional-length)

This is one of the the extended storage control routines, its purpose is to move bytes of data from one Algol W variable to another without doing any type conversion. The variables may be of any simple type. If no length parameter is given, then the minimum of the two implied parameter lengths will be used. This procedure provides, via a copy operation, some of the flexibility of the Fortran EQUIVALENCE construction.

* NEWLINE(string-or-integer-key)

NEWLINE provides the same features as IOCONTROL, but in an active rather than passive way. IOCONTROL determines the action that should be taken when the next write takes place (for keys other than one). This action can be overridden if another call to IOCONTROL is found before the next write. NEWLINE, on the other hand, acts in such a way that every call to it is visible. If an integer key is given, then that number of new lines is produced, to a maximum of forty. A request for zero or fewer lines will cause a single overprinted line to be started. If a string is supplied, then the first character of the string is used as the carriage control character of a new output line. The difference in action means that two IOCONTROL(2) statements produce one new line whereas two NEWLINE(1) statements produce two new lines.

* OBEY(system-command-string)

For operating systems which support a command processor, OBEY allows commands to be issued from within a program. The entire string given is supplied as the command.

* PROTECT(stream)

  If the stream is attached to a file, PROTECT will cause all changes to the file buffers to be written back to the disk copy of the file, so protecting the information from system damage occurring at a subsequent time in the program run. Without this protection, such a crash before the file was released could result in information being lost.

* PUT(stream, format, output-list)

  PUT and PUTON are the main Alberta formatted output procedures. Items specified in the output list are written to the given stream under control of the supplied format string. These format strings are Fortran-like in appearance, with certain changes and extensions for their Algol W implementation. PUT starts a new input record; PUTON continues to append to the current output record for the stream, continuing from where the last operation left off. If the maximum output length of the stream will be exceeded by adding another item then a new output record is started. A new record may also be forced by inserting a slash (/) in the format string.

* PUTCARD(stream, output-string-list)

  PUTCARD provides a procedure similar to WRITECARD, but one in which the first parameter specifies the input stream to be used in the operation.

* PUTON(stream, format, output-list)

  See the description of PUT.

* PUTSTRING(string-destination, format, output-list)

  This procedure provides the means to perform output conversion operations directly into an Algol W string. It is similar to PUT in operation, but the first parameter gives the destination string which is to receive the output rather than the output stream name.

* QUALIFY(stream, qualify-keyword-list)

  QUALIFY is related to CONTROL in that it allows a user to change the attributes of an I/O stream. However, while CONTROL communicates with the operating system, QUALIFY sets those attributes which are local to Algol W. For instance, it is called to change the input and output lengths of I/O streams.

* RCALL(ESDname)

RCALL provides a simple call facility in which parameters are supplied to the called routine by loading them into the machine's general registers. The ESDname parameter is as described in the description of the CALL standard procedure. Parameters are loaded from the pre-declared integer variables R0 and R1 (also available as an 8-byte string R01. As implied by their names, they are used to load general registers zero and one. On return, the contents of these registers are stored in the same variables. Additionally, as for CALL, the value of general register fifteen is saved in R_CODE and that of floating point register zero in the long real R_FLOAT.

- READ(input-list)

The basic input routines READ and READON are unchanged in their mode of operation. However, many of the restrictions on the format of data items have been relaxed. As before, READ starts a new record while READON does not. Either routine will fetch a new input record if there are no further data items in the current one.

+ READCARD(input-string-list)

This procedure, which was originally implemented to read card images into 80-byte strings, has been modified to accept destination strings of any length.

* READER(stream)

There are two stream switching procedures READER and WRITER). The use of READER forces the specified stream to become the current input stream. The next READ, READON or READCARD will use this stream and fetch a new input record from it.

- READON(input-list)

See the description of READ.

* RELEASE(stream)

  This procedure is used to free the resources attached to an I/O stream. If the I/O stream is one of the pre-defined ones (for example one of the logical device numbers 0 to 19), then any remaining information in the output buffers is written to the file-or-device attached and it is then closed down and freed. If the stream is a user defined one (using the ASSIGN procedure), then, in addition, it is deleted from Algol W's internal tables.

* REWIND(stream)

  As the name implies, the specified stream will be rewound so that subsequent reads and writes start at the beginning of the file or device attached. Note that in MTS only the currently active member of a series of concatenated files-or-devices will be rewound. An attempt to rewind a stream which may not be rewound (such as a card reader) will cause a fatal error.

* SENSE(stream, sense-keys, returned-info-list)

  SENSE will be used to return information about the state of I/O streams. The exact form of the keywords has yet to be decided, but information such as the filename, control block address and feasibility of operations such as indexing and rewinding will be available. It is similar to GET in form, but appears to do I/O from the system control block itself rather than from attached files-or-devices.

* STOP(string-to-print-or-NULL)

  Invoking this procedure causes the execution of the program to be terminated. If a string is given as a parameter it will be printed on the ERROR stream.

* STORE(source, destination-address, optional-length)

  This is another of the the extended storage control routines. Its purpose is to store bytes of data from an Algol W variable of any simple type at a destination address which is given as the value of an integer or bits parameter. If no length parameter is given, then the implied length of the source parameter is used.

+ TRACE(any-parameter)

Calls to TRACE do nothing in this version. Eventually, should we implement an interactive debugging system, it will form a user program entry to that system.

* TRANSLATE(source, translate-table, optional-length)

This is the last of the the extended storage control routines. It is provided to give Algol W user programs direct access to the machine Translate instruction. For each byte in the source parameter, a byte is fetched from the translate table at an offset corresponding to the numerical EBCDIC value of the source byte. The byte fetched from the table replaces the byte in the source parameter. This implies that the translate table will normally be assumed to be 256 bytes in length. If no length parameter is given, then the implied length of the source parameter is used. This procedure can perform many varied actions: as well as translation of character codes it can be used to re-arrange the bytes of a string according to a supplied pattern.

- WRITE(output-list)

The basic output routines WRITE and WRITEON are unchanged in their mode of operation. However, some improvements and changes in the output conversions have been added. Floating point values may be output with any desired exponent separator (using the pre-declared variable R_EXPCHAR). Complex values are output in the form (realpart, complexpart), which is acceptable as input. As before, WRITE starts a new record while WRITEON does not. Either routine will start a new output record should the output buffer overflow when an item is converted.

* WRITECARD(output-string-list)

This procedure is added as the complement of READCARD. Although its name implies card images, it will, in fact, output strings of any length. Any previous contents of the writer stream output buffer is written. The string is transferred to the buffer, and then it too is output. In this it differs from the action of WRITE on the same string, because WRITE does not flush the information out until the buffer is full, or a request to do so has occurred. In effect, WRITECARD is a WRITE with an implicit IOCONTROL("NEWLINE") associated with it.

- WRITEON(output-list)

  See the description of WRITE

* WRITER(stream)

  This is one of the two stream switching procedures (the other being READER). It causes the specified stream to become the current output stream, and the next WRITE, WRITEON or WRITECARD will start using this stream. Any record partially built for the previous writer stream will be output at the time of the switch.

* XDELETE(stream, index-list)

  Indexed deletion of lines from a file attached to a stream is performed by this procedure. The index list is a list of integer line numbers to be deleted from the file. In MTS, the internal form of the line number, 1000 times the external form, is used.

* XGETCARD(stream, index, input-string-list)

  This entry provides an indexed form of the GETCARD procedure. Instead of reading the next record in sequence, the record is fetched from the line number given by the index parameter. Line numbers are as described for XDELETE. If no information is present at the given line, it is treated as an end-of-file condition and dealt with as for GETCARD. If more than on string is specified in the list, only the first one is fetched from the indexed line. The second and any subsequent records are fetched serially after the first.

* XPUTCARD(stream, index, output-string-list)

  This entry gives an indexed form of the PUTCARD procedure. Records are index written using the line number given in the index parameter. Line numbers are described under XDELETE. If a record is already present at the given line, it will be replaced by the new one. If more than one string is given, only the first will be index written. The remainder will be serially written following the indexed write.

APPENDIX 4 : Standard Functions

This Appendix provides basic information for all of the standard functions which will be available in Release 6.0 of Algol W. Standard functions are provided to implement standard functions of analysis, transfer functions for data type conversion, and one or two special routines. All have one scalar argument and produce a single scalar result. The types of the argument and result are pre-defined to one combination only.

Some standard functions generate calls to library routines while others generate in-line code. This difference is not apparent to a user calling them. It is worth noting, however, that specification of the LONG compiler option will cause automatic generation of calls to long real and long complex functions when the names of the real and complex ones are given. For instance, SIN would cause a call to LONGSIN.

In the following descriptions of standard functions the character preceding the name indicates the status of the identifier:

*	newly added in Release 6.0
+	significantly changed or extended
-	essentially unchanged from previous releases
$	deleted in this release

```
* real procedure ARCCOS(real value ARG);
* long real procedure LONGARCCOS(long real value ARG);
```

The inverse cosine function.

```
* real procedure ARCSIN(real value ARG);
* long real procedure LONGARCSIN(long real value ARG);
```

The inverse sine function.

```
- real procedure ARCTAN(real value ARG);
- long real procedure LONGARCTAN(long real value ARG);
```

The inverse tangent function.

```
- string(12) procedure BASE10(real value ARG);
```

The real argument is returned as the basic formatted string S+EE-DDDDDDD .

- string(20) procedure LONGBASE10(long real value ARG);

  The long real argument is returned as the basic formatted string S+EE-DDDDDDDDDDDDDDD .

- string(12) procedure BASE16(real value ARG);

  The real argument is returned as the basic hexadecimal formatted string SS+BB-AAAAAA .

- string(20) procedure LONGBASE16(long real value ARG);

  The long real argument is returned as the basic hexadecimal formatted string SS+BB-AAAAAAAAAAAAAA .

- bits procedure BITSTRING(integer value ARG);

  The integer argument is returned as the equivalent bits value.

- string(1) procedure CODE(integer value ARG);

  The string returned is the EBCDIC character corresponding to the value of the argument REM 256 .

- real procedure COS(real value ARG);
- long real procedure LONGCOS(long real value ARG);

  The cosine function.

* real procedure COSH(real value ARG);
* long real procedure LONGCOSH(long real value ARG);

  The hyperbolic cosine function.

* real procedure COT(real value ARG);
* long real procedure LONGCOT(long real value ARG);

  The cotangent function.

* complex procedure CXCOS(complex value ARG);
* long complex procedure LONGCXCOS(long complex value ARG);

  The complex cosine function.

* complex procedure CXEXP(complex value ARG);
* long complex procedure LONGCXEXP(long complex value ARG);

  The complex exponential function.

* complex procedure CXLN(complex value ARG);

*   long complex procedure LONGCXLN(long complex value ARG);

  The complex natural logarithm function.

*   complex procedure CXSIN(complex value ARG);
*   long complex procedure LONGCXSIN(long complex value ARG);

  The complex sine function.

*   complex procedure CXSQRT(complex value ARG);
*   long complex procedure LONGCXSQRT(long complex value
ARG);

  The complex square root function.

*   string(24) procedure DATE(integer value ARG);

  The time and date are returned as a string in a format determined by the integer argument.

-   integer procedure DECODE(string(1) value ARG);

  This integer value returned corresponds to the position of the argument character in the EBCDIC ordering.

-   integer procedure ENTIER(real value ARG);

  One of the three real to integer conversion functions. The others are ROUND and TRUNCATE.

*   real procedure ERF(real value ARG);
*   long real procedure LONGERF(long real value ARG);

  The error function.

*   real procedure ERFC(real value ARG);
*   long real procedure LONGERFC(long real value ARG);

  The complementary error function.

-   real procedure EXP(real value ARG);
*   long real procedure LONGEXP(long real value ARG);

  The exponential function.

-   integer procedure EXPONENT(real value ARG);

  The integer value of the machine representation exponent of the real argument is returned.

*   integer procedure EXTERNAL(string(8) value ARG);

This entry returns the address of an external symbol. The symbol must be given as a literal string of between one and eight characters, with no embedded blanks. The purpose of this function is to allow a user access to data control and Fortran named common sections, when used in conjunction with the STORE and FETCH standard procedures.

* integer procedure FULLWORD(integer value ARG);

This procedure assumes that the integer argument consists of a halfword integer in bits 0-15 and no relevant information in bits 16-31. The result is produced by a 16 bit Shift Right Arithmetic operation.

* real procedure GAMMA(real value ARG);
* long real procedure LONGGAMMA(long real value ARG);

The gamma function.

* integer procedure HALFWORD(integer value ARG);

This procedure assumes that the integer argument is in the range -32768 <= ARG <= 32767 . The halfword result is produced by a 16 bit Shift Left Arithmetic operation. An exceptional condition will be recognised if the argument is outside the expected range.

- complex procedure IMAG(real value ARG);
- long complex procedure LONGIMAG(long real value ARG);

The real argument is returned as a complex value.

- real procedure IMAGPART(complex value ARG);
- long real procedure LONGIMAGPART(long complex value ARG);

The imaginary part of a complex quantity is returned as a real value.

- string(12) procedure INTBASE10(integer value ARG);

The integer argument is returned as the basic formatted string S-DDDDDDDDDD .

- string(12) procedure INTBASE16(integer value ARG);

The integer argument is returned as the basic unsigned hexadecimal formatted string SSSSAAAAAAAA .

* integer procedure LINK(string(64) value ARG);

LINK will provide, in a subsequent release, a call back facility which will enable a Fortran subroutine to call a supplied main code Algol W procedure. The argument will be a literal string giving the name of the procedure which may be called. LINK will only be valid when issued from within a CALL standard procedure parameter list, for safety reasons.

- real procedure LN(real value ARG);
- long real procedure LONGLN(long real value ARG);

  The natural logarithm function.

* real procedure LNGAMMA(real value ARG);
* long real procedure LONGLNGAMMA(long real value ARG);

  The natural logarithm of the gamma function.

- real procedure LOG(real value ARG);
- long real procedure LONGLOG(long real value ARG);

  The logarithm to the base ten function.

- integer procedure NUMBER(bits value ARG);

  The value returned is the integer corresponding to the bits argument.

- logical procedure ODD(integer value ARG);

  Returns a logical value of TRUE if the argument is odd and FALSE if it is even.

- real procedure REALPART(complex value ARG);
- long real procedure LONGREALPART(long complex value ARG);

  The real part of a complex quantity is returned as a real value.

- integer procedure ROUND(real value ARG);

  One of the three real to integer conversion functions. The others are ENTIER and TRUNCATE.

- real procedure ROUNDTOREAL(long real value ARG);

  The properly rounded real value of the long real argument is returned.

- real procedure SIN(real value ARG);
- long real procedure LONGSIN(long real value ARG);

The sine function.

```
*  real procedure SINH(real value ARG);
*  long real procedure LONGSINH(long real value ARG);
```

The hyperbolic sine function.

```
-  real procedure SQRT(real value ARG);
-  long real procedure LONGSQRT(long real value ARG);
```

The square root function.

```
*  real procedure TAN(real value ARG);
*  long real procedure LONGTAN(long real value ARG);
```

The tangent function.

```
*  real procedure TANH(real value ARG);
*  long real procedure LONGTANH(long real value ARG);
```

The hyperbolic tangent function.

```
+  integer procedure TIME(integer value ARG);
```

The TIME function has been extended to provide additional options. Those now available are:

-2	Elapsed time, 1/60 sec
-1	Time of day, 1/60 sec, since midnight
0	Total CPU time, 1/100 min
1	Total CPU time, 1/60 sec
2	Total CPU time, 1/38400 sec
3	Problem state CPU time, 1/38400 sec
4	Supervisor state CPU time, 1/38400 sec

All items other than -1 are times since the program started execution.

```
-  integer procedure TRUNCATE(real value ARG);
```

One of the three real to integer conversion functions. The others are ENTIER and ROUND.

APPENDIX 5 : Pre-declared Variables

This appendix gives a list of all pre-declared identifiers in Release 6.0 of Algol W, other than standard procedures and standard functions which are dealt with in the two previous appendices.

In the following descriptions of pre-declared variables the character preceding the name indicates the status of the identifier:

*	newly added in Release 6.0
+	significantly changed or extended
-	essentially unchanged from previous releases
$	deleted in this release

* integer A_COUNT

This is initialised to zero, and incremented by one each time an ASSERT statement is encountered during execution.

* logical ATTNMARK

The ATTNTRAP standard procedure is used in conjunction with this variable. It is initialised to FALSE when trapping is enabled by calling ATTNTRAP(TRUE). When an attention interrupt occurs ATTNMARK becomes TRUE.

* logical CANREPLY

This variable indicates whether a user program is running in conversational or batch mode. It is TRUE for conversational.

- reference(EXCEPTION) DIVZERO

Floating point divide by zero exceptions are intercepted using DIVZERO.

- reference(EXCEPTION) ENDFILE

End-of-file exceptions are intercepted using ENDFILE. Note that two of the new entries, GETCARD and XGETCARD, bypass this exception mechanism and set the FILEMARK pre-declared logical variable only.

- real EPSILON

The largest real number which, when added to 1.0, gives a sum which is still 1.0 .

- record EXCEPTION

  This is the only pre-declared record, and is used in exception processing. XCPNOTED, XCPLIMIT, XCPACTION, XCPMARK and XCPMSG are field identifiers of this record, and all exception processing interception variables are references to it.

$ reference(EXCEPTION) EXPERR

  Formerly used to intercept exceptions in the evaluation of the EXP and LONGEXP standard functions, it has now been deleted from the new system and its place taken by a new elementary function exception reference named FUNCTION.

$ integer FIELDSIZE

  This was a synonym of R_W. It has never been documented, and it is deleted in the new release.

* logical FILEMARK

  Every input operation sets this variable to be TRUE if an end-of-file occurs and FALSE if it does not. However, unless the ENDFILE exception reference has been suitably assigned, the Algol W error processor will normally take control. If ENDFILE is assigned to NULL, inspection of FILEMARK becomes the only way of detecting an end-of-file.

* long real FN_VALUE

  Under control of XCPACTION(FUNCTION), this variable may be used to provide a replacement value for an elementary function which has encountered an exceptional condition. If it supplies a replacement for a complex or long complex function, the imaginary part will be zero.

* reference(EXCEPTION) FUNCTION

  This variable is being introduced to provide interception for all elementary function exceptional conditions. As a result, the former reference variables EXPERR, LNLOGERR, SINCOSERR and SQRTERR have been deleted. The interception they provided, and that for all of the newly introduced elementary functions, is now vested in FUNCTION. One of the options provided via XCPACTION(FUNCTION) settings will be to load a replacement value from a pre-declared variable FN_VALUE.

$ integer I_LENGTH

This variable has been in the system for a long time, but has never been used and has therefore been deleted from the system. The facility it was intended to provide (control of the input buffer visible length) will be supplied by the QUALIFY standard procedure.

- integer I_W

The integer output field width in the Newcastle formats.

- reference(EXCEPTION) INTDIVZERO

Interception of the integer division by zero exception is performed using INTDIVZERO.

$ integer INTFIELDSIZE

This was a synonym of I_W. It has not been in the documentation since 1973 and the opportunity is being taken to delete it in the new release.

- reference(EXCEPTION) INTOVFL

Interception of the integer overflow exception uses INTOVFL. Note that Algol W will trap this condition by default, whereas most operating systems (including MTS) do not. This has caused problems in the past but the new CALL and RCALL external linkages zero the program mask before they call the subroutine designated. Assignments to this reference are no longer necessary to circumvent such problems.

$ reference(EXCEPTION) LNLOGERR

Formerly this reference was used to intercept exceptions resulting from calls to the LN, LONGLN, LOG and LONGLOG standard functions. It has now been deleted and its place is taken by a new elementary function exception reference named FUNCTION.

* long real LONGEPSILON

The largest long real value which, when added to 1.0L, gives a sum which is still 1.0L .

- integer MAXINTEGER

  The largest integer value the machine can support. It is 2**31 - 1 .

- long real MAXREAL

  The largest floating point value which can be represented on the machine. It is of the order of 7.237 * 10**75 .

$ integer O_LENGTH

  This variable has been in the system for a long time, but has never been used and has therefore been deleted. The facility it was intended to provide (control of the output buffer maximum length) will be supplied by the QUALIFY standard procedure.

- reference(EXCEPTION) OVFL

  This allows floating point exponent overflow exceptions to be trapped.

- long real PI

  3.14159... etc.

- integer R_CODE

  The return code from FORTRAN construction external subroutine calls is found in general register fifteen. R_CODE is provided to receive this, and it now also performs the same function for the linkage via the CALL and RCALL standard procedures.

- integer R_D

  The decimal digits field width for floating point output using the Newcastle "A" format is in R_D. Initially it is zero.

* string(1) R_EXPCHAR

  The exponent separator with floating point output is initially the prime (') but may be set to "E" or "D" if the output items are to be re-input by a Fortran program. R_EXPCHAR should be set to the required character.

* integer R_FIXED

This is a synonym of R0. See the description of R0.

* long real R_FLOAT

On return from an external routine invoked by the CALL or RCALL standard procedures, the value of floating point register zero is saved in this variable. Hence any real or long real result from a function procedure may be retrieved.

- string(1) R_FORMAT

The format designator for the Newcastle formats may be "F" (for Freepoint), "S" (for Scaled) or "A" (for Aligned decimal point). The default is "F" and it is changed by setting R_FORMAT. Some changes have been made to the Freepoint output to make it tidier. Complex values are now output as (A,B) rather than as A+BI: this enables complex values to be re-input.

* integer R_W

The total field width for floating point numbers output using the Newcastle formats.

* bits RDR

This variable is provided so that standard procedures such as GET which specify explicit I/O streams may refer to the current reader stream.

* integer R0

R0 is referenced by both the CALL and RCALL external subroutine linkages. RCALL loads general register zero from it before calling the subroutine. Both entries save the value of register zero on return in R0. Integer and logical return values can hence be recovered.

* string(8) R01

R0 and R1 can be used together as a string. R01(0|4) is R0, and R01(4|4) is R1.

* integer R1

R1, like R0, is used by both CALL and RCALL. In this case it is used to load and save general register one instead of zero.

- integer S_W

  The space field width which will follow any item, other than a string, which has been output using the Newcastle formats is in S_W.

$ reference(EXCEPTION) SINCOSERR

  Formerly this reference was used to intercept exceptions resulting from calls to the SIN, LONGSIN, COS and LONGCOS standard functions. It has now been deleted and its place is taken by a new elementary function exception reference named FUNCTION.

$ reference(EXCEPTION) SQRTERR

  Formerly this reference was used to intercept exceptions resulting from calls to the SQRT and LONGSQRT standard functions. It has now been deleted and its place is taken by a new elementary function exception reference named FUNCTION.

* integer SYSCODE

  At some point we will allow calls to the Alberta style string I/O routines (GETCARD, XGETCARD, PUTCARD and XPUTCARD) to return successfully even if a serious error occurs. When this happens, SYSCODE will contain the I/O routine return code.

* integer SYSINDEX

  This will contain the line (or index) number used by the last completed I/O operation.

* string(256) SYSPARM

  This string contains, on entry to the program, any string provided as a run-time parameter in the par field of the $RUN command or on a /EXECUTE control card. If no such string is given, it will contain all spaces.

* integer SYSTERM

  This variable controls the printing of information when a program terminates execution. The values and their actions are:

0 or less	No timing information printed
1	Total CPU time only printed

2 or more	Total, supervisor, problem and elapsed times are printed.

The conversational default is 1; in batch it is 2 .

- reference(exception) UNFL

  Floating point exponent underflow can be intercepted using UNFL. By default Algol W does not trap this exception.

* logical WRITE_CC

  This variable has been added to give a user program dynamic control over the setting of the run-time CC/NOCC switch. By default Algol W will generate carriage control characters automatically if the Newcastle output entries, WRITE, WRITEON or WRITECARD are used. Setting this switch to FALSE is equivalent to supplying PAR=NOCC in the run parameter field, and will disable this action. The Alberta style entries (PUT, PUTON, PUTCARD or XPUTCARD) never generate implicit carriage control characters.

* bits WTR

  This variable is provided so that standard procedures such as PUT which specify explicit I/O streams may refer to the current writer stream.

- integer XCPACTION

  The third field designator of the EXCEPTION pre-declared record. It specifies which option to take in the exception processing.

- integer XCPLIMIT

  The second field designator of the EXCEPTION record. It specifies the number of exceptions of its particular kind to be processed before terminating execution.

- logical XCPMARK

  The fourth field of the EXCEPTION record. If TRUE, then the message XCPMSG is printed each time an exception occurs.

- string(64) XCPMSG

The fifth field of the EXCEPTION record. The contents are printed as an error message when an exception occurs. This action is under the control of XCPMARK.

- logical XCPNOTED

  The first field of the EXCEPTION record. This flag is set to TRUE when an exception occurs.

APPENDIX 6 : Format Strings

Input/output and other data conversion operations can be performed under the control of format strings. They are used with the procedures

```
GET    (stream, format-string, input-list);
GETON

GETSTRING (string, format-string, input-list);
```

where stream is an integer or a string input device name, format-string> is a string and input-list is a list of variables whose values are to be read, and

```
PUT    (stream, format-string, output-list);
PUTON

PUTSTRING (string, format-string, output-list);
```

where stream is an integer or string output device name, and output-list is a list of expressions whose values are to be written.

The format string is a string literal or a string variable which contains up to 256 characters, and consists of one or more format specifications separated by commas (,), spaces or slashes (/). For each item in the I/O list there should be a corresponding format specification. That specification, or field descriptor, describes, on input, the kind of information in the field or, on output, the appearance of the data when it is printed.

Each field descriptor consists of a letter (I, F, E, L, A, Z, H, X, T which designates the type of information (integer, real, etc.) and a number which designates the field width. The decimal and floating point descriptors (F and E) also require, for output, the number of decimal digits that are to be printed.

Integer (I) and real (F and E) have some rules in common.

On input, a sign, if any, must be the first non-blank character; if no sign appears, the number is assumed to be positive. A completely blank field or embedded blanks will cause an error message to be produced. If no decimal point appears in F or E format, it is assumed to follow the last digit. The number can be placed anywhere within the specified field.

On output, the number appears right justified in the field. If the field is too small it is increased to the default value so that something useful can be printed.

With all format types except the tabulator, T, a repeat factor can be placed in front of the field descriptor. Several field descriptors, of all types, can be grouped within brackets and a repeat factor put before thee opening bracket. For output using the H and X formats, if, instead of an integer repeat factor, the letter "R" is used, the value of the repeat factor is taken from the next item in the data list.

There now follows a description of the different format types.

I-FORMAT

This is used to transmit integer values. The form is Iw where w is the field width. For example, if X and Y are integer variables

```
GET(5, "I3, I5", X, Y)
```

reads x from the first three columns and y from the next five.

F-FORMAT

F-format is used for real values, and takes the form Fw.d where d is the number of digits after the decimal point. The value can be input with or without an exponent: it is output in decimal format, without an exponent, unless the field width specified is too small to permit sensible printing. On input the ".d" is ignored and may be omitted.

E-FORMAT

Real values are always output with an exponent if E-format is specified, but input is exactly the same as for F-format. The form is Ew.d, where w, the field width, must be large enough to allow 4 spaces for the exponent to be printed. For example

```
X:=123.456;
PUT(8, "E9.2", X);
```

prints

```
1.23'+02
```

H-FORMAT

This is used for carriage control and for printing a line of identical characters. If encountered on input, it is skipped. Its form is Hc where c is one character. If it appears as rHc a sequence of r characters is output. For example

```
PUT(6, "H1, F5.2", X);
```

prints the value of X at the start of a new page.

X-FORMAT

The form of this format is wX. On input, w columns are skipped and on output w spaces are printed. No data is transmitted.

T-FORMAT

This is simply a tabulator, of the form Tcol, where col specifies the location of the start of the next field descriptor. If the column specified is less than the current position in the record, the field begins in that column in the next record.

LITERAL STRINGS

The format specification can include literal strings enclosed in single quotes ('). These are printed exactly as they appear.

To force the start of a new record on input or output, a slash (/) may be inserted in the format string.

If the format string runs out before the data list is exhausted, further processing of the list follows the normal Newcastle Algol W free format rules. if the data list is exhausted first, only H, X, T or literal strings are processed. All other format items are ignored.

APPENDIX 7 : Code Generated by the CALL Procedure

For: CALL ("SUBR", A, B); the code generated by Pass Three would be:

```
         LA    R3,A              Load first parameter address
         ST    R3,PARLIST        Store in first word
         LA    R3,B              Load second parameter address
         ST    R3,PARLIST+4      Store in second word
         BALR  R3,0              Save program mask
         SR    R1,R1             Zero register one
         SPM   R1                Zero program mask
         OI    PARLIST+4,X'80'   Set VL end of list bit
         LA    R1,PARLIST        Load parameter list address
         L     R15,=V(SUBR)      Load entry point address
CH1      MVI   CALLFLAG,X'02'    Set "CALL" indication
         BALR  R14,R15           Call external routine
CH2      MVI   CALLFLAG,X'00'    Reset "CALL" indication
         STM   R15,R1,RETCODE    Set R_CODE, R0 and R1
         STD   FR0,RFLOAT        Set R_FLOAT
         SPM   R3                Restore program mask
```

The two instructions marked CH1 and CH2 would not be generated if the NOCHECK option is specified for the compilation.

As can be seen from the example, the instruction overheads are 9 instructions for a parameterless call with 2 instructions for checking information. With parameters, there is one instruction to set the VL bit, and at least two instructions for each parameter. Additional instructions will be generated if the quantity is non-scalar, or if it is a reference or string field identifier.

These instructions are generated in-line for each call; CALL therefore has the status of a macro instruction in the eyes of the compiler (as are READ, WRITE etc.). We believe this instruction sequence to be the minimum safe set. It is not possible to do clever instruction saving things with parameter address calculations because this would clash with the compiler code generation optimising algorithm.

APPENDIX 8 : A Commentary by James Eve

This commentary, which was MTS Workshop paper 9L, was written by Jim Eve after seeing a draft version of this report. We have included it here for the record, as some of information contained is no longer easily obtainable elsewhere.

-----***000***-----

Some Comments on the Evolution of the Algol W Compiler currently used at Newcastle

James Eve

Two days before the beginning of the MTS workshop during a discussion of the report by Alan Hunter and Margaret Hindmarsh, Brian Randell suggested that I should try to write a semi-historical account of the evolution of the compiler. These hurriedly composed notes are the outcome. It would have been nice to have checked a few things properly and removed some unevenness in them but even so they should explain in part why some suggested changes, admittedly desirable, have either not been made or, alternatively, have been made in a way which is not the most aesthetically appealing.

It is worth re-iterating at the outset that the initial design of the Algol W language was completed as long ago as 1965 and the implementation of the compiler occupied roughly the period 1965-67. The general understanding of programming languages and compilers have improved radically since then. It is also worth stating that in 1965 the architecture of the IBM 360 and its idiosyncracies were not the familiar objects that they are now, Nonetheless Algol W and its compiler have been in use first at Stanford and subsequently at many other places for over ten years. The endurance of the Algol W system in view of the subsequent onslaughts upon it and the subsequent development of other languages and compilers is something of a tribute to Wirth's foresight and the skill of the graduate students who implemented it.

The first upheavals in the compiler occurred during its implementation when consideration was given to providing facilities for parallelism and data files; these were eventually abandoned in view of the substantial revision of already completed work which would have been needed. They were responsible for some complexity and generality in the coding which in retrospect were a disadvantage.

The original objective was to produce a "production" compiler rather than a student compiler as typified by WATFOR and the compiler was well on the way to completion before the U-turn leading to its consideration for educational use. In the event it was found that modest changes to the operating system interface sufficed to produce sufficiently high compilation speeds and low enough system overhead to meet the requirements of such environments; the basic structure of the compiler itself was unchanged. Wirth subsequently criticised the result pointing out that this change upset the balance of the design drastically. However that may be it had one very useful consequence - a single compiler which could meet the rather disparate needs of teaching and production. While not an optimising compiler in the usual global sense it does perform a number of local optimisations which made it something of a "best buy" in comparison with current Algol 60 compilers in Wichman's surveys. Some unpublished work showed that its output competed favourably with Fortran G compiled programs though the latter undoubtedly won in the case of programs making heavy use of arrays.

The compiler produced re-entrant code from the outset though not in OS object module format as at present. In view of subsequent events it is interesting that considerable effort was made to provide a good error recovery system to cope with syntax errors. This system was ad hoc and achieved at the expense of syntactically invalid "real programs"; it also depended heavily on the particular grammar then in use.

My knowledge of the compiler's history between 1968 and 1970 is sketchy. To the best of my knowledge a number of extensions and modifications which were made at that time were never completely documented. In that period Sue Graham, one of the implementors of the compiler, used it for experimental work relating to her Ph.D. thesis. She evolved a technique which improved the efficiency of the simple precedence parsing algorithm quite dramatically; overall compilation times were reduced by 10%-15%. Not surprisingly her algorithm was incorporated into the compiler then in use at Stanford. Unfortunately the syntax error recovery system was incompatible and was abandoned - effort to repeat the ad hoc reconstruction was not available. A more primitive "panic mode" recovery, which does not differ radically from that in current use, was installed.

A very much more substantial change to the compiler resulted from Ed Satterthwaite's Ph.D. work which dealt with the provision of source language debugging tools in high level languages. The Algol W compiler was the vehicle he used for practical evaluation of his ideas and the debugging system is the most obvious aspect of his work. Remarkably little of the compiler was unaffected by the incorporation of the debugging system and as a consequence he was probably the first and only person to have a detailed knowledge of the working of the whole compiler. In the course of his work the scanner was recoded completely making it more robust, the run-time support package was reorganised and a large part of it recoded, and the scanner and parser were made simultaneously core resident (both having been reduced considerably in size) improving batch monitor performance. With an eye to the future, this phase and the code generation phase of the compiler were made re-entrant and the run-time support routines became (almost) serially reusable. The compiler was embedded in OS, its output was recast in OS object module format and several performance improvements were made.

Soon after the debugging system was released Ed Satterthwaite spent two years at Newcastle which had already adopted Algol W as its main teaching language. By then he was aware of several constraints - some original design constraints, some due to implementation techniques, others arising from incompatibilities of various extensions - which were causing problems both in reliability and efficiency. At that time Tom Anderson and I were experimenting with an SLR parser embedded in the compiler. Compared with simple precedence parsers, SLR parsers impose much less stringent requirements on the form of productions defining the grammar of the underlying language. It was possible to take advantage of this to remove some of the constraints mentioned.

For a period of a year or two after this no major changes that were visible to the user were made. (The addition of formatted output was fairly trivial.) Quite a lot of work was done, however, to make the compiler more reliable. In particular Ed dealt with a number of known bugs arising from the constraints already mentioned while I coped with tracking down new ones.

By 1972-73 compiler crashes were very infrequent. The batch monitor system logged all crashes and gave partial diagnostics. Weeks would go by then there would be two or three crashes within a couple of days. It took a long time to locate the cause which turned out to be a problem common to all compilers since they all interpret the input text to a certain extents in parallel with the parsing. In the event of a syntax error, a compiler implicitly or explicitly makes changes to the text to effect a repair. This is a non-trivial task but it is insignificant compared to the difficulty of ensuring that data structures built by the interpreting routines are adjusted to be consistent with the repaired form of the text. The interpreting routines are designed to work on valid text - their effect on all possible invalid forms just cannot be comprehended at the design stage.

Inconsistencies between data structures actually built from the erroneous text and what would have been built had the repaired text been processed result subsequently in wild store operations clobbering who knows what. In the context of the batch monitor the effects were particularly unfortunate. The offending user program got its error messages and clobbered say the scanner - the next user program entered execution, the compiler crashed, the user could not fathom his output and threw the job back in, the monitor reloaded the compiler and next time his run was O.K. Then the offending user program returned without the offending error having been fixed!

Once identified it was possible to overcome this problem without too much difficulty. SLR parsers have the property of signalling syntax errors on the first invalid token. The processed text is therefore the prefix of a valid program and the data structures built by the interpreting routines are safe - if a repair can be effected without implicitly changing the processed prefix. This we failed to do but it was possible to adopt a systematic change to prefixes which in practice effectively eliminated the problem.

Soon afterwards we were able to run the compiler in MTS segment 2 so that overwriting of the first two compiler phases was not possible but the run time support library and the operating system interface remained exposed.

Afterwards at my instigation two of our graduate students, Paul Wynn and Peter White, looked at the possibility of error recovery under the constraint that no changes are to be permitted to text already processed. The results were quite promising; sufficiently good to suggest that a recovery scheme with excellent reporting capabilities could be built which would at least match anything currently available. Furthermore since the recovery was based on tables used by the SLR parser, changes to the grammar could be permitted and the error recovery scheme would automatically react. Unfortunately, for various reasons, subsequent developments of these ideas did not take place in the Algol W compiler. In principle a recovery scheme along these lines could be installed in Algol W but most users seem to want other things of Algol W than better error diagnostics and error recovery so that other tasks have taken priority.

More recent developments have been described in the report by Alan Hunter and Margaret Hindmarsh. These notes may shed some light on why some of the things are implemented the way they are. The compiler has now reached a stage where some basic design decisions are limiting factors on further development. The run time register allocation scheme is most notable among these. Attempts to circumvent Algol W's lack of variable initialisation facilities by extensive use of assignments of constants to variables is frustrated by the limit on the number of constants which can be stored in a program segment. This limit is essentially imposed by the IBM 360 addressing mechanism and can only be overcome by redesign of the register allocation scheme to free base registers. An uneasy truce exists between block expressions, the debugging system and the register allocation scheme. Removing the restriction that proper blocks can only be nested to a depth of eight is similarly constrained. Several limits of the latter type are known to more than one phase of the compiler and are used implicitly. For example, certain stacks and tables in the code generation phase would overflow if some of these limits are raised. Which limit affects which data structure is distinctly a problem. There is some evidence that changes made long ago overlooked this fact.

These are but a few examples to show that in certain areas there is little room for manoeuver. The compiler at present is quite robust but ...

APPENDIX 9 : Bibliography

Wirth, N. and Hoare, C.A.R.,
"A Contribution to the Develpment of Algol",
CACM 9 (1966) 413-431

Bauer, H., Becker, S. and Graham, S.,
"Algol W Implementation",
Stanford University Technical Report CS98, Stanford,
Calif., USA (1968)

Wirth, N.
"A Computer System for Educational Environments",
Lecture Notes, International Summer School in New Trends
in Computer Programming, København, Danmark (1968)

Wichman, B.A.
"Basic Statement Times for Algol 60",
NAC Report No.15, National Physical Laboratory,
Teddington, England (1972)

Satterthwaite, E.H.
"Source Language Debugging Tools",
Stanford University Technical Report STAN-CS-75-494,
Stanford, Calif., USA (1975)

Wynn, P.
"Error Recovery in SLR Parsers",
M.Sc. Dissertation, University of Newcastle upon Tyne,
Newcastle, England (1973)

Marsland, T.A (Editor)
"Algol W References"
Technical Report TR75-15, University of Alberta Computer
Science Dept., U. of A., Edmonton, Alta, Canada

Salisbury, R.A. (Editor)
MTS Manuals:
- 1: "The Michigan Terminal System", (1978)
- 3: "System Subroutine Descriptions", (1976)
- 5: "System Services", (1976)
- 16: "Algol W in MTS", (1978)
The University of Michigan Computer Center, Ann Arbor,
Mich., USA

Boettner, D.W and Alexander, M.T.
"The Michigan Terminal System",
Proc. I.E.E.E., 63 (1975) 912-918

APPENDIX 10 : Glossary of Terms

This Appendix provides a glossary of some of the technical terms used in this report. It is intended for readers who do not normally deal with Applications Software at the implementation level.

BLOCK DATA

A sub-program for initialising variables in named COMMON in a Fortran program.

COMMON

A storage region in a Fortran program that may be referred to by the calling program and one or more sub-programs. COMMON blocks may be named or blank (unnamed).

CONTROL SECTION

See CSECT

CSECT

A CSECT, or control section, is an object program module in System/370, containing executable instructions or data. For a CSECT to be re-entrant, this data must be constant.

DSECT

A Dummy Section of an Assembler program. These are used to address data regions. They are particularly useful to a programmer working in a re-entrant environment as they allow dynamically acquired storage areas to be addressed conveniently. The term DSECT tends to be applied loosely to the storage areas themselves as well as their structures.

DUMMY SECTION

See DSECT.

EFL routines

EFL stands for Elementary Function Library. The routines in this library provide a programmer with the commonly required mathematical functions, such as sine, cosine, etc. The set used in Release 6.0 of

Algol W originated at the University of Chicago, and were the work of Hirondo Kuki and his co-workers. These were subsequently modified at the University of Michigan for the MTS Fortran library. The changes required for Algol W use only affect the internal linkage to the error processor.

EQUIVALENCE

A method of controlling allocation of data storage in a Fortran program. In particular, it causes storage locations to be shared by two or more variables of the same or different types. Use of this facility allows a Fortran programmer to, for example, treat most of an array as having floating point numbers stored in it, but cause some elements to be interpreted as integers. Some MTS System Subroutines pass back a vector of information in which the type varies from element to element so an EQUIVALENCE facility can be invaluable.

ESD

ESD stands for External symbol Dictionary, and is usually applied to the loader card of the same name. ESD cards in an object program specify which other program modules or or system entries are required by the module that they head. They do this by specifying the External Symbol name of the modules or entries needed.

ENTRY POINT

The address of the the first machine instruction to be executed when control is passed by a calling routine. Usually there will be an External symbol name associated with an entry point. For instance, the entry point of an Algol W object program is the location of the symbol AWXSTART.

FLOATING POINT REGISTERS

These are high speed storage regions within the machine's central processor used for operations on floating point numbers. In the context of Algol W this means quantities of types REAL, COMPLEX, LONG REAL and LONG COMPLEX. the registers are numbered 0, 2, 4 and 6. Each of the four available provide 8 bytes (64 bits) of storage. However, operations on short precision quantities (REAL and COMPLEX use only the leading 32 bits of the register.

GENERAL REGISTERS

General registers, of which there are 16, numbered 0 to 15, provide high speed storage of 32 bit binary integers. Algol W uses these for operations on quantities of types INTEGER, LOGICAL, BITS and REFERENCE. Their main purpose in System/360 architecture is to hold addresses of storage areas. Machine instructions specify addresses either as the contents of a register, or as a fixed offset from the address contained in a register or registers.

IEBUPDTE

IEBUPDTE is one of IBM's OS utility programs. It contains facilities to establish and maintain the Partitioned Datasets used in OS to store macro (and other) libraries.

INC

This stands for Include, and is an MTS-specific loader card. It is used to specify the name of a file containing further Loader cards required to successfully load the object program.

LOADER CARDS

Loader cards are the output from the code generation phase of the compiler, containing a representation of the machine instructions forming the object program. In MTS the loader, or in MVS the linkage editor, reads these cards to produce an executable program. The MTS loader forms the object program directly in memory to do this. In MVS it is a two stage process. Loader cards normally come in groups of ESD/TXT/RLD/END cards called an object module. These and the other loader cards available are fully described in MTS Volume 5, "System Services".

LCS

The Low Core Symbol loader card. It is used to force a search of a low core symbol table for unresolved external symbols at load time.

LCSYMBOL

This is the main MTS user symbol table, containing the names and addresses of commonly required entries such as READ, WRITE, etc. The subroutines whose access is via this table are the main resident system routines described in MTS Volume 3, "System Subroutine Descriptions".

LDT

The Loader Terminate card. Its purpose is simple. It provides an end-of-file indication to the loader. It may optionally specify a symbol name which is to be the program entry point.

LOW CORE SYMBOL TABLE

A low core symbol table is a simple in-core data structure which is heavily used in the MTS operating system to supply tables of subroutines referenced by the loader. The format is a fullword count N of the number of entries, immediately followed by N 12-byte entries. There is one entry for each symbol defined in the table. Each entry consists of an 8-byte external symbol name followed by a 4-byte entry point address.

NAMED COMMON

See COMMON.

OS TYPE I LINKAGE

This term describes the subroutine calling convention followed in many Assembler programs, and which is, in particular, used by IBM Fortran compiled code. The linkage convention lays down which registers are to be used in a call, and the responsibilities of the calling, and called, routines. Hence:

The calling routine must:
1. Supply a 72 byte fullword aligned register save area addressed by general register 13.
2. Load the entry point address of the called routine into general register 15.
3. Load general register 14 with the address of an instruction to which control is to be returned when the called routine terminates.
4. Pass parameters, if any, via general registers 0 and 1. See later.

The called routine must:
1. On entry, save the caller's general registers in the save area provided from

offset 12 bytes. Registers 14 to 12 are saved in that order.

2. Subsequently, establish a new save area of its own such that:
   a. Word 2 (offset 4 bytes) of the new save area contains the address of the old.
   b. Word 3 (offset 8 bytes) of the old save area contains the address of the new.
   c. General register 13 contains the address of the new save area.
3. On exit, restore the caller's registers, with the possible exception of 15, 0 and 1.
4. Indicate success or failure by a return code in register 15. This is normally positive multiple of 4, with zero indicating success.

Parameters may be passed either as register values (see register call) or by the use of a parameter list (which see). Return values (also defined elsewhere) are passed by a register load. Note that parameter passing, while associated with the OS Type I convention, is not strictly part of it.

PARAMETER LIST

Parameter lists are used to pass the addresses of subroutine parameters to a called routine. In the S-type call, associated with the OS Type I linkage, the parameter list address is passed in general register one. It consists of a contiguous list of fullword addresses, one for each parameter. In the variable length convention, bit zero is set on the last parameter address, and a null parameter list is indicated by zeroing register one before the call.

PARSE

To check a string of tokens for correctness according to a given set of grammatical rules.

PASS ONE

Pass One of the compiler is the lexical scan stage. The input symbols are scanned and the program converted to a tokenised string. The compiler listing is produced as the input records are fetched. The internal tables used by the compiler are built by the scanner but not completed until later in the compilation. Where possible, program correctness is checked in pass one. However, parsing is left until later. The facility in Algol W which allows procedures to pre-reference variables in outer blocks means that checks requiring knowledge of the scope of variables cannot be made until the whole program has been read in.

PASS TWO

Pass two parses the tokenised program string, checking it for grammatical correctness. The program tokens cause the invocation of semantic routines whose output is the tree, or more usually trees, representing the program. Decisions taken at this stage include the segmentation of the object program. There is one tree for each program segment to be produced.

PASS THREE

Pass Three is the code generation stage. By a traverse of each tree produced in pass two, 370 machine instructions are generated. These are output as a standard OS object deck (see Loader Cards).

PROGRAM MASK

System 370 maintains a machine status register called the program status word (actually 8 bytes). Part of this word, one byte, is called the program mask and contains the condition code, instruction length code and four bits specifying a set of interrupts which may be masked from user programs.

PROGRAM SEGMENT

A program segment is a control section output by the compiler code generator. Program segments are generated for main programs, procedures, non-trivial blocks (those with declarations) and certain block expressions.

RECORD DESCRIPTION TABLE

This is a small control section output by the compiler with every Algol W main program containing a table of information required at run time to control storage allocation for records. Its external symbol name is AWXRCTBL.

RE-ENTRANT CODE

An object program is re-entrant if more than one task may simultaneously execute it without mutual interference. This implies that the program is pure code - that is, contains only instructions and constant data. Self-modifying code is not permissible. Data areas changed by the program must be maintained as separate copies for each task which is active. On System/370 the normal method of achieving this is to have at least one general register dedicated as a DSECT base register.

REGISTER CALL

Some simple subroutine do not use a parameter list for passing data values but instead load parameter values or addresses into general registers, usually zero and/or one. Several of the basic MTS I/O entries take their parameters in this way. (See OS Type I Linkage).

RETURN VALUES

Return values are the results of computation by a subroutine. The term is usually applied to values left in registers, in which case integer values are left in general register zero and real or long real values in floating point register zero.

RIP

This stands for Reference If Present, and is another MTS specific loader card. It is used to remove the need for forward referencing of symbols in a one-pass loader environment. Symbols which are required later in a program load can be specified on an RIP card to force loading or address resolution as they are encountered.

SAVE AREA

A 72 byte region supplied by a calling routine so that a called program may save the caller's registers. It becomes part of a doubly-linked list of areas chaining subroutine calls together. See OS Type I Linkage.

SCALAR

A simple variable occupying one cell of machine storage, as opposed to an array or other structure.

TOKENISED PROGRAM

The output from the scanner phase is a string representation of the source program as a series of tokens. Comments and blanks are removed, and all reserved words and identifiers are converted to codes held in the compiler tables. This tokenised program forms input suitable for the parser.

TRANSFER VECTOR

Because of limitations in the representation of object programs in standard object deck form, routines requiring a large repertoire of external subroutines tend to pass the addresses of groups of routines in a single array called a transfer vector. The displacement within the vector of the address of a particular routine will usually be indicated in a DSECT describing the vector.

TREE

The output from the parser semantic routines is a linearised tree. Each item in the tree representation is a operation code for an intermediate machine. These codes are used by the code generator to produce System/370 object code. In fact, for all but very simple programs, a series of trees is output, one for each program segment.

V-CONSTANT

In Assembler, references to to external routine addresses are made using constants of data type V, hence the term.

VARIABLE LENGTH CONVENTION

See Parameter List.

VECTOR

A contiguous array of scalar data cells.

WEAK EXTERNAL

Weak externals are external symbols which need not be resolved at load time. If the loader cannot find the relevant symbol, its address constant is filled in as zero. This allows a routine to test for the existence of an optional external routine before attempting to transfer control to it.