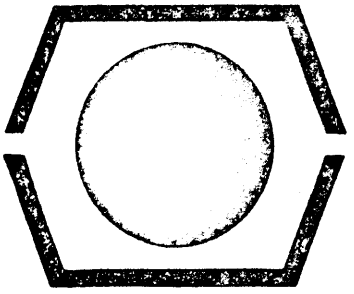


DECUS
PROGRAM LIBRARY

DECUS NO.	11-231
TITLE	ALGOL, RT11
AUTHOR	Gregory D. Hosler
COMPANY	Digital Equipment Corporation Maynard, Massachusetts
DATE	November 1975
SOURCE LANGUAGE	ASSEMBLER/ALGOL

GENERAL INFORMATION

Object Computer(s) Any PDP-11 Source Computer (if different) _____

File Name ALGOL Version No. V6.6001

Title ALGOL

Author Gregory D. Hosler

Submitter (if other than author) _____

Affiliation Digital Equipment Corporation, Software Engineer

Address 32-1 Royal Crest Drive

Marlborough, Massachusetts 01752 Country USA

Monitor/Operating System RT-11 V2B DEC No. _____

Core Storage Required 16K Starting Address _____

Peripherals Required DECtape

Other Software Required RT-11 Linker DEC or DECUS No. _____

Source Language Assembler/ALGOL Category Programming Language

Restrictions, Deficiencies, Problems Fairly bug free

Date of Planned or Possible Future Revisions _____

TAPES AVAILABLE

Paper Tapes Object Binary ☐ Object ASCII ☐ Source ☐ Other _____

DECtape ☒ LINCtape ☐ Format RT-11 Magtape: 7 Track ☐ 9 Track ☐ BPI _____

Object Files ☒ Source Files ☐ Documentation Files ☒ Other _____

ABSTRACT

RT-11 ALGOL is a compiler and run-time system for the ALGOL-60 language which operates on 16K or larger RT-11 V02B systems.

This implementation of the ALGOL-60 language features dynamic allocation of program and data segments through a software virtual memory system. All ALGOL-60 statement components are supported, plus several extensions, such as the THRU statement, numbered and unnumbered CASE statements, and the string REPLACE and SCAN statements. Data types supported are 16-bit INTEGER, 16-bit BOOLEAN, and one or two dimensional arrays (of INTEGER or BOOLEAN elements) with variable upper and lower bounds. Procedures may be typed INTEGER or BOOLEAN, or may be untyped. A Burroughs-compatible implementation of string operations (using pointer variables) is provided.

Other features include partial word operations, bit concatenation, IF and CASE expressions of all types, record-oriented random-access and stream sequential I/O. The I/O operations read and write standard RT-11 files.

The virtual memory support, RT-11 input/output, and all other operations are handled through an interpreter which executes the code files output by the ALGOL compiler.

TABLE OF CONTENTS

DECUS Program Library Write-up DECUS NO. 11-231

1. THE ALGOL RUN-TIME SYSTEM

THE SIMULATOR

This simulator was built in conjunction with the ALGOL compiler to make the job of the compiler as easy as possible while making the simulator and run-time system as efficient as possible. Because the ALGOL compiler lends itself very easily to a recursive descent parse, the code being generated is ideal for a stack oriented machine.

Therefore, this simulator is a stack machine, which means that all computations take their operands from the stack and leave their results in place of the operands. Values and addresses then are 'pushed' and 'popped' onto the stack during the evaluation of some expression so that between statements on the same level of statement nesting, the stack remains the same.

The simulator, run on a PDP-11, takes full advantage of the PDP-11's 16-bit word and stack capability.

Both data and instructions represented on the simulator are full 16-bit words. The user is referred to APPENDICES A, B, and C for the details of instruction specifications.

The code run through the simulator is similar in many respects to code run through the Burroughs B55ØØ or B67ØØ. Many operators were borrowed from these machines and some are the bases for more sophisticated operators found within this simulator. Code run through the simulator generated by the ALGOL compiler is pure, i.e. re-entrant, and can be used by more than one user at a time. However, to date, a multi-user operating system has not been developed for the PDP-11 which will allow users to share the run-time system, much less an ALGOL code file.

In general, code files run through the simulator are created by the ALGOL compiler; however, there is no reason why an assembler or a different compiler could not create similar code files. The setup for data descriptors and code files is found in APPENDICES F through K.

MEMORY MANAGEMENT

This ALGOL run-time system is setup to be dynamic.

Code segments (procedures or segmented blocks) are brought into core only as required. Array rows get allocated only as they get touched (i.e. on first access). However, dope vectors for two-dimensional arrays get allocated upon execution of the declaration. If an array is segmented, the dope vector gets created at the first access to any element but the rows remain un-allocated until some element in the particular row gets touched.

Array rows, when allocated, are assigned space in a swap file so that if the array row is to get swapped out, it has a unique place to be stored. Optimization is done when swapping data out; if an array row has not been changed since it has been swapped in, it is not written back out because there is a duplicate copy of it in the swap file.

Code segments are not written back out because it is assumed that code cannot modify itself.

The algorithm for obtaining memory space is relatively simple. A search is made through the list of memory for the first unused memory segment at least as big as needed. If there is none, then the first non-save

code segment is chosen which is at least as big as requested. If one is found then the associated program descriptor is marked non-present. If none is found, the first non-save data segment at least as big is chosen. If one is found then the associated data descriptor is marked non-present and the contents are swapped out.

If all of the above fail, memory is searched and everything non-save is swapped out. The above algorithm is applied and upon failure to find any space large enough, a 'NO MEMORY' error is put out. This means that there is not enough contiguous non-save memory to honor the request made. A stack history is dumped and the programmer can reduce the size of the code or data segment we are attempting to get space for, or reduce the size of other code or data segments or reduce the amount of saved core by removing SAVE declarations.

For a detailed description of the memory links and memory descriptors, the user is referred to APPENDIX I.

FILE HANDLING

There are five kinds of files. They are BINARY,

ASCII or DISK, KB or REMOTE or TTY, TEKT, PRINTER or LP.

PRINTER files are write-only and go out to the line printer. Default MAXRECSIZE for printer files is 40 if compiled without compiler /L option and 66 if compiled with it.

REMOTE files are read-write. Input/Output come from and go to the console. Default MAXRECSIZE is 36 (72 column). I/O transfer amount is the minimum of MAXRECSIZE, array row size, specified transfer count, and the number of characters encountered before a line terminator (input only). The following are considered to be line terminators: Line-feed, Form-feed, and Vertical-tab.

TEKT files are unformatted REMOTE files. They are treated exactly like REMOTE files except that I/O is not terminated by a line terminator.

ASCII files can be either read or written but not read/write. They are sequential. Default MAXRECSIZE is 36. Default MYUSE is Ø. Records are read from files sequentially until a line terminator is reached. These files can be contiguous but need not be. They are used

primarily for ASCII oriented I/O.

BINARY files may be read, write, or read/write. They may be read or written sequentially or randomly. Default MAXRECSIZE is 128. Default MYUSE is Ø. If an access is made randomly and then sequentially, the sequential access will start at the first record after the random access I/O. Also, if a read follows a write or a write follows a read and both are sequential, then they will access two sequential records. If an I/O requests more than the MAXRECSIZE number of words to be read/written, then several sequential records will be read/written until the request is satisfied.

If the MAXRECSIZE of a BINARY file is not 128 and the array row to be I/O'ed to is segmented and the I/O is for more than 128 words, then the results are predictable and explainable but not usually what the programmer wants. This is semi-complicated so an explanation will not be given here. The interested user is referred to the code for COMM 3 in the communicate module of the run-time system. Hence in these cases it is suggested that the programmer declares the array to be a LONG array.

To access files on a device other than the system device as BINARY or ASCII files, put the device name and a ':' preceding the file name in the TITLE part of the file declaration.

i.e. FILE DTSRC(KIND=ASCII,TITLE="DT2:SOURCE");

or REPLACE DTSRC.TITLE BY "DT2:SOURCE";

will access file 'SOURCE' on 'DT2' as an ASCII file.

FILE OPENINGS

File openings occur with the first access to the file through an I/O or by forcing it to be open (F.OPEN:=TRUE). Files are closed by a block exit in which the file was declared, closing the file via a CLOSE statement, or by forcing the file to be closed (F.OPEN:=FALSE).

When a file is opened, if MYUSE is IN and the file is not present or MYUSE is OUT and the file is present, then an error message is generated and the job is terminated. If MYUSE is I/O (BINARY only) then if the file is present it is opened for I/O. If not present it is allocated. If a BINARY file is allocated then it is created such that the number of blocks allocated will contain

MAXRECNO. The first record in the file is record number zero. If there is not enough contiguous disk to contain the record MAXRECNO, then the file will be assigned the largest contiguous area on disk. When the file is opened MAXRECNO will be calculated and filled in depending upon the size of the file and the MAXRECSIZE.

If MYUSE is Ø then if the file is present, it is opened for INPUT and if the file is non-present it is opened for OUTPUT.

HOW TO RUN ALGOL PROGRAMS

The first step is to compile the source into an ALGOL code file. This is explained in the following section on running the compiler. Once you have a code file ready to run, the rest is easy.

1. The first thing you do is to fire up the run-time system. This is done by running ALGOL5.

i.e.

$R ALGOL5

The run-time system will respond with a pound sign '#' indicating that it is ready for the name of the code

2. file to be executed. Enter the name, extension, and user code of the file. If no user code is specified then your user code is assumed to be the default. If no extension is specified then the default extension of '.ALG' is added to your file name. Therefore a null extension should be specified as '.'.

Examples

#NOEXT.

#NOEXT is same as #NOEXT.ALG

If no file name is specified then the default program to be run is the compiler (ALGOL5.ALG[1,1]).

i.e. #

results in running the compiler.

After typing the file name, the user can specify an option list to the run-time system. These options are in absolute addresses 174 and 176 (octal) and can be addressed in ALGOL programs. The setup of the option words is described in APPENDIX L.

The run-time system resides in high core occupying slightly over 8K. DOS resides in low core occupying slightly over 5K. That leaves the rest of core for the

memory of the run-time system and the non-resident drivers of DOS. DOS will bring them in when needed. Also each ASCII file requires a 1/4 K buffer in DOS. The area for these buffers and drivers is immediately above DOS. There is no way of telling DOS where you are so that he will not overwrite you. Hence it is necessary to leave room for DOS to expand.

Programs which open a lot of files on several devices may need more than the pre-allocated space. To increase the DOS filler space, at the end of the input string after the option list add in a ';n' where n is the number of 1/4 K blocks to add to the DOS expansion area.

If a program gets an F342, or a 'BAD DELETE' error message while running, first attempt to correct the problem by increasing the DOS expansion area size.

Examples

```
#TEST.35/S;5
#/S;6
#NEWTST;2
```

Only the 4 order low bits of the character after the ';' are looked at. The maximum value this could then

be would be 15 as in the letter 'O'

i.e.

#LARGE.ONE;O

In the current run-time system, an attempt to catch DOS when he overwrites the data area is done but may not always be successful.

BUILDING THE INTERPRETER AND RUN-TIME SYSTEM

There are six assembler modules required to be assembled and linked together to build the run-time system and interpreter. They and their purposes are:

AOSYS5 - The main operating system. This module contains the memory management unit and the initialization code.

AINTP5 - The main component of the interpreter. This module contains the instruction decoder and all of the operators.

ACOMM5 - All of the communication routines are located in this module.

AIOHR5 - This module contains the run-time system I/O

handler.

ADOS5 - This module is the interface between DOS and the run-time system. This module is the only system dependent module, i.e., only this module need be modified if this run-time system is to be run on other operating systems.

LIBR45/LIBR2Ø - These two modules are equivalent. They contain the non-standard arithmetic operators for PDP-11's. They are the DIV, MUL, and ASHC instructions. LIBR45 is for those with these instructions and LIBR2Ø is for those machines without these instructions.

In each of the first five assembler modules there is an assemble time variable called 'PDP11'. It is currently set to 45. Set this variable to whatever model of PDP-11 that this run-time system is to be run on. (i.e. PDP11=2Ø for running on PDP11/2Ø's).

When linking the modules together, they may be linked together in any order (only the appropriate LIBR should be linked in).

Examples:

```
$R LINK
#ALGOL5,ALGOL5<AOSYS5,AINTP5,ACOMM5,LIBR45,ADOS5/E
```

for those machines with hardware MUL/DIV instructions

or

```
$R LINK
#ALGOL5,ALGOL5<AOSYS5,AINTP5,ACOMM5,LIBR2Ø,ADOS5/E
```

for those machines without them.

There will be one undefined symbol in AINTP5 and ACOMM5 (.COM14). This can be ignored.

The run-time system uses the space between itself and DOS for swapping. Therefore it is suggested that the run-time system be linked as high in core as possible in order to make this swappable space as large as possible.

NOTE: The initialization code is non-reentrant. Not only that, in order to get as much swap space as possible, the initialization code is also used for swapping, i.e., it gets overwritten with data and code. Hence it is the case that a CONTROL C, BE and a CONTROL C, RE will not work and will cause unpredictable results (probably an F342 or F344). Therefore a CONTROL C, KI is suggested and

then run the run-time system again.

2. THE ALGOL COMPILER

COMPILER OVERVIEW

The ALGOL compiler implemented for this project is a recursive descent compiler generating polish post-fix notational code. The code generated is stored in a file which is then run through the ALGOL run-time system and simulator described in Part 1. An attempt was made to make this compiler as compatible as possible with the EXTENDED ALGOL compiler on the Burroughs B67ØØ. In many respects this goal was accomplished. However, there were some hardware differences which made implementation of certain of the data types slightly different.

The first major difference is that the B67ØØ has a 51-bit word (48 for data and 3 for tags). The PDP-11 has a 16-bit word of which I decided to use all 16 bits for data. Hence there are no tags in the PDP-11 simulator. The second major difference is the fact that the B67ØØ handles arithmetic in signed magnitude notation whereas

the PDP-11 handles arithmetic in two's complement notation. This only affects partial word and concat operations on negative numbers.

Other hardware differences follow from these two, i.e., the largest number representable on the PDP-11 is $2^{15}-1$ and the smallest number representable is -2^{15} (follows from the word size difference).

Taking these differences into consideration, a substantial number of non-trivial programs can be run on both machines yielding the same output for the same input.

To implement this ALGOL compiler put the files ALGOL5.ALG and ALGOL5.ERR under [1,1] as contiguous files. The first file is the compiler and the second is its error message file.

COMPILER SOURCE INPUT

The following data structures have been implemented:

INTEGER - 16 bit two's complement

BOOLEAN - 16 bit with bit #Ø as TRUE/FALSE bit

ARRAY - one or two dimensional typed INTEGER or BOOLEAN

POINTER - 8 bit pointers only

TRUTHSET - can represent numbers from Ø to 127 as well as any 8-bit character

FILE - record oriented array row read/write

The following control structures have been implemented:

LABELS, SWITCHES, and GO TO's

CASE and numbered CASE statements

DO-UNTIL, WHILE-DO, FOR-DO, THRU loops

PROCEDUREs - untyped or typed BOOLEAN or INTEGER

Assignment statement - may be of type INTEGER, BOOLEAN, or POINTER

REPLACE, SCAN, FILL, CLOSE, I/O, SWAP, BLOCK, COMPOUND, LOCK, and RELEASE statements with all but the last two found in EXTENDED ALGOL.

DEFINEs and DEFINEs with up to 1Ø parameters have been implemented.

CASE expressions and conditional expressions have been implemented for arithmetic, pointer, and designational expressions.

See APPENDIX N and APPENDIX O for ALGOL Reserved words and ALGOL built-in functions.

Syntax and implementation of ALGOL was kept as similar as possible to that of the B67ØØ/EXTENDED ALGOL.

The user is referenced to a B67ØØ EXTENDED ALGOL LANGUAGE REFERENCE MANUAL (FORM NO. 5ØØØ649, 5-2Ø-74) available from Burroughs Corp. for a price of seven dollars.

Differences, limitations, and extensions are listed below:

ARRAY

One dimensional segmented arrays have a maximum length of 16383 entries (half that for real arrays). One dimensional LONG arrays have a maximum row size of 8191.

Two dimensional arrays may not have a dope vector of size greater than 8191 with rows (which are LONG) of size no bigger than 8191.

CASE statement

The maximum number of cases allowed in a case statement is 1Ø1 (Ø through 1ØØ). This is a restriction put on by the compiler and can be changed by changing CASESTMT within the compiler.

Extensions to numbered case statements:

1) more than one case label may appear on a statement.

2) if cases n through m are to execute a labeled statement where n is greater than m then a label of the form n-m is permitted.

EXAMPLE

```
CASE I OF
BEGIN
     3: J:=1;
        L:=2;
     4: 2: 8: J:=I;
     5-7: J:=L*I;
END;
```

CLOSE statement

Three forms of the CLOSE statement were implemented. They are:

A) CLOSE(<fileid>);

B) CLOSE(<fileid>,PURGE);

C) CLOSE(<fileid>,<bexp>);

Type A will close the file and lock it (leave it

on the device).

Type B will close the file and then delete it (remove it from the device).

Type C will close the file. If the <bexp> is true (bit #Ø on) then the file will be removed (purged).

DECLARATIONS of INTEGERS, BOOLEANS, and POINTERS

Identical to B67ØØ EXTENDED ALGOL with the following extension to each of these three types of Declarations. You may initialize local pointers via an assignment expression.

EXAMPLES:

```
INTEGER     I,J:=5,K:=REAL(B1)*2+3,L;
BOOLEAN     B2:=TRUE,B3,B4:=B OR C;
POINTER     P1,Q:=P,R:=Q+(2*3),S;
```

DEFINEs

Defines without parameters are handled exactly the same as on the B67ØØ. Defines with parameters may have up to ten parameters. However, the invocation must be with parenthesis and cannot be

with brackets as allowed on the B67ØØ.

DO-UNTIL statement

The DO-UNTIL statement is handled exactly as on the B67ØØ.

EXPRESSIONS, ARITHMETIC

The operators MUX and TIMES were not implemented. The operators @ and @# were added which do a load and load-byte operation respectively. They operate on primaries.

EXAMPLE:

@174 is the contents of absolute location 174 (decimal).

@#176+2 loads the contents of byte 176 absolute and then adds 2 to that result.

Other than these differences and those mentioned under machine differences on page 7 arithmetic expressions are handled exactly the same as on the B67ØØ.

EXPRESSIONS, BOOLEAN

The operators IMP, EQV, IS, ISNT were not

implemented. Otherwise boolean expressions are exactly as in EXTENDED ALGOL.

EXPRESSIONS, POINTER

Syntatically, pointer expressions are parsed exactly the same as in EXTENDED ALGOL. Semantically there are a few differences. A pointer expression with a <skip part> on the B67ØØ is checked for segmented array in the expression evaluation. The PDP-11 run-time system only checks for segmented array when a pointer is used in a character scanning or moving operation. The second difference is that the <skip part> and not its absolute value is added.

i.e. P+<prim> results in P-ABS(<prim>) if <prim> is negative.

Other than that, pointer expressions are evaluated the same.

FILL statement

The following construct of the <initial value> part of the <value list> of the fill statement

was not implemented:

<unsigned integer> (<value list>)

Each fill statement generates a new data segment which is put into the code file to fill the array row with.

FOR-DO statement

The number of <for list elements> to a FOR loop varies between 3Ø and 1ØØ depending upon the type of <for list element>s used. An invalid index in compiler segment #Ø115 indicates that you have exceeded this limit.

Expansions:

Besides the <for list element>s implemented in EXTENDED ALGOL, the <for list element> <ae1> UNTIL <ae2> has been added which compiles as <ae1> STEP 1 UNTIL <ae2>.

FOR loops have been implemented exactly as on the B67ØØ.

GO TOs

Syntax is exactly the same, i.e., you may go to a

designational expression.

Limitation:

If you are exitting a procedure via a GO TO, then the block that you end up in must either be the main program block or the main block for some procedure.

I/O statements

Only array row I/O is implemented. The format part must be an expression which is taken to be the number of words to do I/O to. The list part must be an array row designator. The file part may have a record part appended to it.

On BINARY files this record part indicates the record number of the record to be read/written. (Zero is the first record of the file.) If the record number evaluates to a negative number or if the record part is omitted in the I/O statement, then the next record is assumed to be the destination of the I/O.

On ASCII files output the record number indicates,

if greater than zero, the number of blank lines to precede the line to be written. If the record number is zero on ASCII output, then it indicates that the record is to be written without a carriage return or line feed after it. This is equivalent to a WRITE-STOP in EXTENDED ALGOL. On input from ASCII files, the record part is ignored. REMOTE, and PRINTER files are treated as ASCII files.

EXAMPLES:

```
READ(CODE[5],128,CODEARRAY);
WRITE(CODEFILE[I*J+2],J*128,B[*]);
WRITE(LINE[5],66,HEADING);
WRITE(KB[Ø],1Ø,ASKFORINPUT);
READ(BINARYFILE,512,A[I,*]);
```

PROCEDUREs

Limitations:

Parameters are call by name unless specified to be call by value. Call by name parameters may not be expressions. ARRAYS, TRUTHSETS, FILES must be call by name. INTEGERS, BOOLEANS, and POINTERS may be either call by name or call by

value. A subscripted array element is considered to be an expression. Typed and untyped procedures of zero parameters may be passed by name. LABELS and SWITCHES may not be passed as parameters.

The number of parameters to a procedure should not exceed 145. The compiler may act unpredictably should this happen.

REMARKs

There are three types of remarks: the end-remark, the comment-remark, and the escape-remark. (2-7 in Manual). This compiler's end-remark is terminated only by END, UNTIL, ELSE, or ';'. All other characters will be ignored (except the '.' after the final END).

REPLACE statement

Extensions:

If the count part is the constant 1 in a word or convert transfer, then the class 2 reserved words WORD, DIGIT may be used in place of the reserved words WORDS, DIGITS.

The convert part has been greatly enhanced. You may specify the base of conversion as well as whether or not to zerosuppress it. Default base of conversion is DECIMAL. The default is not to zerosuppress. Base of conversion may be BINARY, OCTAL, DECIMAL or HEX.

EXAMPLES:

```
REPLACE Q:P+1 BY I FOR 3 BINARY DIGITS, J
     FOR 1 HEX DIGIT, K FOR 5 ZEROSUPPRESSED
     DIGITS;
```

Ambiguities:

It would be difficult to parse these properly.

1) An indexed array element may be a pointer expression, i.e., P:=A[5]+3 generates a pointer pointing to the third character after A[5]. But A[5]+3 is also an arithmetic expression and is treated as such in the replace part. A[*] will result in 'PRIMARY MAY NOT BEGIN WITH THIS TYPE QUANTITY' on the '*'. The solution here is to use POINTER(A[*]) and above use POINTER(A[5])+3.

2) FILEID.MAXRECSIZE in the replace part

will result in the error message '.TITLE' EXPECTED on the 'MAXRECSIZE'. The solution here is to put the entity inside of parens, i.e., (FILEID.MAXRECSIZE).

The same is true for the other arithmetic file attributes.

EXAMPLE:

```
REPLACE P BY (FILEID.MAXRECSIZE) for 5 ZERO-
    SUPPRESSED DIGITS;
```

Differences:

The scanning and character moving operators move in the direction of increasing array indexes. However, within each word the low byte is scanned first and then the high byte. The same is true for a replace operation. On the B67ØØ the direction of the scan is from high byte to low. All string constants are wrapped from the last character to the first.

SCAN statement

Differences:

Same differences as noted in the REPLACE statement.

SWAP statement

Arrays need only agree in the number of dimensions.

SWITCHES

Limitations:

A switch cannot be used outside of the block in which it was declared. The maximum number of indicies to a switch is 251. An attempt to declare a SWITCH with more than 251 indexes will result in an invalid index inside of SWITCHDEC. This can be fixed by changing the upper limit on the number of indicies in SWITCHDEC. Switches may not reference another switch.

EXAMPLE:

```
SWITCH SW:=L1,L2,L3, IF B THEN L4 ELSE L5;
SWITCH S2:=CASE I OF (L1,L2,SW[J],L2);
```

The above example will result in a syntax error on the use of SW within S2.

THRU statement

Limitations:

If the number of times that a statement is to be THRUed is evaluated to be greater than $2^{15}-1$ then the statement will not be executed. (Numbers greater than $2^{15}-1$ are negative on the PDP-11.)

TRUTHSETS

The capability to use a subscripted variable and the capability to address the bits of the truthset were not implemented.

The following TRUTHSETS have been pre-defined for the user.

ALPHAONLY - A-Z upper case and lower case.

NUMERIC - Ø - 9

ALPHA, ALPHANUMERIC - ALPHA or NUMERIC

SPECIAL - printable characters not in ALPHANUMERIC

WHILE-DO statement

Exactly the same as on the B67ØØ.

The length of an identifier is limited to 63 characters as in EXTENDED ALGOL. Characters after the first 63 will be treated as comment. If a line is greater than 72 characters only the first 72 will get read and the rest will be ignored.

If a line ends with a reserved word or identifier immediately preceding the line terminator and the following line begins with an identifier or a reserved word in column 1 and the compiler is being run with the /S option to the run-time system, then the two will be concatted together to form a single identifier. This is due to the fact that the /S option takes out the trailing spaces.

Alternately, if an identifier is split in two on two consecutive lines and the last character on the first line was not in column 72 and the /S option was not used, then the separate parts of the identifier will be treated as such.

A good programming practice would be to not split identifiers up onto several lines and to have at least one leading space on lines other than the first. (This would be the case anyway if you were blocking your program!)

HOW TO RUN THE COMPILER

The first thing to do to run the compiler is to run the operating system and simulator. Do this by entering $R ALGOL5.

It will reply with a '#'. Your reply to this will be the compiler name followed by the switches to the run-time system followed by the DOS expansion factor (only necessary if you have a DEC-TAPE file).

EXAMPLES:

```
#/S
#ALGOL5.ALG/S
#
#/S;6
#;7
```

The compiler, when run properly will reply

```
ALGOL VØ5.6.xxx
#
```

where xxx is the patch update number. Each time the compiler is patched and re-booted, this number gets incremented. Your reply to the '#' is of the form

[<code file>][,<list file>]<<source file>

Files in brackets are optional and may be omitted. If extensions for any of the files are left out then default extensions are used. The default extension for the code file is '.ALG'. The default extension for the listing file is '.LST'. If the source file has no extension, first the file with the extension of '.SRC' is looked for. If this is not found then the file with the null extension is looked for. If this is not found then an error message is generated.

Options may appear after any file id. Only the first character of the option is looked at; the rest is ignored. Options may appear in any order.

EXAMPLES:

```
#FOO/C,FOO/LONG/PRT/DEBUG<FOO
#,LP:<TEST1.NEW/L
#CODE.2/C<CODE.1
```

See APPENDIX P for an explanation of the compiler switches.

THE BOOTING PROCESS or WHERE DID THE FIRST COMPILER COME FROM?

If the compiler is itself written in the language it compiles and there is no other ALGOL compiler on this PDP-11, where did the first ALGOL code file of the compiler come from?

To the answer of this question I have only limited knowledge. The original version of this compiler was an ALGOL-60 compiler with strings. It was booted from a Burroughs B5500 to a PDP-8 which explains why the code that it generates is so much like the code found on the B5500. It was later booted from a PDP-8 to a DATA GENERAL NOVA via BARRY JAMES FOLSOM and RICK SHAW. I received from Rick, copies of the ALGOL-60 compiler code file and compiler source (written in itself), and copies of the NOVA assembler run-time system.

Using the NOVA run-time sources as a guideline, I developed a run-time system for the PDP-11. I then had to modify the code file because the compiler does some 'hard' addressing which was machine dependent upon the NOVA. I then modified the compiler source to make it

dependent upon PDP-11 addressing rather than the NOVA. After many trials and tribulations that booted onto the PDP-11 and became version 6.4.

I then added all the constructs of EXTENDED ALGOL and blocked and commented the compiler (which became a non-trivial task). Then I had to rewrite the compiler in itself. Next I rewrote the run-time system to provide better error diagnostics as well as handle the new operators necessary to boot up the current version of the compiler (version 6.5).

All in all I'm still surprised that it worked.

POSSIBLE ENHANCEMENTS

It would be nice if REAL arithmetic were added to the compiler. Real arrays have already been implemented in the simulator and run-time system. Equally as nice would be the addition of FORMATS, LISTS, and formatted I/O. This I think will end up being a non-trivial task. My suggestion would be to write the formatter in ALGOL and B55ØØ simulator-assembler then bind that code into code files which use formats. For this task though, a binder

would have to be written. A RESIZE statement probably is the most feasible enhancement. Monitor statements would help in debugging, as would an XREF and an XREF analyzer. Another welcome enhancement would be to make ASCII files character-record oriented as well as the current word-record orientation. This would require the file attribute UNITS to be added (i.e., UNITS=CHARACTERS or UNITS=WORDS). This will require the use of another bit in the file descriptor. Possibly you could use the upper byte of the word which now contains MYUSE.

A little bit more challenging would be an ON statement to catch things such as:

```
ON INVALIDINDEX
ON ZERODIVIDE
ON ENDFILE(<fileid>)
ON INTEGEROVERFLOW
ON CONTROLC
```

I'm not quite sure how something as this would be implemented, but if somebody ever figures out call by reference, this should follow without too much difficulty.

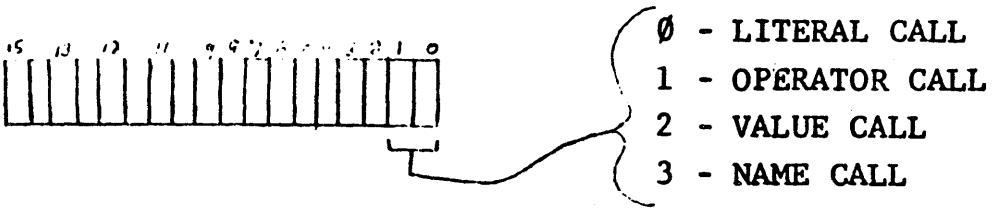
Those, I think, are the most feasible. Something to consider is that the addition of more PDP-11 assembler

code to the run-time system results in less space for the compiler to reside in, i.e., the space left for swapping will be decreased by whatever the run-time system is increased by. Hence it may be somewhat plausible to write REAL arithmetic in ALGOL routines and use the simulator for conversions and storage operators. This may be slow and non-desirable for number crunching though.

APPENDIX A

SIMULATOR DECODING

Each instruction syllable of the simulator can be of four types depending upon the value of the low order two bits of the instruction. (i.e. bits #Ø,1.) Each syllable is decoded as follows:



A literal call (LITC) takes as its value bits 15 through 2 and pushes this value onto the stack. Note that this value has a range from Ø to 37777 octal (16383 decimal) because bits 16 and 15 will always be zero. An operator call does just that. Depending upon the value of bits 15-2, a specified operator is called upon to perform operations using the information in the stack. These operators and a brief description of their job is found in APPENDIX B.

The value call (VALC) and name call (NAMC) operations use bits 15-2 to compute an address either within

the stack or in the Program Reference Table (PRT). The PRT contains all OWN, global (level 1), and EXTERNAL data descriptors as well as all program descriptors. The NAMC operator pushes the computed address onto the stack, the VALC operator pushes the contents of the computed address onto the stack. The algorithm for the address computation is found in APPENDIX C.

APPENDIX B

SIMULATOR OPCODES

OPCODE*	MNEMONIC	DESCRIPTION**
24Ø1	IADD	INTEGER ADD
24Ø5	AOC	ARRAY OPERAND CALL
2411	ASD	ARRAY STORE DESTRUCT
2415	ASN	ARRAY STORE NON-DESTRUCT
2421	BRUN	BRANCH UNCONDITIONAL
2425	BRTR	BRANCH TRUE CONDITION
2431	BRFL	BRANCH FALSE CONDITION
2435	ENTR	ENTER SEGMENTED BLOCK
2441	CHS	CHANGE SIGN
2445	COMM	CALL SYSTEM COMMUNICATE
2451	DEL	DELETE TOP OF STACK
2455	DIVR	INTEGER DIVIDE (ROUNDED)
2461	DUP	DUPLICATE TOP OF STACK
2465	NEQ	NOT EQUAL COMPARE
2471	EQL	EQUAL COMPARE
2475	GEQ	GREATER THAN OR EQUAL COMPARE
25Ø1	LSS	LESS THAN COMPARE
25Ø5	GTR	GREATER THAN COMPARE
2511	LEQ	LESS THAN OR EQUAL COMPARE
2515	LOAD	LOAD VALUE WHOSE ADDRESS IS ON TOS
2521	LOR	LOGICAL OR
2525	LAND	LOGICAL AND
2531	MKS	MARK THE STACK
2535	REP	REPLACE OPERATOR
2541	IMUL	INTEGER MULTIPLY
2545	LNG	LOGICAL NEGATE
2551	REL	RELEASE (UN-LOCK) STORAGE
2555	RTN	RETURN (BLOCK EXIT)
2561	SAV	SAVE (LOCK) STORAGE
2565	SBR	PROCEDURE ENTER
2571	SHL	SHIFT LEFT

* In octal.

** For a more detailed description the user should see the Assembler listing of the module AINTP5.

OPCODE	MNEMONIC	DESCRIPTION
2575	SHR	SHIFT RIGHT
26Ø1	STOD	STORE DESTRUCT
26Ø5	STON	STORE NON-DESTRUCT
2611	ISUB	INTEGER SUBTRACT
2615	XCH	EXCHANGE TOP TWO STACK CELLS
2621	SCAN	SCAN OPERATOR
2625	IMOD	INTEGER MOD
2631	ADC	ARRAY DESCRIPTOR CALL
2635	FDI	FIELD ISOLATE
2641	BPS	BUMP STACK POINTER
2645	SWAP	SWAP ARRAYS
2651	IEXP	INTEGER EXPONENTIATE
2655	FID	FIELD ISOLATE DYNAMIC
2661	RSDN	ROTATE STACK DOWN
2665	RSUP	ROTATE STACK UP
2671	INOP	TRUTHSET 'IN' TEST
2675	OCX	OCCURS INDEX
27Ø1	LODB	LOAD BYTE
27Ø5	DIVT	INTEGER DIVIDE (TRUNCATE)
2711	FIS	FIELD INSERT
2715	FISD	FIELD INSERT DYNAMIC
2721	FIND	LOCATE ADDRESS
2725	ONES	COUNT NUMBER OF BITS THAT ARE ON
2731	FONE	FIRST ONE (LEFTMOST ONE)
2735	B1D	BUILD 1-DIMENSIONAL ARRAY DESCRIPTOR
2741	B2D	BUILD 2-DIMENSIONAL ARRAY DESCRIPTOR
2745	DUPL	DUPLICATE & LOAD INDEXED ARRAY VALUE
2751	BLD	BUILD ARRAY DESCRIPTOR
2755	PLOD	LOAD WHAT POINTER POINTS AT
2761	PART	LOAD PARTIAL WORD VALUE
2765	PSTN	POINTER STORE NON-DESTRUCT
2771	PSTD	POINTER STORE DESTRUCT
2775	SCMP	STRING COMPARE
3ØØ1	PLNK	LINK POINTER

APPENDIX C

SIMULATOR ADDRESSING

There are four types of relative addressing used by the simulator depending upon the location of desired information. They are PRT-PLUS (R+) for items in the PRT, PROGRAM COUNTER PLUS (C+) for entries within the current program segment (in floating data pools located beyond the current PC), FRAME-PLUS (F+) for local entities of procedures, and FRAME-MINUS (F-) for parameters to procedures.

Hence there are three basic registers, strangely enough, called the R, C, and F registers. The R register always points to the base of the PRT. The C register always points to the next instruction to be executed, and the F register points to the current Return Control Word (RCW). See APPENDIX D for the setup of an RCW. Addresses are computed as follows:

P+ : Bit 15 of instruction syllable equals Ø, bits 14-2 is word index into the PRT.

C+ : Bit 14 of instruction syllable equals Ø, (bit 15 = 1), bits 13-2 is word index from next instruction to be executed.

F+ : Bit 13 of instruction syllable equals Ø, (bits 15,14=1), bits 12-2 is word index into local stack space (subtract times 2 from F).

F- : Bits 15,14,13 = 1, bits 12-2 is word index into parameter space (add times 2 to F).

APPENDIX D

RCWs and MKSCWs

Return Control Words (RCWs) are used to remember where we go after we finish a procedure call, i.e. the return address. Mark Stack Control Words (MKSCWs) are used to remember where the stack was before the procedure was entered. This is used in a block exit to 'cut back' the stack.

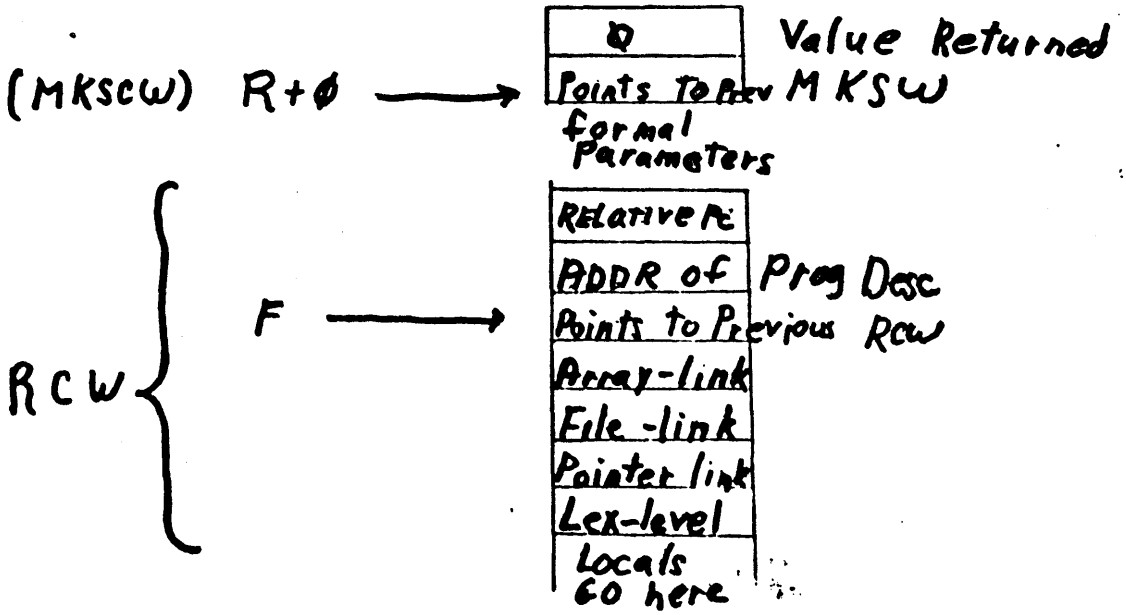
The stack is marked with a MKS operator before pushing on the parameters and after the stack cell for a typed procedure (to pass through the answer) has been allocated. This is done with a level Ø MKSCW. If we are to enter a segmented block (not a procedure, i.e. IF be THEN BEGIN INTEGER I; ...) and that block is declared at lex-level (not to be confused with the begin-end level) N, then the stack is marked with a MKSCW of level N. The MKSCW of level Ø is located in the PRT as its first entry, the MKSCWs of other levels are allocated dynamically into the PRT as needed.

A MKSCW always points to the current mark stack word (MKSW) of its level. Each MKSW points to the previous MKSW

of its level. In this fashion the stack is linked.

RCWs work in a similar manner. The F register points to the most recent RCW. Each RCW points to the next most recent RCW. RCWs are created mainly by the SBR instruction but can be created (or fuged up) by a COMM-9 call for a segmented block exit.

An RCW is composed of three major parts. One part tells which block and where in that block this procedure was called from. A second part describes where the previous RCW is, and the third part tells which, if any, files, arrays and pointers were declared locally to the procedure being exitted. Files must be closed, arrays de-allocated, and pointers un-linked. This is done by the RTN operator. A typical RCW is outlined on this page.



One block exit cleans the stack such that the value being returned (under the MKSW) is left on top of stack.

For exitting a nested block or procedure via a GO TO or by reaching the end of a segmented block and 'fall out', COMM-9 is used as follows:

If we are exitting to a procedure block and crossing through at least one other procedure boundary, then we fake up the RCW to the procedure that we are go to-ing to and change the return address (relative PC) in the RCW and make the F register point to that RCW. Then we do a RTN which does a block exit after fixing up the MKSCW for level Ø. This will clean up the environment for us.

In all other cases we are not crossing a procedure boundary. We build a RCW on the top of stack with a return address (relative PC) and address of program descriptor we wish to go to. Then we fake up all the links so that when we do a RTN our environment will get cleaned up for us. Then we do our RTN which will pull us into the proper block and clean the stack.

APPENDIX E

ARRAY LINKS, POINTER LINKS, & FILE LINKS

The data structures ARRAY, POINTER, and FILE are linked up in the order of their respective declarations for the following reasons:

Files declared locally to blocks must be closed when the block is exitted. Arrays declared locally to blocks must be de-allocated and their overlay cells freed for other array rows. Pointers are linked up to prevent the infamous up-level attach where a global pointer is attached to a local array. The block is then exitted and the array descriptor, being built in the stack, gets over-written and is no longer an array descriptor, yet is treated as such when the pointer is accessed. Well, we managed to get around that problem!

There are three linked lists in the ALGOL run-time system; one for files, one for arrays, and one for pointers. Each list has two entry points; one for pre-linking, and one for post-linking. Globals, in the outermost block, and own declarations are pre-linked in their appropriate list the first time that their declaration is encountered.

Thereafter they are not linked. Locals are post-linked into their appropriate list each time their declarations are encountered. That is because they get unlinked by the block exit routine.

When we do a block enter via a SBR, we save the current post-links so that when we do a block exit, we can 'un-declare' the declarations for that particular block. The links work as follows:

FILES -

Each file link points to the file descriptor of the file declared after it. This is initialized to Ø for ground until the file after it is declared. The file-link is the fourth word of the file descriptor (see APPENDIX F).

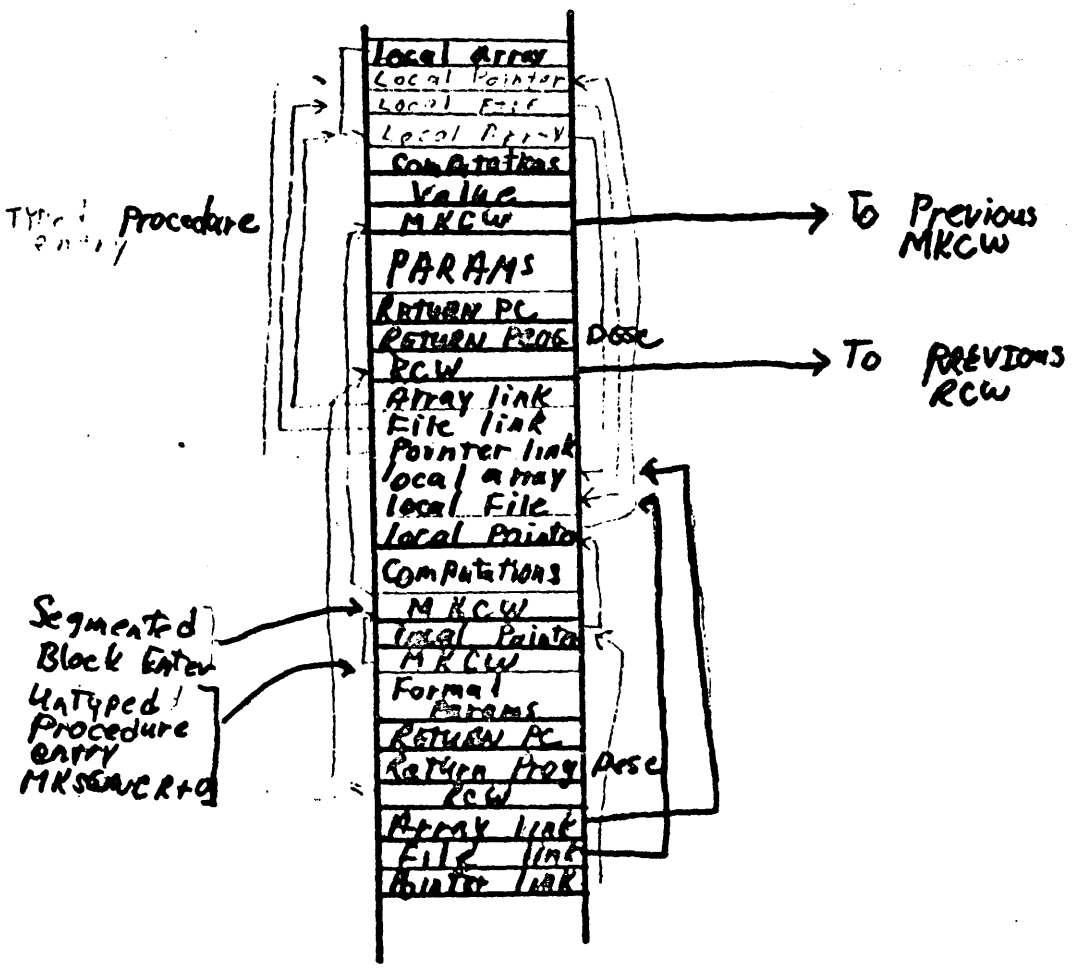
ARRAYS -

Each array link points to the array link of the array-link of the array declared after it. This also is initialized to Ø until the array declared after the current declaration gets declared. The array link is immediately before the array descriptor (see APPENDIX G).

POINTERS -

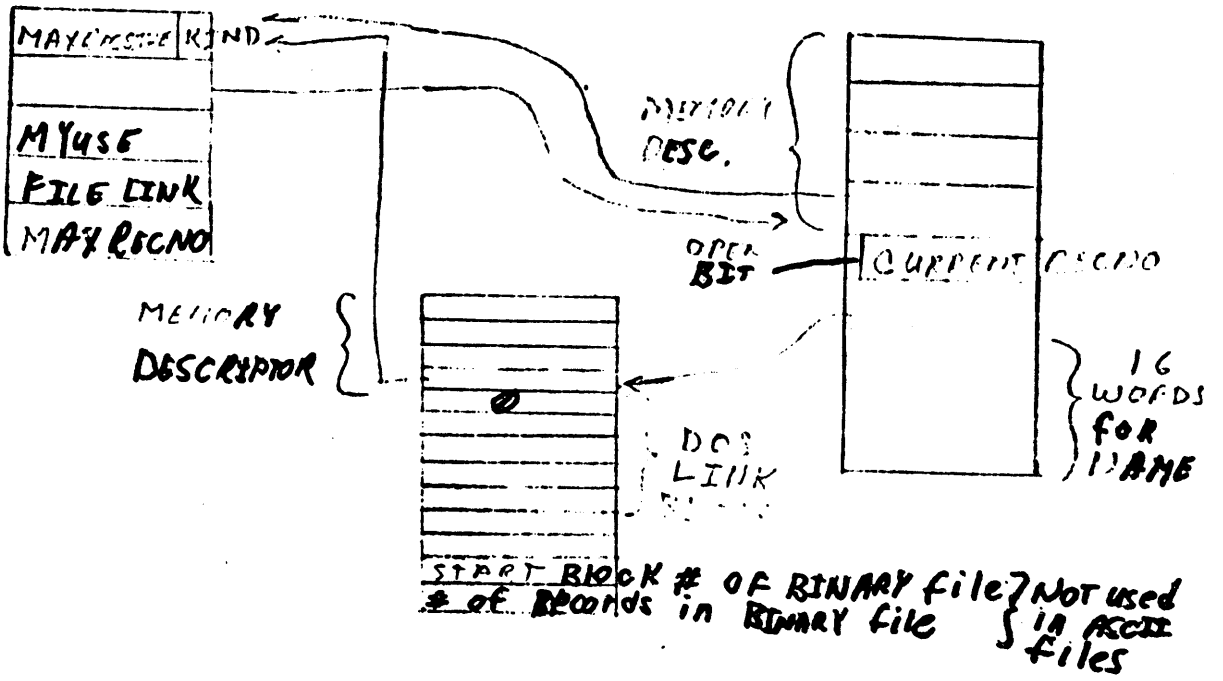
Each pointer link points to the pointer link of the previously declared pointer. The pointer in the front of the list points to ground (Ø). The pointer-link is the third word of a pointer descriptor (see APPENDIX I).

A stack configuration then could look like the following with local descriptors built in the stack and linked RCWs and MSCWs from procedure entrances:



APPENDIX F

FILE DESCRIPTORS



MAXRECSIZE - bits 15-3

KIND - bits 2-Ø

value	file-kind
Ø	BINARY
1	ASCII
2	REMOTE
3	unformatted REMOTE
4	PRINTER
5,6,7	ILLEGAL DEVICE (7 is default KIND)

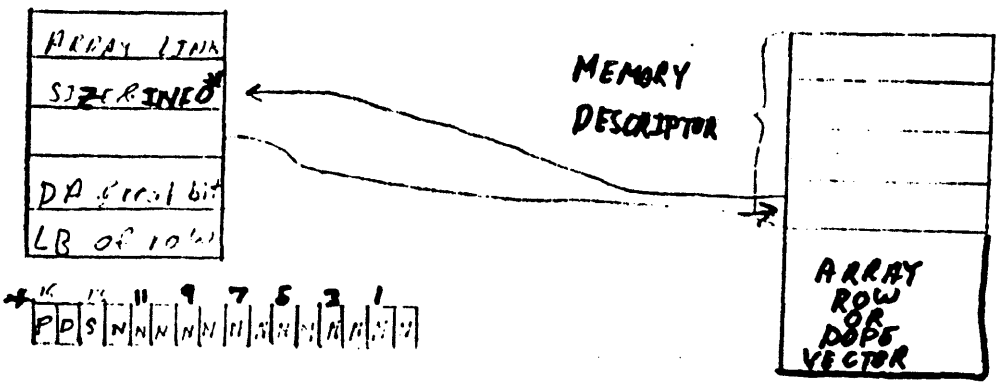
OPEN BIT - bit 15

MYUSE - bit Ø = IN, bit 1 = OUT

CURRENTRECNO - (BINARY files only) record number to do next sequential I/O to (first record of file is record number Ø)

APPENDIX G

ARRAY DESCRIPTORS



LB - LOWER BOUND

P - PRESENCE BIT (bit 15)

D - TWO-DIMENSIONAL BIT (bit 14)

S - SEGMENTED BIT (bit 15)

N - NUMBER OF ENTRIES IN ROW OR DOPE VECTOR (bits 12-Ø)

The pointer word, word 2 (word 1 being the size and info word), contains the address of the word immediately preceding the first entry of the array row or dope vector if the present bit is on. If either of bits 13 or 14 are on (equal 1) then the word pointed to by word 2 is a dope vector. Each entry in the dope vector is a four word entry identical to an array descriptor without the array-link. If the dope vector is of the segmented type (bit 14 on) then the lower bound entry in each of the dope vector

entries is set to Ø. Otherwise the lower bound entry in each of the dope vector entries is set to the lower bound of the second dimension (array rows). Note that bits 13 and 14 cannot ever both be on (dope vectors never get segmented). If bit 14 (D) is on then the LB entry in the array descriptor is the lower bound of the first dimension (the dope vector dimension). In all other cases the lower bound entry is the lower bound of the array row. Array rows of two-dimensional arrays never get segmented. If a one-dimensional array is declared long, it also will never be segmented. If a one-dimensional array is not declared long, then if the size of the array row is less than 257 words, it also will not be segmented.

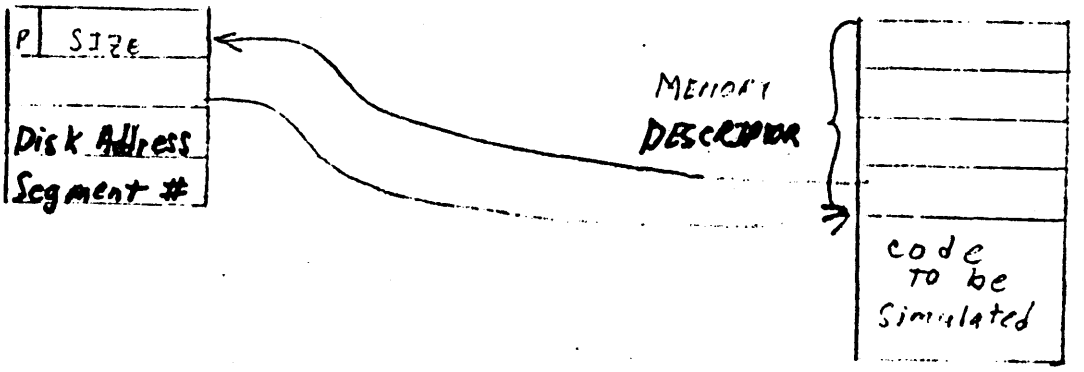
Word 1 of the array descriptor reflects the number of entries in the row pointed to by word 2 not the number of words in that row. If the row is a row of REAL entries, each entry is two words long. The indexing algorithm takes this into consideration when it indexes into the array row.

The third word of the array descriptor is the disk address of where the array row can be found if it is non-present but allocated. This entry is initialized to zero for boolean and integer arrays and to minus one for real

arrays. If this entry is its initialized value when an indexing is attempted then an array row is created and assigned unique swap space in the swap file. The record number of this space is put in the third word of the array descriptor in bits 13-Ø. If the initial entry in this word was negative then bit 14 of the disk address records this by remaining on; otherwise it is off. The MAXRECSIZE of the swap file is 128, the same as the segment size of segmented arrays.

APPENDIX H

PROGRAM DESCRIPTORS



P - PRESENCE BIT (bit 15)

SIZE - NUMBER OF WORDS IN SEGMENT

DISK ADDRESS - RECORD NUMBER INTO CODE FILE OF WHERE THE CODE SEGMENT MAY BE FOUND IF IT MUST BE SWAPPED IN. SET BY THE COMPILER.

SEGMENT NUMBER - SEGMENT NUMBER OF COMPILED SOURCE. SET BY COMPILER. USED IN DUMPING STACK HISTORIES.

APPENDIX I

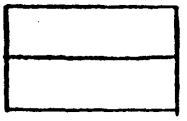
OTHER DESCRIPTORS

INTEGER, BOOLEAN DESCRIPTORS

1-word descriptor contains a 16-bit value

All 16 bits are used for data. For booleans, bit Ø is the TRUE/FALSE bit with 1 as TRUE and Ø as FALSE.

REAL DESCRIPTORS

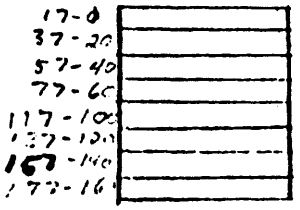


2-word descriptor

POINTER DESCRIPTORS

address of array descriptor attached to
(or Ø) byte index into that array row
(base of Ø) pointer-link

TRUTHSET DESCRIPTOR

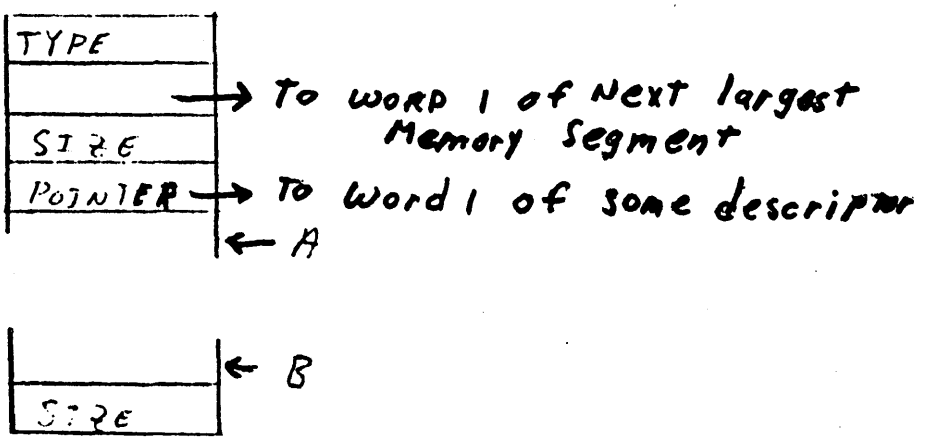


8 words for bit index from Ø to 177 octal.
Bits are numbered from right to left.

These descriptors are always present. They are either in the PRT or are built in the stack. Hence there is no presence bit for these descriptors.

APPENDIX J

MEMORY LINKS and MEMORY DESCRIPTORS



Memory is linked in a link list of memory areas or segments such that the smallest memory segment is first in the list and the largest memory segment is last in the list. The SIZE words contain the number of words between words A and B inclusive.

The TYPE word indicates the type of information contained between words A and B inclusive. If TYPE = 3"177772" then this is the PRT and stack memory area. If TYPE = 3"17777Ø" then this area is one of the File Information Blocks (FIBs). Otherwise the following bits in the TYPE word have the following meanings:

BIT 15 - SAVE BIT. If this bit is on then the memory contained between words A and B inclusive is so

important that either we or the programmer decided that it should not leave core once it was brought in. The programmer can create save storage through SAVE declarations or by the use of the LOCK statement.

BIT 14 - IN-USE BIT. If this bit is on then there is a descriptor somewhere that points to this area. Note that if a segment is saved then the in-use bit is also on.

BIT 13 - PROGRAM/DATA BIT. If the IN-USE bit is on and the memory in this segment is data then this bit is on. If the memory is program then it is off.

BITS 12-4 are unused.

If this segment is in use and the PROGRAM/DATA bit is on indicating that this segment contains ALGOL data, then the following applies. Otherwise Bits 3-Ø are unused.

BIT Ø - If on then this area contains REAL entries (2-word as opposed to 1-word entries).

BIT 1 - If this bit is on then this segment contains a dope

vector (4-word entries). Bit Ø will also be on.

BIT 2 - This is the I-HAVE-BEEN-CHANGED bit. If this bit is on when we are about to swap some data out, then the swap out to the swap file is necessary. If this bit is off then there is an exact duplicate of this data in the swap file and we need not write this data out this time. This bit is set when an array row is created, changed by an assignment to an indexed array element, or any replace statement, or whenever a read I/O is done to an array row.

BIT 3 - If this bit is on then bits Ø and 1 are also on and this area is a dope vector. This bit says that the dope vector points to a REAL array row. This is used in the indexing algorithm done for segmented arrays.

APPENDIX K

SPECIFICATIONS OF THE CODE FILE

1. The code file must be a BINARY file with a maxrecsize of 128. Under DOS that means that the file must be contiguous.

2. Record Ø (first record of the file) must contain the following information:

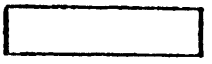
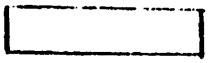
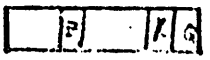
WORD	CONTENTS
Ø	Record number of where the PRT is located in the code file
1	number of words in the PRT
2	Ø (compiler puts the number of errors encountered here)
3	1 (compiler puts a 1 if this is a main program and a Ø if a bindable procedure)
4	Record number of where the external symbol table is located at
5	Size (in words) of the external symbol table
6	-1 (compiler sets this upon successful compile)
7	Link to the first program descriptor to be compiled. Word 2 of that descriptor then points to word 1 of the next segment to get compiled. This is for the binder. The last segment to get compiled has a Ø.

WORD	CONTENTS
8	Next segment number available. Used also by the binder.
9	Version of the compiler that compiled this code. This must agree with the version of the run-time system as the compiler generates version-dependent code.

3. The 13th word in the PRT (R+12 decimal) is the program descriptor for segment number 1 (the main block).

APPENDIX L

ABSOLUTE ADDRESSES IN THE ALGOL RUN-TIME SYSTEM

170*		Points to FIB of code file being run
172		Points to FIB of swap file
174		Option word 1 (options A through P)
176		Option word 2 (options Q through Z)

The only option to affect the operating system is the /S option (bit number 2 in word 176). If this bit is off, the following will happen on INPUT/OUTPUT to ASCII files. On input trailing spaces will be filled in place of and after the line terminator to fill the record for the specified word count. On output the entire record will be transferred with a carriage return-line feed after the last character to be transferred.

If this bit is on, the trailing spaces on input will be filled as trailing null characters (ascii zeros). On output trailing spaces will not be transferred.

* All addresses are in octal

On ASCII input then, the line terminators other than FORM feed will not be tranferred. If the line terminator is a carriage return line-feed then neither character will be transferred. Both will be filled.

APPENDIX M

PRT CELL ASSIGNMENTS

WORD*	CONTENTS
Ø	MKSCW for level 1
1	Not used
2	Used to hold zerosuppression character in binary to string conversion. Used also by REPLACESTMT in BY <ae> [FOR ae] replace.
3	
4	SCRATCH POINTER 1
5	
6	Used by compiler for temporary storage (only once in REPLACESTMT)
7	
1Ø	ARRAYS built here during declarations
11	
12	SCRATCH POINTER 2
13	Not used
14	
15	PROGRAM DESCRIPTOR for segment 1
16	
17	

* All address in octal

WORD*	CONTENTS
2Ø	I/O toggle (neither READ nor WRITE). Used to communicate between the compiled code and the I/O routines.
21	TOGGLE (READ-ONLY). Contains result of most recent SCAN or REPLACE.
22-31	ALPHAONLY truthset
32-41	NUMERIC truthset
42-51	ALPHANUMERIC truthset
52-61	SPECIAL truthset
62	First cell assigned by the compiler.

* All address in octal

APPENDIX N

ALGOL RESERVED WORDS[1]

All reserved words in PDP-11 ALGOL have the syntactical structure of identifiers. The reserved words are divided into three types: type 1, type 2, and type 3.

Type 1 reserved words are those words that cannot be used as identifiers, that is, they cannot be associated with any entity, declared or specified, in the program. In the reserved word list, type 1 reserved words are denoted by (1). For example, BEGIN(1).

Type 2 reserved words are those words that can be declared to be identifiers (overriding their previous meaning). That is, everywhere within the scope of the declared or specified entity, the type 2 reserved word references the declared or specified entity and not the function normally referenced by the reserved word. In the reserved word list, type 2 reserved words are denoted by (2). For example, ALPHAONLY(2).

1) Reprinted from B67ØØ/B77ØØ EXTENDED ALGOL MANUAL, Appendix A-1.

Type 3 reserved words are words that can be declared to be identifiers, but, when used in the language as specified by the syntax, have the reserved meaning. They are therefore 'context sensitive'. In other words, whenever an identifier that coincidentally spells a reserved word of type 3 is used in the language where the syntax calls for a reserved word of type 3, the identifier is not considered by the compiler to be reference to some entity, but rather the reserved word of type 3. If, however, the identifier appears in the language where the syntax does not call for a reserved word of type 3, the identifier is taken by the compiler to be a reference to some entity declared or specified in the program, in which case the particular entity being referenced is determined by the rules of scope. Note the difference in the example on the following page. Reserved words of type 3 are file mnemonics or file attributes. In the reserved word list, type 3 reserved words are denoted by (3). For example, KIND(3).

```
% THIS PROGRAM DEMONSTRATES TYPE 3 RESERVED WORDS
% USING THR TYPE 3 RESERVED WORD 'KIND'
BEGIN
  FILE F;
  INTEGER KIND;
% IN THE FOLLOWING STATEMENT 'KIND' REFERENCES THE INTEGER
% VARIABLE 'KIND'.
  KIND:=2;
% IN THE NEXT STATEMENT 'KIND' IS A TYPE 3 RESERVED WORD.
  F.KIND:=VALUE(PRINTER);
  KIND:=F.KIND:=KIND+1;
  END.
```

ABS(2)
ADDR(2)
ALPHA(2)
ALPHANUMERIC(2)
ALPHAONLY(2)
AND(2)
ARRAY(1)

BEGIN(1)
BINARY(2)
BOOLEAN(1)
BY(2)

CASE(2)
CHAIN(2)*
CLOSE(2)
COMMENT(1)
COMPILETIME(2)
CURRENTRECNO(3)

DECIMAL(2)
DEFINE(1)
DELTA(2)
DIGIT(2)
DIGITS(2)
DISK(3)
DIV(2)
DO(1)

ELSE(1)
END(1)
EQL(2)
EXTERNAL(1)*

FALSE(1)
FIELD(1)*
FILE(1)
FILL(2)
FIRSTONE(2)
FOR(1)
FORWARD(1)

GEQ(2)
GO(1)
GTR(2)

HEX(2)

IF(1)
IN(2)
INTEGER(1)
IO(3)

KB(3)
KIND(3)

LABEL(1)
LEQ(2)
LIST(1)*
LOCK(2)
LONG(1)
LP(3)
LSS(2)

MAX(2)
MAXRECNO(3)
MAXRECSIZE(3)
MIN(2)
MOD(2)
MYUSE(3)

NEQ(2)
NOT(2)
NUMERIC(2)

OCTAL(2)
OF(2)
ONES(2)
OPEN(3)
OR(2)
OUT(3)
OWN(1)

POINTER(1)
POLISH(2)
PRESENT(3)
PRINTER(3)
PROCEDURE(1)
PRTBASE(2)
PURGE(2)

READ(2)
REAL(1)
RELEASE(2)
REMOTE(3)
REPLACE(2)

SAVE(1)
SCAN(2)
SHIFT(2)
SHL(2)
SHR(2)
SIGN(2)
SIZE(2)*
SPECIAL(2)
STEP(1)
SWAP(2)
SWITCH(1)

TEKT(3)
THEN(1)
THRU(2)
TIME(2)
TITLE(3)
TO(2)
TOGGLE(2)
TRUE(1)
TRUTHSET(1)
TTY(3)

UNTIL(1)

VALUE(1)

WHILE(1)
WITH(2)
WORD(2)
WORDS(2)
WRITE(2)

ZEROSUP-
PRESSED(2)

* These words are reserved at the indicated level but the compiler does not know what to do with them yet.

APPENDIX O

ALGOL BUILT-IN FUNCTIONS

FUNCTION	RESULT
ABS(<ae>)	INTEGER. Returns the absolute value of <ae>.
*ADDR(<id>)	INTEGER. Returns absolute address of <id>.
*ADDR(<1-dimensional array name>)	INTEGER. Returns absolute address of first element in array.
*ADDR(<indexed array element>)	INTEGER. Returns absolute address of indexed element. NOTE. It should be noted that arrays get swapped in and out as need be and that the last two functions return the current absolute address.
BOOLEAN(<ae>)	BOOLEAN. Returns the value of <ae> as boolean.
#COMPILETIME(<integer>)	INTEGER. Allows the user to obtain various time functions at the time of compilation. The argument must be a constant between Ø and 5 inclusive. Refer to the TIME intrinsic for the values returned.

* EXTENSIONS to this ALGOL not found in EXTENDED ALGOL.

\# IMPLEMENTATION is slightly different from EXTENDED ALGOL.

FUNCTION	RESULT
#DELTA(<pe1>,<pe2>)	INTEGER. The number of characters given by <pe2> minus the number of characters given by <pe1> is returned. Does not check for segmented array as on the B67ØØ.
FIRSTONE(<ae>)	INTEGER. Returns the bit number of the left most non-zero bit, plus one. It is set to zero if no non-zero bit is found.
INTEGER(<ae>)	INTEGER. Returns the value of <ae>. When implemented properly, <ae> should be REAL and value returned is <ae>+Ø.5.
#INTEGER(<pe>,<ae>)	INTEGER. Returns the decimal value represented by the character string starting with the character indicated by the <pe>. The length is determined by the expression <ae>. The value is determined by interpreting the low order 4-bits of each character as decimal. B67ØØ allows pointer update in <pe>.
MAX(<ae1>, ...,<aeN>)	INTEGER. Maximum value of <ae1>, ..., <aeN> is returned. N ≥ 1.
MIN(<ae1>, ...,<aeN>)	INTEGER. Minimum value of <ae1>, ..., <aeN> is returned. N ≥ 1.
ONES(<ae>)	INTEGER. Returns the number of non-zero bits in <ae>.

FUNCTION	RESULT
POINTER(<1-dimensional array name>)	
POINTER(<indexed array element>)	A pointer is generated to point to the low byte of the word specified.
POINTER(<array row designator>)	
POLISH	INTEGER. When assigned to pushes the value onto the top of stack. When used in an expression, pops the top of stack for its value. Care should be exercised when using this intrinsic.
PRTBASE	INTEGER. This variable contains the value of the MKSCW of level 1. This variable should not be written into. Its address is PRT+Ø.
*SHIFT(<ae1>,<ae2>)	INTEGER. Returns the value <ae1> shifted to the right or left the number of bits specified by ABS(<ae2>). The shift is to the left if <ae2> is less than zero. Zeros are brought in from either side during the shift.
*SHL(<ae1>,<ae2>)	INTEGER. Returns the value <ae1> shifted to the left the number of bits specified by <ae2>. A negative <ae2> results in a shift to the right of -<ae2> bits.

FUNCTION	RESULT
*SHR(<ae1>,<ae2>)	INTEGER. Returns the value <ae1> shifted to the right the number of bits specified by <ae2>. A negative <ae2> results in a shift to the right of -<ae2> bits.
SIGN(<ae>)	INTEGER. Returns +1 if < ae> is greater than Ø, Ø if <ae> equals Ø, -1 otherwise.
#TIME(<ae>)	INTEGER. TIME makes various system times available to the user as follows: Ø - current second, 1 - current minute, 2 - current hour, 3 - day of month, 4 - month of year, (JANUARY is month 1) 5 - Julian year
#TOGGLE	BOOLEAN. Returns the result of the most recent SCAN or REPLACE statement. TOGGLE is set to TRUE if termination of the operation was due to max-count underflow and FALSE if termination was due to condition failing. Maxcount is checked before condition. TOGGLE is global in this implementation whereas on the B67ØØ it is local.

APPENDIX P

MNEMONIC FILE ATTRIBUTES

MNEMONIC	KIND	RESULT
CURRENTRECNO	INTEGER	Used only with BINARY files. Contains which record is to be READ/WRITTEN on the next I/O if it is sequential. READ/WRITE.
KIND	INTEGER	When read returns the filekind of the file. When written (not allowed on open files) sets this attribute.
MAXRECNO	INTEGER	When read will return the number of blocks in an ASCII file and the maximum record an I/O is allowed to in a BINARY file. This attribute cannot be changed on an open file.
MAXRECSIZE	INTEGER	When read/written returns/sets the MAXRECSIZE of the file. This attribute cannot be changed on open files.
MYUSE	INTEGER	When read/written returns/sets the MYUSE attribute of the file. This attribute cannot be changed on an open file.
OPEN	BOOLEAN	When read indicated whether the file is opened (TRUE) or closed (FALSE). When written will open or close a file depending upon whether the boolean value is TRUE or FALSE.

MNEMONIC	KIND	RESULT
PRESENT	BOOLEAN	When read indicates whether the file exists (TRUE) or not (FALSE). When written will create or purge the file depending upon whether the boolean value is TRUE or FALSE.
TITLE	STRING	When read or written (can only be done in REPLACE statements) returns or sets the external title of the file. When set, a null should follow the file name.

APPENDIX Q

COMPILE-TIME OPTIONS

There are eight compile-time options. These options control the compilation and listing of the source. Each control card can contain at most one compile-time option. A control card is denoted with an at-sign in column 1 and the compile-time option in column 2. Any other required information follows in column 3. If an option is not recognized, it is ignored. The options, when encountered, are togged from TRUE to FALSE and from FALSE to TRUE with the exceptions of the C, F, and I options. The options, their default conditions, and a description of what they do follows:

C<title>

This is the CHAIN option. It may be in either the primary or secondary input file. It closes the current source input file and opens the file denoted by <title> for the remainder of its input from that file.

That is, a chain option encountered within the INCLUDE file will close the INCLUDE file and open the file

denoted by <title> to be the INCLUDE file. This option will not reopen the file in which the CHAIN option was encountered. The <title> must appear in columns 3-35.

D (RESET)

When set prints out the code generated in both octal and B55ØØ assembler mnemonic form along with the source in the listing file.

E (RESET)

When set produces only an error listing. This is equivalent to a RESET-LIST on the B67ØØ.

F cannot be SET or RESET

When this option appears a top of form is printed in the listing file (only if the current page of listing is non-blank).

I <title>

This is the INCLUDE option. It can only appear in the primary input file. When it appears, it will open the file designated by <title> and read and compile source from this file until it encounters an end of file at which time

it will close the INCLUDE file and continue to read and compile source from the primary input file continuing with the card after the INCLUDE card. More than one INCLUDE card may appear in the primary input file. The <title> must appear in columns 3-35.

O (RESET)

When this option is set any non-control cards will be treated as comment.

P (RESET)

When set all address generated by the compiler will appear in the listing file as the compiler generates them.

APPENDIX R

COMPILER COMMAND STRING SWITCHES

The following are the allowable switches in the compilers command string. Only the first letter of each switch is significant. Any other switch than these will result in an 'ILLEGAL SWITCH' error message. The options and their results are:

/C option (CRUNCH)

The code file has to be pre-allocated because of the way DOS handles contiguous files. In order to compile the compiler (which is currently 126 blocks big) the code file is pre-allocated as being 14Ø blocks long. The /C option will crunch the file at the end of the compile to be only as long as is necessary. This option should only be used on files smaller than 75 disk blocks.

/D option (DEBUG)

Sets the default state of the @D, @P compile-time options to TRUE.

PREFACE

The purpose of this manual is to instruct ALGOL programmers in writing and using pointer expressions. This manual covers the various syntactical constructs related to pointer expressions and illustrates the pointer manipulations involved in their execution. Examples of ALGOL pointer expressions are given with complete explanations of their use. The B 6700 uses pointer expressions as in-line code strings generated by the compiler. Thus, extremely fast handling of alpha/numeric editing in ALGOL programs is permitted. The B 5700 compiler, in using a pointer expression, calls a communicate and passes parameters. This procedure slows down the processing speed but still allows effective use of pointer expressions.

Consequently, this manual has been written from the point of view of the user of the B 6700. Notes have been added, however, that explain the syntactical constructs needed by the user of the B 5700.

At present, the information in this training manual is up to date and accurate. This information is subject to change, however, because changes in system software are likely to occur. It will be advisable to refer to the latest edition of the B 6700 Extended ALGOL Information Manual for details about the most recent implementation of pointer expressions.

For their contributions to this manual, the author wishes to thank John Rooney and James Keen of the Field Support Organization, William Johnson of the Systems Support and Planning Department, and Phillip Shafer of the Advanced Development organization.

TABLE OF CONTENTS

1.0 DEFINITIONS

The following mnemonics are used in this manual. The definitions of these mnemonics are as follows:

Mnemonic	Definition
SB ::	Source string before execution
SA ::	Source string after execution
DB ::	Destination string before execution
DA ::	Destination string after execution
AE ::	Arithmetic expression
PE ::	Pointer expression

The term "pointer" is defined as follows:

pointer — a representation of the relative address of a character position with respect to the beginning of a one-dimensional array or an <array row> of a multidimensional array. Thus we say it "points" to a character position.

2.0 POINTER DECLARATION

A pointer word must be reserved in the B 6700 stack (in the B 5700 PRT) for each pointer identifier needed for the program. The syntactical rules for a pointer declaration are the same as for any ALGOL declaration.

Example:

```
BEGIN

        REAL A, B, C;

        POINTER POINT, PA, T;

        DEFINE P = POINTER# ;

        etc....
```

The pointer declaration exemplified above would reserve words for storing the addresses pointed to by POINT, PA, and T.

3.0 POINTER ASSIGNMENT STATEMENT

3.1 The pointer assignment statement assigns an address of a character within the designated array, which is the next character to be accessed within the string, in the reserved word designated by the pointer declaration.

Syntax:

<pointer assignment> ::= <pointer identifier>

<replacement operator> <pointer expression>

Example:

```
BEGIN

    ARRAY ABLE [0.3];

    POINTER PA, PB, PC, PD;

        PA := POINTER (ABLE); % CASE A

        PB := POINTER (ABLE [0]); % CASE B

        PC := POINTER (ABLE [2]); % CASE C

        PD := POINTER (ABLE [1]) + 3; % CASE D
```

In cases A and B above, the actions are identical. The addresses placed in PA and PB are the same.

```
PD := POINTER (ABLE);

(ABLE) SB :: ABCDEF  GHIJKL -----.
              ▲
              PA
```

PA now points to the first character of array ABLE.

In case C the address is set to the base location of array ABLE plus 2 memory words. Assumming that ABLE [0:3] was declared, then the array assignment is 0-1-2-3; therefore ABLE [2] is the third word of the array.

In case D the character desired is not the first character of the word but the fourth, as the character assignment of a word is 0-1-2-3-4-5.

```
PB := POINTER (ABLE [1]) + 3;

(ABLE) SB :: ABCDEF  GHIJKL  MNOPQR  STUVWX
                       ▲
                       PD
```

3.2 The B 6700 has three types of word format:

a. BCL -- Six-bit characters, eight characters per word.

b. EBCDIC -- Eight-bit characters, six characters per word.

c. NUMERIC -- Four-bit characters (on B 6700 only).

The B 6700 assumes an eight-bit character (EBCDIC) as a default value when not specified. The B 5700 assumes a six-bit character (BCL) as a default value. On the B 5700 only one character (six bits) is permitted; however the B 6700 can handle any one of the three. On the B 6700, when using the six-bit or four-bit case, a character size identifier of 6 or 4 must be specified.

```
PA := POINTER (A, L)
```

where L specifies character bits.

Example:

```
PA := POINTER (ABLE [1], 6);

(ABLE) SB :: ABCDEFGH  IJKLMNOP  Q ----.
                       ▲
                       PA
```

Once the six-bit configuration is assigned, the computer will assume all character movement to be eight characters per word, until a new assignment is made.

The statements PA := POINTER (ABLE) and PA := POINTER (ABLE, 8) are synonymous on the B 6700.

3.3 Restrictions: A pointer assignment statement may not point to a word beyond the declared area of the array declarations.

Example:

```
ARRAY BAKER [0:1];

POINTER PB;

PB := POINTER (BAKER [2], 8);

%% ILLEGAL ASSIGNMENT
```

3.4 Pointers may be assigned to each other as long as they are declared pointers.

Example:

```
POINTER PA, PB, PC;

      PA := POINTER (ABLE);

      PB := POINTER (ABLE [2]);

      PC := POINTER (BAKER [1]) + 3;

PA := PB; % PA now points to ABLE [2];

PA := IF (BE) THEN PB ELSE PC;
```

If Boolean expression (BE) is true, then PA will point to ABLE [2]; otherwise it will point to BAKER [1] + 3.

3.5 Pointers can also point to multidimensioned arrays; however, the pointer will always point to the last dimension.

```
ARRAY DICK [0:6, 2:8];

PA := POINTER (DICK [3,4]) + 3;
```

PA now points to the fourth row (0,1,2,3), the third word (2,3,4...), character four of array DICK.

4.0 UPDATING POINTERS

4.1 Pointers are not automatically updated after use; they can be reassigned to a new character by use of the following statement.

```
PA := PA ± N;
```

where N is a signed integer or arithmetic primary. If N is not an integer, it will be rounded prior to the execution.

N can move the pointer forward or backward. At run time a SEG ERROR will occur if the pointer overflows or underflows the boundaries of the array.

4.2 Two methods are available for updating pointers:

a. By assignment, as previously discussed.

b. By update constructs used in REPLACE and SCAN statements.

4.3 Syntax for Update Construct

```
::= <update pointer> : <pointer expression>
```

4.4 Example of Update Constructs

Basic example of update constructs follows:

```
REPLACE PA:PB. etc..

SCAN     B:Q.. etc.,

P:P

PB : PA + (A+B/C)       (Note:  arithmetic primary must be in parentheses.)

T : Q := POINTER (ARA [16], 8)
```

Example:

```
BEGIN    .
ARRAY ABLE [0:4];POINTER A;
A := POINTER (ABLE,8) + 4;
SB :: ABCDEF  GHIJKL  MNOPQR  ----
          ▲
          A

REPLACE P:A BY "ZYX";

SA :: ABCDZY  XHIJKL  MNOPQR  ----
          ▲    ▲
          A    P
```

Example:

```
SB :: ABCDEF  GHIJKL  MNOPQR -----

REPLACE PB : A := POINTER (ARA, 8) + 10 BY "CB" ;

SA :: ABCDEF  GHIJCB  MNOPQRS ----
                  ▲   ▲
                  A   PB
```

4.5 It may be necessary to use a pointer identifier as a recursive identifier. In these cases, the statement would occur as follows:

```
 REPLACE  PA:PA-6 BY etc....
```

At the start of this execution, PA would be decremented by 6 and the character transfer started. At the completion of the execution, PA would address PA-6 plus the characters replaced.

Example:

```
DB ::  ABCDEF  GHIJKL  MNOPQR ---
          ▲
          PA

REPLACE  PA:PA-2 BY PB FOR 6;

DA :: ABCDEF  GHIJKL  MNOPQR ---(DESTINATION)
                ▲
                PA
```

5.0 STRING STATEMENTS

There are four basic string statements:

a. String transfer statement.

b. String scan statement.

c. String translate statement.

d. File name change statement.

5.1 String Transfer Statement

The purpose of this statement is to transfer a string of characters from one area of memory to another.

5.2 String Scan Statement

The purpose of this statement is to test a string of characters for a match to a desired character or a desired type of character string.

5.3 String Translate Statement

The purpose of this statement is to transfer a string of characters from one area to another and translate the bit coding of each source character to a new character in the destination area.

5.4 File Name Change Statement

The purpose of this statement is to allow a programmer to modify certain file attributes within the program.

6.0 STRING TRANSFER STATEMENT

For these examples it is assumed that the following declarations have occurred within a program.

```
BEGIN

ARRAY ARA, ARB [0:13];

POINTER PA, PB, PC;

INTEGER I, J;

I := 80;

PA := POINTER (ARB);

PB := POINTER (ARA);
```

6.1 REPLACE <destination> BY <source> FOR <AE> <units>;

Example:

```
REPLACE PB BY PA FOR 6 WORDS;
```

This would transfer the first six words starting from ARA [0] through ARA [5] to ARB. This is an eight-bit character set; therefore, 36 characters would have been transferred.

Example:

```
REPLACE PA + 4 BY PB FOR 10;

(ARA) SB :: ABCDEF  GHIJKL  MNOPQR  STUVWX  YZ
              ▲
             PA

(ARB) DB :: THIS _I  S_A_SA  MPLE_O
              ▲
             PB

(ARB) DA :: ABCDTH IS_IS_ A_OPQR  STUVWX  YZ
```

Note that here ARA pointer PA was advanced forward by 0, 1, 2, 3, 4 before transferring started. Also note that only the addressed area is overwritten.

Example:

```
REPLACE PC:PB+7 BY PA:PA+2 FOR 3;

(ARA) SB :: ABCDEF  GHIJKL · MNOPQR  ST---
            ▲
           PA

(ARB) DB :: (blanks)

(ARA) SA :: ABCDEF  GHIJKL  MNOPQR  ST--
                ▲
               PA

(ARB) DA :: ------  -CDE--  ---
            ▲       ▲
           PB      PC
```

This illustrates the use of updating pointers for sequential replacement of a string of characters. PA was updated plus two digits to start the transfer; and it now points to F, the next digit to be transferred. PB was reset to its starting point, while PC was updated to point to the next character after E in ARB.

Example:

```
J := 8 ;

REPLACE PA:PC := POINTER (ARB [1], 8) BY PB:PB := POINTER

                    (ARA [2]) FOR J;

SB & DB NOT SHOWN

(ARA) SA :: ABCDEF  GHIJKL  MNOPQR    STUVWX
                            ▲         ▲
                           PB(before) PB(after)

(ARB) DA :: ------  MNOPQR  ST----
                    ▲       ▲
                   PC      PA
```

Four items of interest are shown here.

a. PC was first set to ARA [1], first character.

b. PA was updated to retain the final PC address.

c. PB was initialized and updated; this overwrote the initial assignment statement.

d. A variable was used as the character count.

Example:

```
REPLACE POINTER (ARB) BY POINTER (ARA [3]) FOR 2 WORDS;

(ARB) SA :: STUVWX  YZ0123 ----
```

Two items appear here:

a. Temporary (undeclared) pointers are used to point to ARB [0] and ARA [3]. These pointers are valid only during this statement.

b. Transfer was declared in words instead of characters.

6.2 REPLACE <destination> BY <source> FOR <arithmetic expression> <units>

Examples:

```
REPLACE POINTER (ARB) BY POINTER (ARA) FOR 6;

REPLACE PA BY PB FOR 80;

REPLACE PA BY PB FOR 14 WORDS;

REPLACE PA BY PB FOR IF I = 6 THEN 10 ELSE 4;
```

If <units> is <empty> then characters are assumed; otherwise use WORDS.

6.3 <source> FOR <max count> <condition>

This statement allows the transfer of characters to be under two controls:

a. The satisfying of a Boolean condition.

b. The exhausting of a maximum count of characters or words.

   1. A reserved word called TOGGLE is used in this construct. This word references a hardware (flip-flop) type logic which can cause severe problems. TOGGLE is not reset automatically and can travel from one statement to another in a set condition and can cause premature ending of the operation. In order to reset TOGGLE, a dummy pointer statement may be necessary.

   2. The following rules govern TOGGLE.

      (a) TOGGLE is reset when <condition> halts the statement.

      (b) TOGGLE is set when <max count> halts the statement.

      (c) TOGGLE is called as a Boolean identifier within the program.

3. An excellent construct for using TOGGLE is:

   DO <block using pointer expression> UNTIL TOGGLE;

   This allows the pointer expression to condition TOGGLE prior to the Boolean test.

6.4 <max count> ::= <residual count> : <arithmetic expression>

Example:

```
REPLACE PB BY PA FOR J:10 WHILE NEQ "A";

(ARA) SB :: BCDEAF  RPSTVW

(ARB) DA :: BCDE--

TOGGLE is set FALSE

J now equals 6
```

Notice that the value of J is decremented for each character transferred. J was set to 10 at the start of the execution and then decremented every time the WHILE NEQ "A" was TRUE.

Example:

```
J := 7;

REPLACE PA BY PB FOR I:J WHILE LEQ "O";

SB :: ABCDEF  GHIJKL  MNOPQR--
```

Since J = 7, the Boolean would not be satisfied and the transfer would end after seven characters had been transferred.
TOGGLE is now TRUE, and I = zero.

Example:

```
REPLACE PA BY PB WHILE LSS 6;
```

This construct allows scanning until the Boolean test for 6 is FALSE.
A maximum count of 524,287 is assumed.

6.5 Condition

Condition has four forms:

a. WHILE <relational operator> <arithmetic expression>

b. UNTIL <relational operator> <arithmetic expression>

c. WHILE IN <table pointer>

d. UNTIL IN <table pointer>

6.6 WHILE and UNTIL are similar constructs which can be interchanged.

a. REPLACE PA BY PB WHILE LSS 0;

b. REPLACE PA BY PB UNTIL GEQ 0;

Both "a" and "b" above will do the same operation; however the test for TRUE is made under a different perspective.

6.7 WHILE IN and UNTIL IN are again quite similar in use; however, the TABLE POINTER requires explanation.

6.8 <table pointer> ::= ALPHA | <subscripted variable> | ALPHA6 | ALPHA8
where <subscripted variable> ::= <array name> [subscript list]

Examples:

REPLACE PA BY PB WHILE IN ALPHA

REPLACE PA BY PB UNTIL IN ALPHA

ALPHA here is defined as A through Z, 0 through 9.

REPLACE PA : PA BY PB : PB WHILE IN ALPHA6;

This construct calls a fixed table which checks only six-bit (BCL internal) characters.

REPLACE PA:PA UNTIL IN ALPHA8; this is the same as using ALPHA;

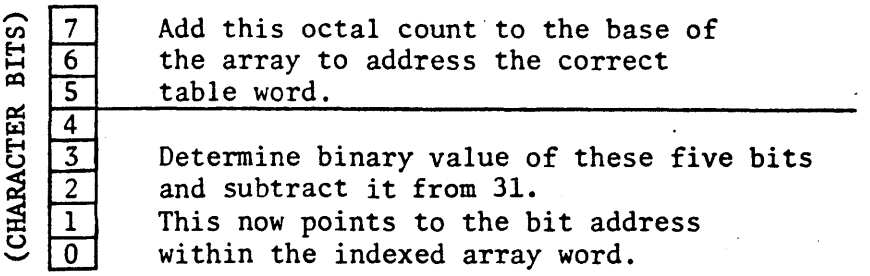
6.9 Table Pointers

There are times when it is desired to do a search for a special set of characters or bit configuration other than the ALPHA6 of ALPHA8 array on the system. These cases require you to generate your own table array.

The table array is an array of eight words of which the least significant 32 bits of each word are used as Boolean test bits.

By decoding the bits of each character, an address pointing to one bit in one of the eight words of the table array can be selected. If this bit is "on" then the test is TRUE, otherwise it is FALSE.

The tested character is hardware decoded in this fashion.



Example:

Suppose we have an array called

ARRAY SPECIAL [0:7];

and we wish to test for an A which is

EBCDIC (1100 0001).

The (110- ----) would equal an octal 6; therefore this would add to the base of ARRAY SPECIAL to point to address SPECIAL [6].

The (---0 0001) would be subtracted from 31 for a count of 30.

The test would now be on the 30th bit of SPECIAL [6], counting the least significant bit as 0.

TEST WORD =

				29	26	23	20	17	14	11	8	5	2
			31	28	25	22	19	16	13	10	7	(4)	1
			(30)	27	24	21	18	15	12	9	6	3	0

If bit 30 was "on" (one) then the test would be TRUE.

Example:

Suppose a test for a $, EBCDIC (0101 1011).

(010- ----) would indicate SPECIAL [2].

(---1 1011) would be Binary (27)

31 - 27 equals bit four (4) from word SPECIAL [2].

6.10 <source part> ::= <string> <optional unit count>

This statement transfers the string under the control of the unit count. If the unit count calls for fewer characters than the string provides, the leftmost characters will be transferred.

Example:

```
REPLACE PA BY "ABCDEF" FOR 3;

DA :: ABC---
```

If the unit count is omitted then the entire string is transferred.

Example:

```
REPLACE PA BY "THIS IS A STRING";

DA :: THIS IS A STRING
```

This could be written:

```
REPLACE PA BY "THIS IS A STRING" FOR 16;
```

If it was written

```
REPLACE PA BY "THIS IS A STRING" FOR 18;
```

then undetermined run-time errors are returned.

If a string is less than a word (48 bits), the word will be concatenated with itself to create a full word whose characters are transferred repeatedly to satisfy the unit count.

Example:

```
REPLACE PA BY "ABCD" FOR 10;
```

The word ABCD would be made into a word ABCDAB.

Therefore the DA would be ABCDABABCD.

```
REPLACE PA BY "EMPTY" FOR 10 WORDS;
```

will make 10 words in PA read EMPTYE. Note that if PA was not pointing to the beginning of the word, it would be updated automatically to the next word.

6.11 If the bit size is needed it can be added.

a. BY 8 "ABCDEF" is equivalent to BY "ABCDEF".

b. If the BCL mode is being used then

BY 6 "ABCDEFGH" is used. (Eight bits are standard on B 6700; six bits are standard on B 5700.)

6.12 The form <arithmetic expression> <unit count> will transfer the character or word once for each unit count number, with the character type determined by the destination assignment statement. (Translation can occur.)

Example:

```
ALPHA X;

REPLACE PA BY X FOR 3 WORDS;
```

If X = ABCDEF (eight-bit character) then the string pointed to by PA would equal ABCDEF ABCDEF ABCDEF.

6.13 Conversion from Arithmetic to Alpha

Whenever there is a need for printing or storing a type REAL or INTEGER in alpha mode, the word DIGITS indicates a conversion to decimal digits is needed.

Example:

```
I := 63;

REPLACE PA+4 BY I FOR 4 DIGITS;
```

Internally, I now equals hexadecimal 3F.

On execution:

```
DB :: ABCDEF GHIJKL

DA :: ABCDOO 63IJKL
```

a. The unit count indicates the number of alpha characters to be replaced in the destination string.

b. If I was type REAL, then it would have been rounded into an integer before conversion.

c. A limitation of 12 characters maximum is placed on the statement by the hardware.

7.0 STRING SCAN STATEMENT

Whenever a scan of a string of characters is needed for the interrogation of a wanted character, the SCAN statement is used.

Syntax:

```
<string scan statement> ::= SCAN <source> FOR <max count> <condition>
                            or
<string scan statement> ::= SCAN <source> <condition>
```

The use of <max count> <condition> and <condition> are covered under REPLACE in section 6.

Example:

```
SCAN PA:PA FOR 12 UNTIL EQL "A";

SB :: 123456  7890AB
      ▲
      PA

SA :: 123456  7890AB
                ▲
                PA
```

PA will be pointing to A on completion of this operation.

Example:

```
SCAN PA:PA FOR 12 WHILE GEQ "0"; (ZERO)
```

This will also stop with PA pointing to A. However, it would also stop with any other alpha character.

Basic examples of SCAN statements:

```
SCAN PA:PA + 6 WHILE IN ALPHA;

SCAN PA:PA - 3 UNTIL IN ALPHA;

SCAN PB:PA FOR J : 10 WHILE NEQ " ";

SCAN PB:PA := POINTER (ARA [3]) + 3 FOR J:I UNTIL = "0";
```

Example:

```
SCAN PC : PA := PB-N FOR K : N + 3 WHILE IN TRUTHTABLE;
```

This statement would be executed in the following sequence:

a. The address in PB would be subtracted by N characters and the new address stored in PA.

b. The value in unit count would be N + 3, and the scan would begin.

c. TRUTHTABLE would be a pointer membership table as shown in paragraph 6.9.

d. On completion, PC would be the updated value of PA, and K would be value of (N + 3) down counted for every character scanned.

e. If (N + 3) characters were scanned, then TOGGLE would be set and the operation terminated.

7.1 Remarks

Whenever a scan is incremented or decremented upon assignment, the tally of the <residual count> is not counted to reflect the skip value in the pointer expression.

Example:

If reading an 80-column card

```
SCAN PA:PA+6 FOR 80 WHILE IN ALPHA;
```

would scan the 86th character before stopping on <max count> condition.

For this reason, a scan statement being used in a repetitive block would be written as follows:

```
SCAN PA:PA+N FOR CNT:CNT-N WHILE GTR "0";
```

In this case, PA would be incremented by N while on each execution CNT would first be decremented by N. Therefore, a count of 80 for CNT would be accurate to scan an 80-character card.

8.0 STRING TRANSLATE STATEMENT

The B 6700 handles the BCL, EBCDIC, and USASCII character sets. Internal translation from one character set to another is done by the use of software translation tables.

The string translate statement employs hardware speed in the search of a software table in order to translate at high speed. At the present time there are two tables resident in the system which can be called programmatically. These tables are as follows:

a. EBCDICTOBCL (EBCDIC to BCL).

b. BCLTOEBCDIC (BCL to EBCDIC).

8.1 Translator Table

The translator table is an array that is 16 words long for BCL and 64 words long for EBCDIC. A table word is laid out as follows:

		0	1	2	3	Character Position
		W	X	Y	Z	Translate Character

The indexing into the table is done in the following manner for each character to be translated:

Input character in eight-bit code:

7	Points to the array word in EBCDIC
6	to BCL translation.
5	Points to the array word in
4	BCL to EBCDIC translation.
3	Array base plus increment.
2	
1	Points to character within
0	the word.

Take the EBCDIC 1 which is (1111 0001).

a. The (---- --01) would select number one character position.

b. The (1111 00--) would be added to the array base address as BASE + 60 (binary value of "111100") for EBCDIC translation.

In the above chart the X would be substituted for the EBCDIC "1" input. The index would be TABLE [60], character one.

8.2 String Translate Syntax and Example

Syntax:

REPLACE <source> BY <destination> FOR <unit count> WITH <translate part>;

Example:

```
REPLACE PA BY PB FOR 30 WITH BCLTOEBCDIC;

REPLACE PB := POINTER (ARA,6) BY PA := POINTER (ARB [2], 8) FOR CHARLENGTH

WITH EBCDICTOBCL;
```

Note that the pointer bit value denotes the bit size of each character.

8.3 Multiple Assignments

The REPLACE statements can have multiple <string relation> expressions following the pointer expression.

Examples:

```
REPLACE PA BY PB BY "ABC" FOR 3, "XYZ" FOR 3;

.REPLACE DATE BY "JANUARY ", DAY FOR 2 DIGITS, ", 1971" FOR 1 WORD;
```

9.0 POINTER-VALUED ATTRIBUTES

The file attribute TITLE is a pointer-valued attribute and is accessed without a pointer assignment statement. The syntax for this expression follows.

Syntax:

```
REPLACE <file name>. TITLE BY "<title list>.";
```

Example:

```
REPLACE DEST.TITLE BY "NEWTITLE/ONE/TWO.";

REPLACE SORCE.TITLE BY "CARDS.";
```

9.1 A title may be assigned also from an array in the following manner.

```
BEGIN

    POINTER PA;
    ARRAY ARA [0:5];
    FILE OUTTAPE;
    REPLACE PA:= POINTER (ARA,8) BY "NEWTAPEFILE";
    REPLACE OUTTAPE.TITLE BY PA FOR 11, ".";
    etc...
```

This would change the orginal title of the file OUTTAPE.to a new title NEWTAPEFILE. Notice that in each case the name must be terminated by a period and that a limitation of 17 characters maximum is placed on the number of characters used in the title.

10.0 COMPARE EXPRESSIONS

When the interrogation of a string of characters is necessary, one of the following statements is used.

a. IF <pointer expression> <boolean relation> <string> THEN...
   IF PA = "CARDS" THEN ... ELSE ...;

b. IF PA:PA:= POINTER (ARA) NEQ PB:PB FOR 6 THEN ...

c. IF PB:= POINTER (ARB) EQL "NEXT ITEM" THEN ...

d. IF PA:PB EQL "ABCDEF" FOR 12 THEN ...

The use of these expressions is treated like the use of a normal IF statement. The pointer terms are used to find the selected characters to be scanned. After the pointers are set, a character-by-character compare is made. The outcome of this comparison is used to determine whether or not the THEN or ELSE branch of the statement is to be executed.

11.0 DELTA FUNCTION

The DELTA function will return to the program the amount of character displacement between the two pointers declared.

The statement is as follows:

```
DELTA (P1, P2);
```

where P1 and P2 are pointer parameters.

```
SB :: ABCDEF  GHIJKL  MNOPQR
      ▲       ▲
      A       B
```

The expression DELTA(A,B) would return 7.
The expression DELTA(B,A) would return -7.

Examples:

```
IF DELTA(A,B) NEQ 0 THEN ....

SCAN PA:PB+DELTA(A,B) FOR ....

SCAN PA:PB FOR J: DELTA(PA:PB) WHILE ....

J := DELTA(PA:PB);
```

12.0 TYPE TRANSFER FUNCTIONS USING POINTERS

Whenever an alpha character is needed for arithmetic expressions, a conversion to arithmetic notation is required.

The integer function handles this.

```
INTEGER (PE,AE)
```

where PE = Pointer Expression and AE = Arithmetic Expression for number of characters to translate.

Example:

```
BEGIN

  INTEGER I;
  POINTER PA;
  ARRAY ARA [0:1];

  REPLACE PA:=POINTER(ARA) BY "ABC63D  EFGHIJ";
  I := INTEGER (PA+3,2);
```

This will convert the 63 to a 77 octal and store it in I.

13.0 LIST STATEMENTS USING POINTERS

Pointer identifiers can be used in read and write statements as the list identifiers. These statements must be used with a format other than a "*". Note: The format prevails and controls the termination of the read/write statement.

The following formats can be used with pointer expressions.

a. Type A or C. The number of characters specified by the W field are transferred; the pointer is used as the starting character location.

   The input is considered type ALPHA; the bit configuration is determined by the pointer assignment statement.

b. Type Ø. Similar to type A except that a word transfer takes place.

c. Type V. Used for controlled editing.

14.0 USING PICTURE IDENTIFIERS

With a pointer expression, a transfer of characters from one string to another can be done by using a picture identifier to edit the data during the transfer. This facility allows the programmatic insertion of periods, commas, dollar signs, etc., as desired.

Syntax:

```
<string statement>::= REPLACE <destination> BY <source> WITH
        <picture identifier>
```

14.1 Picture Declaration

The editing string is shown in the picture declaration with an identifier. The syntax for this construct is as follows:

```
<picture declaration> ::= PICTURE <identifier>(<picture>)
                            or
<picture declaration> ::= PICTURE <identifier> (<picture>), <identifier>
      (<picture>) etc....
```

Example:

```
BEGIN
  REAL A,B,C;
  PICTURE CHECK (FFBOIAA);
  PICTURE FINAL (F(3)IA(2)), SUB (A(3)I);
  INTEGER I,J;
  etc.
```

A picture consists of a named string of editing symbols which are enclosed in parentheses. The named string is composed of a mixture of five types of control characters.

a. Introduction codes.

b. Control characters.

c. Single picture characters.

d. Picture characters.

e. Picture skip characters.

14.2 Picture Editing

The following operations can be implemented by the use of pictures.

a. Unconditional character moves.

b. Move characters with leading zeros.

c. Move characters with leading zeros and floating character insertions.

d. Move characters with conditional character insertion.

e. Move characters with unconditional character insertion.

f. Move numeric part of character only.

g. Skip source characters (forward or reverse).

h. Skip destination characters forward.

i. Insert overpunch sign on previous characters.

14.3 Introduction Code

There are six letters (called introduction characters) that are used to introduce characters into a destination string. The introduction characters are as follows:

B | P | M | C | U | N

These characters are used to instruct the system to do a specific task on the present character within the string. The syntax of the construct follows:

::= <introduction code> <new character>

The meaning of each introduction code follows:

B - Replacement for leading zeros.	(SPACE)
C - Conditional insertion of new character.	(,)
N - Unconditional insertion of new character.	(.)
M - Insert new character if field is minus.	(-)
P - Insert new character if field is plus.	(+)
U - Special floating character insertion.	($)

After each meaning above, a default value for an introduction character is shown in parentheses. This character is assumed to be the new character, unless assigned another character within the string.

Example:

If the introduction code is N and the new character is 0, then the construct would be as follows:

NO

By using this construct within a string

"IIINOIN.II" in the picture would cause
"...0.." to be written into the destination string.

The default value for N = (.); this generates the three periods when inserted by I. However, N was given a new picture of 0. Therefore, a 0 was inserted next and then N was reassigned back to a period to complete the last two periods.

14.4 Picture Character

Picture characters are used with the introduction code to control the editing of the source string. The syntax for this follows:

::= <picture character>(<repeat part>)

The picture characters listed below perform the following actions:

A - Move <repeat field> characters from source to destination.

9 - Move <repeat field> numeric portion only of the source character to destination.

E - Editing Move. For <repeat field> count move numeric portion of character from source to destination with the following editing:

a. Suppress leading zeros by inserting a B character.

b. If the field is positive, then insert a P character in front of first nonzero character.

c. If the field is negative, then insert an M character in front of the first nonzero character.

d. At the first nonzero character, end the float action and transfer characters from source to destination to exhaust <repeat field>.

Remember, by default, B = blank, P = +, and M = - above.

F - Editing Move. Insert $. This is similar to E above with the following actions:

a. Suppress leading zeros by inserting a B character.

b. At the first non-numeric character, insert a U character ($ by default) before character and transfer characters as in E.

D - If the E or F float operation was terminated by a nonzero, then insert a C character (,).

If E or F terminated on repeat field count, then insert a B character (blank).

R - If the E or F operation was terminated by a nonzero, then insert the M character (-).

If the E or F terminated with exhausted repeat field, then insert a P character (+).

I - Insert the N character (.) unconditionally.

X - On input operations, skip the destination field forward by the number of characters indicated in repeat field count.

On output operations, insert repeat field count of blanks in the string.

14.5 Control Characters

Two characters are used to force special control operations into the stream.

Q - This forces a sign overpunch to be placed in the preceding character, if the field is negative.

: - This reinitiates the placement of leading zeros into the stream.

14.6 Single Picture Character

There are two characters which are used without repeat fields. These characters are as follows:

J - If an E or F float operation has not inserted a float character, then stop the float operation and insert the U character ($). If the U character has been inserted, then this character is a no-operation character.

S - If the field is plus, insert a P character (+); otherwise insert an M character (-).

14.7 Picture Skip Characters

There are two skip characters used, and these perform the following action:

< - Skip the source pointer backward by the repeat field count.

> - Skip the source pointer forward by the repeat field count.

14.8 Picture Semantics

Note: At the time this manual was being written, the picture constructs were not fully implemented on the B 6700; therefore, these constructs were not tested. The following description was written, however, to give the user some insight about the purpose of picture clauses and their use. The statements described here will probably need modification if they are to be compiled successfully.

Picture constructs are used principally in banking and payroll applications, particularly in editing the dollar quantity on checks.

As an example, consider setting up a string of digits to write the amount on a check. The quantity 000164375 is used to print $1643.75. Also, 000000065 is to be printed $.65. In order to do this, a picture declaration would appear as follows:

```
PICTURE CHECKTOTAL (F(7)JIA(2));
```

Example:

```
BEGIN
  REAL A,B,C;
  PICTURE CHECKTOTAL (F(7)JIA(2));
    . . .
    . . .
    . . .
   REPLACE PA:PA BY PB:PB FOR 9 WITH CHECKTOTAL;
    . . .
    . . .
END
```

The following actions should take place.

a. The "F(7)" indicates scan the first seven characters for a nonzero digit; when nonzero digit is encountered then add the preceding $ sign and transfer the remaining characters to exhaust the repeat field.

b. If the "F" <repeat count> has been exhausted and no "$" inserted, then unconditionally insert the "$".

c. In either case, now insert the N character, which is the period (.).

d. Transfer the last two digits.

Therefore,

```
123456789 would be edited as  $1234567.89
000006789 would be edited as       $67.89
000000089 would be edited as         $.89
000000000 would be edited as         $.00
```

Example:

Suppose a different format were desired where

```
123456789 would be edited as $1234567.89
000006789 would be edited as  ....$67.89
000000789 would be edited as  .....$7.89
000000089 would be edited as  .....$0.89
```

The picture for this would be as follows:

```
(B.FFFFFFJAIAA)

     or

(B.F(6)JAIA(2))
```

15.0 POINTER EXAMPLES

This section includes tested pointer sentences and brief write ups on their use.

15.1 The following is a define construct used in scanning an input card for free-form format words.

```
DEFINE
    SCN(N) = SCAN PA:PA+N FOR LM:LM-N UNTIL GEQ "A"#,
    RPL(H) = REPLACE PB BY PA FOR N:H WHILE GTR "Z" #,
    CKATT(ST, SCNLG, ERNO, RPLG, CND, CODE) =
         IF PA=ST THEN
         BEGIN
         SCN(SCNLG); IF TQGGLE THEN ERR(ERNO)
                     ELSE RPL(RPLG);
         IF PA LEQ "Z" OR CND THEN ERR(ERNO) CODE
         SCN(N); SETATT :=TOGGLE; GO FOUND;
         END; #;
    COMMENT OF CKATT
         ST IS THE STRING TO BE MATCHED ON THE CARD,
         SCNLG IS THE NUMBER OF CHARACTERS IN THE STRING,
         ERNO IS THE CASE # OF THE ERROR MESSAGE IN ERR,
         RPLG IS THE LENGTH OF NUMERIC DATA TO BE TRANSFERRED,
         CND IS CHECK ON THE CONVERTED VALUE OF NUMERIC DATA,
         CODE IS A TEST CODE FOR NUMERIC DATA LIMITS;
```

A call on this define would be written as follows:

```
CKATT("PARITY",6,10,2,T:=INTEGER(PB,N:=1)<0 OR T>1,
      ELSE S.PARITY:=T;  );
```

The information on the scanned card would read:

```
PARITY = 1  (any place on the card)
```

Initially, the value of LM:=80, and the result of the operation would set a 1 into S.PARITY. The construct would call procedure ERR, if TOGGLE was set, indicating that a count of 80 was reached, or if T was greater than 1 (T>1), or if T was less than 0 (T<0).

15.2 If it is desired to use free-form format within a field, then the following construct can be used.

Suppose that columns 20 through 25 of a card make up a field called CFIELD. The characters "----16" or "000016" could be punched in columns 20 through 25. However, it is easier to start in column 20 and punch "16" and then use pointers to determine the size of the number in significant digits.

```
BEGIN
    POINTER TMP;
    INTEGER I,J;
    DEFINE P = POINTER#;
    ARRAY TM(0:13);

    TMP := P(TM);
    SCAN TMP:TMP+20 FOR I:6 UNTIL GTR "0";
       % THIS FIND FIRST NONZERO DIGIT AFTER COL 20
    SCAN TMP FOR J:I UNTIL LSS "0";
       % THIS COUNT DIGITS UNTIL BLANK
    CFIELD := INTEGER(TMP,(I-J);
       %(I-J) DETERMINES VALID DIGITS TO CONVERT


END.
```