HAI*BAS

REFERENCE

MANUAL

COPYRIGHT NOTICE

This manual describes a proprietary product of OVAL SOFTWARE.

Copyright (C) 1994 - 2010. All rights reserved.

OVAL SOFTWARE has taken care to ensure that the contents of this manual are accurate, but cannot be held responsible for the consequences of any inaccuracies.

OVAL SOFTWARE reserves the right to change the product specification as a result of our policy of continued product development.

HAI*LINE, START*LINE and HAI*EUROP are registered trademarks of HOLLAND AUTOMATION INTERNATIONAL

Last revision – April 2010

TABLE OF CONTENTS

1. INTRODUCTION

Business Basic

HAI*Basic is designed to meet the requirements of business applications. It contains many easy-to-use facilities to realise high-quality applications.

Highlights

Easy-to-use display and keyboard control statements with a full range of options are available to control color displays and for keyboard input validation (see the ACCEPT statement).

Accurate computation with a large number of decimals when needed.

At the same time the variables occupy the smallest possible space by defining the length for every individual variable (see the DIM statement). This provides for very efficient use of both internal memory and disk space.

HAI*Basic has a set of statements to provide for easy access to peripherals and files. These features also include access to indexed files and multi-user facilities on record level.

HAI*Basic has built-in facilities to display windows with in-context HELP texts.

HAI*Basic versions

This manual applies primarily to the HAI*Basic Plus versions on DOS, IBM PC-network and Novell Netware. Earlier versions of HAI*Basic do not have all facilities described in this manual.

Organisation of this manual

This reference manual contains the following chapters:

Language features
Describes the general concepts of the HAI*Basic language. Describes all commands, functions, statements and system variables in alphabetical order.

Drivers
Describes all features of the peripheral drivers and pseudo-drivers for communication with the host operating system.

Error codes
Describes the handling of errors and exceptional conditions. It enumerates all error codes and their meaning.

System parameters
Describes the definition of the parameter file HAI.PAR

File structure
Describes the internal structure of the HAI*Basic files.

Utilities
Describes the function of the HAI*Basic Formatted List utility and the HAI*Basic compiler.

2. LANGUAGE FEATURES

2.1. INTRODUCTION

This chapter starts with a description of the general concepts of the HAI*Basic language. It also presents the details of all HAI*Basic statements, commands, functions and system variables.

Statements

Statements are instructions within a HAI*Basic program and they are therefore preceded by a statement number.

Example: 1010 PRINT "Good morning"

Statement numbers range from 1 to 64999.

Commands

Commands are instructions that do not make sense within a HAI*basic program.

Example: LIST 10,200

The distinction between statements and commands is not always very strict:

1. Certain statements and command are succesfully executed as an instruction by the source interpreter, but the do not function in compiled programs.

   Example: TRACE

2. Statements typed in without a preceding statement number are considered as a command and they are immediately executed. They are useful for test purposes. It is the responsibility of the programmer that they make sense in the test environment.

   Example: PRINT ASC(T$(7,1))

Multiple statements

A statement may consist of more than one substatement seperated by backslashes.

Example: 250 A=1 \ B=0 \ C$=""

See also the use of multiple statements in the IF ... THEN statement.

Maximum length The maximum statement length is 250 characters.

It is possible to type in statements longer than 250 characters by using function key F2. This is not recommended since it is not possible to edit these long statements (except by retyping them).

2.2. EXPRESSIONS

Expressions consist of references to constants, variables and functions. These expression elements are tied together by operators.

Example: (MOD(C,7)+1)*10

The simplest expression is a single numeric constant (like 1) but expressions can have any degree of complexity.
The reader of this text is supposed to have a general knowledge about expressions in programming languages.

Data types

There are two types of data: numeric data and string data.

Numeric data
Numeric data is written as a signed or unsigned integer number consisting of at most 14 digits.
The actual value ranges from - 140 737 488 355 328 to
+ 140 737 488 355 328 due to its internal representation as a binary number in two's complement notation. Its length ranges from 1 to 6 bytes (see the DIM statement for more details).

String data
String data consists of at most 1024 characters represented by the 8 bits an a byte.

Internal representation

The internal character codes are almost identical to the character codes on IBM-like PC's running PCDOS or MSDOS:

0 to 31	Used for special purposes. They cannot by used in a regular PRINT statement.
32 to 223	According to DOS on IBM-like PC's.
224 to 255	Not used.

You can however use any value from 0 to 255 as a string character (see the CHR function).

Data type conversion

Implicit data type conversion in expressions does not exist.
Conversion between numeric data and string data is explicit by using the functions ASC, CHR, VAL, STR and the mask operator.
An illegal mix of data types results in error code 62.

Numeric constants

A numeric constant is a signed decimal number.

Examples: 10, -3, 12345678901234

String constants A string constant consists of 0 to 1024 characters enclosed in quotes.

Examples: "ABC", "P Q", " ", ""

Numeric variables The name of a scalar numeric variable can be a letter A to Z, optionaly followed by a digit 0 to 9.

Examples: A, D5, Y7.

Numeric arrays have the same naming convention, but the name is followed by a number of dimensions definitions between parentheses.

Examples: A(10), D5(20,4), Y7[10], F3[1,2,3]

String variables The name of a scalar string variable can be a letter A to Z, optionally followed by a digit 0 to 9 and then always followed by a dollar sign $.

Examples: C$, F4$, Z3$

String arrays have the same naming convention, but the name is followed by a number fo dimension definitions between *square brackets* (The parentheses are used for substring references).

Examples: C$[5], F4$[3,3], Z3$[5,10,20]

Current string length String variables (and all elements of a string array) have a current length ranging from 0 to the DIMmed length (see the LEN function).

Substring reference Part of the contents of a string variable (or of a string array element) can be referenced as a substring.

Examples:

A$(5,L) L characters starting at position 5.

Y$(P) All characters starting at position P until the current length of the string.

H$[I+1,J](8,1) The 8th character of the array element.

The name of a numeric variable may be re-used for a string variable and for a numeric and string array, since the $ and the (and the [are part of the name. The variables B, B$, B(10,20) and B$[10,20] are all different and they may exist at the same time.

The first element of an array has subscripts (1), (1,1), (1,1,1) etc.

See the DIM statement for more details.

System variables

There are five system variables in HAI*Basic: DAY, HELP, PASS, PASS$ and USER.
See the definition of these variables in this chapter and also the LET statement.

Operator priority

The following operators are available. They are listed in descending order of priority. Operators without a separating blank line have equal priority.

Arithmetic operators

- unary minus

* multiplication
/ division

\+ addition (or string concatenation)
\- subtraction

Mask operator

:

Relational operators

= equal to
< less than
<= less than or equal to
> greater than
>= greater than or equal to
<> not equal to

Logical operators

NOT
AND
OR

Arithmetic operators

The arithmetic operators act as usual.

String operator

The concatenation operator + appends the second operand to the first operand.

Examples: F$+"ABC"
"PQ"+J$+"xy"

Overflow

An overflow error is produced in any of the following circumstances:

1. Assignment of a too large numeric expression result to a numeric variable.
2. Assignment of a too long string expression result to a string variable.
3. A numeric value does not fit in the string format as defined by the mask definition.
4. Attempt to divide by zero using the division operator / or the MOD function.

Note: The intermediate numeric result during expression evaluation may have a length of 28 decimal digits. The result of the entire expression must not exceed 14 digits (apart from other restrictions dependant on the purpose of the expression result).

The error code is 0. The overflow error can be trapped using the ON OVERFLOW GOSUB statement. Ignoring the overflow error causes incorrect expression results.

Relational expressions

Relational operators compare (sub)expressions. The two operands must either have both numeric type of both string type.
The result of the operation is either the numeric value -1 (true) or 0 (false).
String values are compared according to the internal character codes.

The following examples are all true:

"ABC" < "B"
"ABC" < "ABCD"
"Z" < "a"

The values true and false are numeric values. You may use them in expressions.

Example: LET A=B=10 is a valid statement. The result in A will be -1 or 0.

Logical operators

The logical operators are used to combine the result of relational operations.

Examples: IF A>B AND C<10 GOTO 2010

AND, OR and NOT operate on the individual bits. This is consistent with the definition of the values true and false. The value -1 in two's complement notation consists of all binary ones. The value 0 consist of all binary zeros.

Mask operator

The mask operator converts a numeric value to a formatted printable string representation according to the mask definition.

Examples: P$=A:"###0.00"
PRINT A:M$

The operation is controlled by the mask characters:

0	yields 0 if leading zero.
#	yields spaces if leading zero.
*	yields * if leading zero.
$	yields a $-sign in the rightmost $-position (floating $-sign).
.	yields decimal point unless preceded by a suppressed leading zero.
,	yields comma unless preceded by a suppressed leading zero.
+	yields + if positive value or else -
-	yields space if positive value of else -
/	yields a slash /

Functions

Functions need zero, one or two arguments and return a numeric or string value.

Display and printer control function have string type. The following example is valid:

```
2020  D$=CS+@(20,10)+"Empty Screen"
2030  PRINT D$
```

See the function definitions in this chapter.

2.3. FILE MANAGEMENT AND PERIPHERAL DRIVERS

File management

HAI*Basic has a set of powerful file management statement for
- sequential,
- random on number and
- random on index
file access.

These statements are:
ENABLE and DISABLE
OPEN and CLOSE
READ, WRITE and INSERT
DELETE
INPUT and PRINT
GET and PUT

All file management statements have a format similar to:

OPEN (n UNIT=u ERR=stno MODE=m)f$

All options are defined in this chapter.

Multi-user

HAI*Basic supports multi-user access on both file level and record level. File access can be exclusive or shared. Individual records can be locked in case of shared file access.

See the I/O statement definitions in this chapter and also the chapter on file management.

Peripheral drivers

Peripheral hardware can be controlled by OPENing a driver instead of a file.

```
Example:    3010   OPEN (1)"$LPT"
                   ...
            3050   PRINT (1)"Printer output"
                   ...
            3100   CLOSE (1)
```

See the chapter on drivers for all details.

2.4. BASIC EDITOR

The basic source program editor allows to key in and to update basic programs. The commands are defined later on in this chapter (LOAD, SAVE, LIST, etc.). There are however a few general points to be described here.

A statement can be added by keying in the statement preceded by its statement number and followed by RETURN.
It replaces a possible already existing statement with the same number.
Keying in only a statement number deletes that statement.

The editor handled encoded compressed basic source files and ascii files.
Ascii files are used to merge pieces of basci code.

2.5. SYNTAX CONVENTIONS

The remainder of this chapter defines all language features in alphabetical order.

The definitions are presented by its format(s), explanatory text and and example.

Although you may type keywords and variables in upper or lower case, this manual follows a typographical conventions.

Keywords

HAI*Basic keywords are in uppercase characters.

If an option keyword is followed by a = sign then the = sign is part of the keyword.
If a function keyword is followed by a parenthesis (then the (is part of the keyword.

Examples: ERR=5000 (ERR= is different from ERR)
MOD(A,10)

Expressions

Expressions are presented in their simplest form: a scalar variable of type numeric or string as appropriate. The use of lowercase characters indicates that the variable represents an expression.

Examples: n
f$

Variables are in uppercase characters and they may have embedded subscript expressions.

Examples: Y
A$(I+2,J)

Lists

A list of variables in an I/O statement is named a var_list.

Example: A B$[3,4] C(A+1,T)

A list of expressions in a DATA statement is named a expr_list.

Example: H F*4+1 F$ "XYZ"

A list of expressions in a PRINT statement is named a print_list.

Example: PRINT A$,B,C:"##0"

Commas

Commas in a print_list are equivalent to the function HT (horizontal tab). Commas between the variables in a var_list or between the expressions in an expr_list are superfluous. The meaning of the list is unambiguous anyway.

Statement numbers

The statement numbers (in the range of 1 to 64999) are represented by **stno**.

Example: GOTO stno

2.6. SUBROUTINES, ROUTINES AND FUNCTIONS

Subroutines, Routines and Functions are used in HAI*BAS to produce reusable code and break programs into smaller modules.

2.6.1. HAI*BAS program environment

The environment in which a HAI*BAS program runs consists of 3 parts:

1. Data (including HELP file).
2. Code.
3. Files.

In release 4 there can only be 1 instance of these environments. CHAIN can run a new code module but it does not return. Similarly, CLEAR wipes out data without any chance of recovery and CLOSE does the same for file access.

Release 5 provides a stack mechanism that allows local environments to be created by a child and restores the parent environment on exit. The decision to create a local environment is taken by the child; this means it is possible to remain compatible with the BASIC philosophy of allowing global access to all items (with all the inherent dangers of such an approach).

The following keywords work with this stack mechanism (they are described more fully later on):

GOSUB — The most limited statement, GOSUB can only load a new code module (such as an overlay). The parent module is restored by the RETURN statement.

Note that there is no possibility for GOSUB to create a new environment. In practical terms, this means that the stack remains part of the parent and so RESET (or BEGIN or CLEAR) will discard all GOSUB modules to restore the parent program and reset the stack. The parent is reloaded from disc, if necessary.

By convention, code that is activated by GOSUB is referred to here as a Subroutine. A Subroutine may be local (ie within a code module and identified by a statement number) or global (ie a code module itself and identified by a module name).

Subroutines are typically used for minor tasks within a program and are generally specific to that program.

NOTE that in releases before 5.87 CALL was used for Subroutine modules but RESET within the Subroutine DID NOT reset to the parent, but remained at the level of entry to the Subroutine.

Examples of Subroutine usage:

```
100 GOSUB "subr1"
200 GOSUB 2000
```

CALL

CALL has all of the features necessary to write code that is truly independant of the parent program. It can load a new code module and allow new data and file environments to be created.

Since CALL can create a new environment, the stack is considered part of the CALLed child and RESET (or BEGIN or CLEAR) will reset the stack to it's state on entry.

By convention, code that is activated by CALL is referred to here as a Routine. A Routine may be local (ie within a code module and identified by a statement number) or global (ie a code module itself and identified by a module name).

In general, a Routine starts with full access to all parent data and files. The only exceptions are variables with the same name, and files with the number, as used locally.

The decision to create a purely local environment is taken by the child using BEGIN, CLEAR or CLOSE.

CLEAR provides a local data environment. No parent variable can be directly accessed except when passed by reference in the parameter list supplied to the Routine (this, the VAR option, is explained later on).

CLOSE provides a local files environment. No parent file can be directly accessed except when passed by reference in the parameter list supplied to the Routine (the FILE option, like the VAR option, is explained later on).

BEGIN is simply a combination of CLEAR and CLOSE.

A less 'pure' Routine may elect to retain access to the parent environment but still requires some local data or files. The LOCAL keyword can prefix DIM or OPEN for this purpose, for example:

```
50      LOCAL DIM LEN=14 W LEN=1024 X$
60      LOCAL OPEN (5) "MyIndex.hix"
```

These resources automatically disappear or are closed on exit from the Routine.

Examples of Routine usage (the CALL statement is fully explained later on):

```
300     CALL "rout1"
310     CALL "rout2"
320     CALL "rout2" ()
330     CALL "rout3" (A$, 35, VARZ$[], FILE 5)
400     CALL 11000
410     CALL 12000
420     CALL 12000 ()
430     CALL 13000 (A$, 35, VARZ$[], FILE 5)
```

FUNC, FUNC$

These two keywords have all the power of the CALL statement (see above) and can be used within any valid HAI*BAS expression to call code that returns a value (numeric for FUNC, string for FUNC$).

By convention, code that is called by FUNC or FUNC$ is referred to here as a Function. A Function may be local (ie within a code module and identified by a statement number) or global (ie a code module itself and identified by a module name).

A Function can be treated as a Routine (ie activated by CALL instead of FUNC or FUNC$); the return value is then ignored. It is an error to use GOSUB for a Function.

Examples of Function usage (FUNC and FUNC$ are fully explained later on):

```
500     A = FUNC "rout1" ()
510     IF FUNC$ "rout2" () = "OK" THEN ...
520     PRINT FUNC$ "rout2" ()
530     B$ = FUNC$ "rout3" (A$, 35, VARZ$[], FILE 5)
600     A = FUNC 11000 ()
610     IF FUNC$ 12000 () = "OK" THEN ...
620     PRINT FUNC$ 12000 ()
630     B$ = FUNC$ 13000 (A$, 35, VARZ$[], FILE 5)
```

2.6.2. Parameter passing

The concept of parameter passing is essential to a full understanding of the CALL, FUNC, FUNC$ and ENTRY keywords.

Parameters are used in many computer languages to pass specific information to a general purpose routine. The purpose is to reuse the routine as much as possible by reducing it's dependencies.

A simple form of reusable code exists in virtually every HAI*BAS program: the GOSUB Subroutine. This is limited in it's reusability by a dependence on physical location (the statement number) and by dependence on global names (eg A$, file 6). Any variable or file handle the Subroutine uses is visible to the caller and great care must be taken to avoid conflicts.

Parameter passing allows access to specific details (passed by the caller) without depending on the names (the A$, file 6) of these details.

Local names are always used for the parameters passed so the code, although always appearing the same (and hence more easily reusable), will work on whatever information is supplied to it.

For example, take a small Subroutine to sort an array of index file keys:

```
100     S = 0
110     FOR I = 1 TO 99
120     REM "If adjacent elements not in order, then swap them"
122     IF COMPARE(A$[I], A$[I + 1], 5, 1) > 0 THEN S = 1 \
        W$ = A$[I] \
        A$[I] = A$[I + 1] \
        A$[I + 1] = W$
130     NEXT
140     REM "If a swap was made"
142     IF S THEN GOTO 100
190     RETURN
```

What disadvantages does this Subroutine have? Firstly, it must be placed at statement 100. In addition, it can only sort 1 array (A$[]) of a fixed size (100 elements) by using the COMPARE() function for index 1 of file 5. It also alters variables S, I and W$.

How can we change this? Firstly what does the routine do? The end effect is that it has changed the array A$[]. In order to do this it needs to know the size of A$[] and what details determine the sort order.

In other words it requires 4 parameters:

The array to be sorted.

Note that this array needs to be within the parent environment. It would be inefficient to copy it into local data for the Routine and copy the sorted result back on the return. It is better, in this case, to work directly on the data.

The size of the array.

This can be supplied as a direct value by the parent. Supplying a parameter as a value ensures that a copy is used by the child. It has 2 advantages, firstly that the parent variable is NOT affected and secondly that a constant can be supplied without first being put into a variable.

The file number.

This could also be supplied as a numeric value but this would mean that if the child issues BEGIN (or CLEAR) it can no longer access the parent file since this is not in the child's environment.

A better approach is to pass the actual file (in the same way that the actual array was passed) so that the child can work directly in this part of the parent's environment.

The index number.

Again this is a simple value (like the array size) that can be passed to the routine.

When the code is rewritten as a Routine, it appears like this:

```
10      ENTRY "ArraySort" (VAR A$[], L, FILE 1, X)
20      BEGIN
30      DIM LEN=14 S I LEN=1024 W$
100     S = 0
110     FOR I = 1 TO L - 1
120     REM "If adjacent elements not in sequence, then swap them"
122     IF COMPARE(A$[I], A$[I + 1], 1, X) > 0 THEN S = 1 \
        W$ = A$[I] \
        A$[I] = A$[I + 1] \
        A$[I + 1] = W$
130     NEXT
140     REM "If a swap was made"
142     IF S THEN GOTO 100
190     RETURN
```

We will now examine the changed statements in more detail.

10 ENTRY "ArraySort" (VAR A$[], L, FILE 1, X)

The ENTRY statement introduces a Routine (or Function) and names it ("ArraySort"). The parameter list supplies the local names for all items passed to the routine. For example, although the array is still named A$[] this can represent whatever array is passed by the caller.

By default, all parameters are passed by value. In other words L and X are set to the value of whatever the caller supplies. This ensures separation from the parent data; the Routine can freely use them as work variables without any impact on the callers environment.

The keywords VAR and FILE allow parameters to be passed by reference.

VAR A$[] is a reference to the actual array that is supplied by the caller. Any changes made in A$[] work directly in the parent's data environment and so are available to the caller on return from the Routine.

FILE 1 is a reference to whatever file number is supplied. Again, the Routine works with the actual file. Any action on that file (including CLOSE) can affect the caller's file environment.

When the Routine is called, the parameter list that is supplied must match that in the ENTRY statement, as described later on.

20 BEGIN

BEGIN cuts off the Routine from direct access to parent data and files. This is generally good programming practice as it produces more reliable code that is easier to maintain.

All parameters, including those passed by reference (VAR A$[] and FILE 1, in this case), remain available after the BEGIN (or CLEAR or CLOSE).

30 DIM LEN=14 S I LEN=1024 W$

Since the BEGIN statement was used there is no danger that this DIM will confict with any variables used by the parent; the child has it's own local data environment.

110 FOR I = 1 TO L - 1

Here the supplied length (L - 1) replaces the constant value 99, allowing arrays of any size to be sorted.

Note that it is not, in fact, necessary to pass the size of a complete array dimension since the system function DIMOF() can be used instead. This is explained later; passing the length as a parameter is quite valid for the purposes of this example. It also more flexible, allowing part of a dimension to be sorted.

*122 IF COMPARE(A$[I], A$[I + 1], 1, X) > 0 THEN S = 1 *

Here the constant values 5 and 1 are replaced by FILE 1 and X respectively. The file number 1 is the local number for whatever file was passed to the function, just as

A$[] is the local name for whatever array variable is passed. X is the local name for the value of the index number; it is not the number itself and so cannot affect the representation of that number in the parent's environment.

Note that if the file number was passed as a value the BEGIN at statement 20 would ensure an error here when trying to access the file. Passing by FILE reference is the correct solution, NOT removing the BEGIN!

Now, how is the Routine used? Since it is a Routine (not returning a value), the CALL statement must be used. The CALL must also supply a parameter list to tell the routine which items it is to work on. For example, to sort 12 items in the M$[] array using index 2 in file 53:

```
CALL "ASORT" (VAR M$[], 12, FILE 53, 2)
```

The array sort Routine still uses the same local names so after this CALL the parameters L and X will contain 12 and 2 respectively. For parameters passed by reference, VAR A$[] now refers to the M$[] array and FILE 1 to the callers file number 53.

2.6.3. Parameter list

Purpose Passing parameters to a Routine or Function.

Format (parameter_list)

Remarks The parameter list ties together the items supplied by the caller (by CALL, FUNC or FUNC$) and the local names used in the ENTRY statement of the Routine or Function.

The parameters must match, both in quantity and type. The precise matching requirements are described later.

The parameter list can pass information in the following forms:

Value By default all items are passed by value. Any HAI*BAS expression is valid; the result is stored in a local variable within the Routine (see the ENTRY statement).

Value passing provides a safe, one way, transfer of data to the Routine. Any variable passed by value cannot be modified by the routine once it has cut itself off from the parent data by BEGIN or CLEAR.

An example is:

```
10      CALL 110 ("HAI*BAS 5")
110     ENTRY "Name" (A$)
```

Variable reference

The VAR keyword specifies that the following variable is to be passed by reference. The variable may be freely used by the Routine (using its local name, as described for ENTRY) and any value assigned into it will be there on return to the parent.

Scalar variables, arrays, subdimensions of arrays, array elements, substrings and system variables can all be passed by reference.

Variable references provide a means for the routine to update parent data without the rigid (and error prone) access to all global data. A general purpose array sort can be written or a module to provide CUA compatible screen access for the majority of applications.

An example is:

```
20      CALL 200 (VAR A$, VAR B$(11,10), VAR C$[])
120     ENTRY "Example" (VAR X$, VAR Y$, VAR C$[])
```

NOTE that when a substring is passed by reference all assignments to it CANNOT DECREASE the active length of the parent string variable. For example:

```
100     W$ = "abcdef"
110     FUNC 900 (VAR W$(3,2))
120     REM "W$ is now abXdef"
        ...
900     ENTRY "SubStr" (VAR A$)
910     A$ = "X"
990     RETURN
```

File reference

The FILE keyword is similar in concept to VAR but works for file numbers instead of variable names. It specifies that the following numeric expression gives a file number that is to be passed by reference. The file may be freely used by the Routine (using it local number, as described for ENTRY). Any actions, including CLOSE, affect the file for the parent also.

File references provide a means to write standard file handling modules. For example, a single module can be passed a command code, file number and variable references. It would then handle all I/O actions (and, possibly, variable DIMensioning) for a particular file format.

An example is:

```
30      CALL 130 (FILE 53)
130     ENTRY "FileRead" (FILE 1)
```

2.7. ALPHABETICAL LIST OF LANGUAGE FEATURES

The remainder of this chapter consists of the description of all statements, commands, functions and system variables in alphabetical order.

@ function

Purpose Positions the cursor on display of changes the current position on the printer. It is used within an ACCEPT or a PRINT statement.

Format @(c,l) or @(c) or @(,l)

c is the column number
l is the line number

Ommision of the line number implies positioning on the same line.

Ommision of the column number implies positioning in the same column.

New line on printer

It is now allowed to position backwards on a printer. That's why positioning to a column number less than the current column position implies a new line action in order to position to the specified column number on the next line.

Form feed on printer Positioning to a line number less than the current line number implies a form feed action in order to position the specified line on the next page.

Example

```
1100    PRINT @(1,14) "Amount" @(21) A
```

ABS function

Purpose Returns the absolute value of a numeric expression

Format Y=ABS(x)

Example 1100 Y=ABS(X)

ACCEPT statement

Purpose Prompts for, accepts and validates keyboard input and passes control to a specified statement number.

Format
```
ACCEPT [([IND=w] [MODE=m] [SECTOR=l] [TRACK=t])] [print_list] KEY=k [V]
                                                  [CHECK<c1]
                                                  [CHECK=c2]
                                                  [CHECK>c3]
                                                  [CHECK<=c4]
                                                  [CHECK>=c5]
                                                  [CHECK<>c6]
                                                  [EXACT=e1.e2]
                                                  [MAX=m1.m2]
                                                  [MIN=m3.m4]
                                                  [GOTO stno]
                                                  [(or GOSUB stno)]
```

Remarks The minimal format of the ACCEPT statement is:
ACCEPT KEY=k
all other elements are optional.
One expression of the expression pairs e1 and e2, m1 and m2, m3 and m4 may be omitted. You must specify the decimal point if you omit the first one.

The print_list may contain any expression that is allowed for a PRINT-to-CRT statement.

The print_list may end with a display attribute function. The attribute will then be effective for the ACCEPT input field.
Example: The BI function may be used to enter a password.

Input field The input field on the display starts at the current cursor position as left by the print_list. Or at the next **foreground** character if the character at the current cursor position has the **background** attribute.
The input field ends
- at the first background character, or
- after 250 characters from the start, or
- at the end of the display,

whichever comes first.

Default input The user enters and edits the keyboard data in the input field. The print_list or any other PRINT or ACCEPT statement in the program may fill the input field with a default value which can be edited or accepted. The first key stroke other than the special edit keys (see below) clears the input field.

Edit keys The special edit keys are:

- The Clear key (i.e. the tab or Home key on DOS systems)
- The horizontal arrows to move the cursor.
- The vertical arrows to move the cursor vertically if the input field extends to the next line on the display.
- The Home key to move the cursor to the start of the input field.
- The End key to move the cursor to the end of the input field.
- The PgUp key to move the cursor a number of positions to the right.
- The PgDn key to move the cursor a number of positions to the left.
- The Del key removes the current character from the input field.
- The Ins key put the input system in insert mode until pressing the Ins key again.

System dependant

The edit keys are named according to the implementation on the IBM PC keyboard and with PCDOS. It may be different on other hardware and/or other host operating systems. See the system parameter file HAI.PAR for the keyboard parameterization.

Validation The keyboard input is only accepted if all validations are successful. Incorrect input generates a bleep (if in effect, see parameter file HAI.PAR) and the program waits for valid keyboard input.

IND w specifies the window number for the accept. 0 is the current parent, 1 to 999 are children of the current parent. Values above 1000, such as obtained by GET(SECTOR=5), can be used for specific user window numbers. If there is no IND= then the current window is assumed.

MODE The MODE= option can be used for several purposes, m specifies:

+1 Prevents clearing the ACCEPT screen area if the 1st key is pressed is printable.

+2 Restrict input area to within a single window row (redundant, see SECTOR=)

+8 Prevents conversion of single character input to upper case.

SECTOR Is used to specify the maximum length. This is still terminated by a background character but is otherwise only limited by the window size.

TRACK Screen window area. 1 is the top, 2 (or 0) is the panel area and 3 is the bottom. If there is no TRACK= or if TRACK=0 is used then the panel area is assumed.

TRACK= was introduced with release 5.60. IND= sets the default for further actions; TRACK= does not.

KEY Function key with number k is required.

The keyboard input value is assigned to the variable V if all validations are successful.
The receiving variable has either numeric or string type, which implies a data type (and length) validation. Omission of V implies function key input without preceding keyboard data.

CHECK The CHECK option specifies allowed input values. The expression c1 must be in the same data type as receiving variable V. Every sensible combination of CHECK options is allowed.

Single character
A single lower case input character is converted to upper case. This feature is especially useful for Y(es)/N(o) answers (see MODE= option).

EXACT The expressions e1 and e2 in the EXACT option specify the exact number of decimal digits required before and/or after the decimal point. It specifies the exact number of characters for character string input.

MAX The expressions m1 and m2 in the MAX option specify the maximum number of decimal digits required before and/or after the decimal point.
The number of decimals after the point of the input value may be less than the number specified in the MAX option. In this case the input value is scaled accordingly before assignment to the receiving variable V (see the example below).
The position of the MAX within the option list may influence the result of the ACCEPT statement since the options are evaluated from left to right.
The MAX option specifies the maximum number of characters for character string input.

MIN The expressions m3 and m4 in the MIN option specify the minimum number of decimal digits required before and/or after the decimal point.
It specifies the minimum number of characters for character string input.

IntegerThe decimal point is for input purposed only. Keep in mind that the input value is **always** stored as an integer value.

GOTO/GOSUB
The GOTO option specifies the statement number to pass control to if all validations are successful.
This option may also by a GOSUB option, which is only useful if the subroutine must return to the statement following the ACCEPT statement (See also 'ACCEPT groups' below).
Control is passed to the next statement after successful validation if no GOTO or GOSUB option is specified.

ACCEPT groups

ACCEPT statements can be grouped together by placing them consecutively in the program. Two groups must be separated by at least on other statement (possibly a REM statement).
Only the print_list of the first ACCEPT statement of a group is executed.
The group is scanned for the first ACCEPT statement that validates successfully and the corresponding GOTO or GOSUB option is executed. The keyboard input is not accepted if no ACCEPT statement validates successfully.

Help

The system variable HELP must contain the correct help text number before executing the ACCEPT statement. See the chapter on HELP.

Single key input

The ACCEPT statement requires input followed by a function key. Single key input is possible by explicit access to the $CRT driver (see the chapter on drivers).

Example

```
3010   ACCEPT @(10,10) SB "Amount ("SF"     "SB")"
                       @(18) KEY=2 D$ GOTO 2000
3020   ACCEPT KEY=1 A MAX=3.2 CHECK>=10
                                    CHECK<=50000
```

The six character input field is terminated by a background character. The prompt text also has the background attribute. This way the contents of a 'form-on-the-screen' can been cleared by a single PRINT CF statement.

The dummy string variable D$ must have a DIMmed length of at least six characters. Any input terminated by function key 2 causes the program to go back to statement 2000.

Input	Variable A
1.23	123
1.2	120
1	100
.2	20

This scaling is performed before the CHECKs (left-to-right rule).

ASC function

Purpose Returns the result of a string expression as a numeric type value.

Format Y=ASC(x$) or Y=ASC(x$,n)

Remarks The internal representation of x$ is considered as a numeric value without any further conversion.

The length value n ranges between 1 and 6 characters (i.e. bytes). Omission of n implies a length of 1 byte.

The current length of expression result x$ must be at least equal to n. If the current length of x$ is more than n, the left most n characters are taken.

The function ASC is different from function VAL which is a real conversion from external ascii to internal binary representation.

Special case If C$ is equal to CHR(255), than CHR(C$) yields 255 and CHR(C$,1) yields -1.

Example

```
1100  A=ASC(B$,3)
```

BB function

Purpose Activates the blinking display attribute. It is used within an ACCEPT or PRINT statement.

Format BB

Remarks The function name BB has its original meaning for monochrome displays. It is however possible to associate any attribute control sequence with the BB function.
See parameter file HAI.PAR for the color display specification.

See the PRINT statement for general properties of display attribute functions.

Example
```
1100  PRINT @(10,10) BB "Blinking characters"
```

BD function

Purpose Activates the dimmed display attribute. It is used within an ACCEPT or PRINT statement.

Format BD

Remarks The function name BD has its original meaning for monochrome displays. It is however possible to associate any attribute control sequence with the BD function.
See parameter file HAI.PAR for the color display specification.

See the PRINT statement for general properties of display attribute functions.

Example

```
1100    PRINT @(10,10) BD "Dimmed characters"
```

BEGIN statement

Purpose Closes all open files,
removes all user variables,
sets the default length of numeric variables to 14 digits,
sets the default length of string variables to 250 characters,
restores the DATA pointer and
empties the return address stack.

Format BEGIN

Remarks BEGIN normally is the first executable statement of a program.

If the program does not start with begin, all DIMmed variables and OPENed files from the previously running program are still available.

Additional variables may be DIMmed and more files may be OPENed. See also the CHAIN statement.

The return address stack contains the nesting information of GOSUB/RETURN, FOR/NEXT and STOP/GO statement pairs.

Example

```
10      BEGIN
```

BI function

Purpose Activates the invisible display attribute. It is used within an ACCEPT statement to enter a password.

Format BI

Remarks See the PRINT statement for general properties of display attribute functions.

Example

```
1100    ACCEPT @(10,10)"Password (      "SB")"
                       @(20) BI KEY=1 P$
```

BM function

Purpose Activates the BM display attribute. It is used within an ACCEPT or PRINT statement.

Format BD

Remarks The function name BM does not have a specific meaning anymore. It is possible to associate any attribute control sequence with the BM function.
See parameter file HAI.PAR for the color display specification.

See the PRINT statement for general properties of display attribute functions.

Example

```
1100  PRINT @(10,10) BM "Characters with BM attr."
```

BS function

Purpose Moves the cursor one position backwards on a display. It is used within an ACCEPT or PRINT statement.

Format BS

Example

```
1100    ACCEPT @(10,10)"Number 0-9"(  " SB")"
                    BS BS KEY=1 N
```

BU function

Purpose Activates the underline display attribute. It is used within an ACCEPT or PRINT statement.

Format BU

Remarks The function name BU has its original meaning for monochrome displays. It is however possible to associate any attribute control sequence with the BU function.
See parameter file HAI.PAR for the color display specification.

See the PRINT statement for general properties of display attribute functions.

Example
```
1100  PRINT @(10,10) BU "Underlined characters"
```

BV function

Purpose Activates the reversed video display attribute. It is used within an ACCEPT or PRINT statement.

Format BV

Remarks The function name BV has its original meaning for monochrome displays. It is however possible to associate any attribute control sequence with the BV function.
See parameter file HAI.PAR for the color display specification.

See the PRINT statement for general properties of display attribute functions.

Example
```
1100  PRINT @(10,10) BV "Char. in reversed video"
```

CALL statement

Purpose Call a global or local Routine.

Format CALL [(i/o clauses)] name_exp | stno [(parameter_list)]

i/o clauses

The standard i/o clauses may be applied immediately after the CALL. No immediate use is seen for this except compatibility with the past and options for the future.

name_exp | *stno*

For a global Routine name_exp is a string expression that defines the HAI*BAS program module that is to be loaded.

Alternatively, stno is the statement number of a local Routine.

parameter_list

The parameter list (described above) determines what is to be passed to the Routine.

If nothing is passed the brackets are optional (unlike FUNC and FUNC$, see below). In this case the ENTRY statement is also optional at the start of the Routine.

Remarks If an non-empty parameter_list is supplied, the first executable statement in the Routine, after any REMarks, must be ENTRY.

Examples

```
100   CALL "ASORT" (VAR A$[], 12, FILE 53, 2)
110   CALL 1000
120   CALL 2000 (W$)
```

CF function

Purpose Clears only the characters with the (default) foreground attribute on the entire disply. It is used within an ACCEPT or PRINT statement.

Format CF

Remarks You can set up a complete form on the display with all text in background mode. All input fields are in foreground mode.
The CF function clears all input fields without affecting the background texts.

Example

```
1100  PRINT CF
```

CH function

Purpose Sets the cursor to the first position of the first line of the display. It is used within an ACCEPT or PRINT statement.

Format CH

Example

```
1100  PRINT CH "Left upper corner"
```

CHAIN statement

Purpose Loads a HAI*Basic program from from a disk(ette) file and starts its execution.

Format CHAIN (UNIT=u ERR=stno)f$

Remarks The CHAIN statement is the only way to load and start a compiled HAI*Basic program.
The CHAIN statement acts as a combined LOAD/RUN command in source interpreter mode.

The string expression f$ must yield a five character program file name. The source interpreter assumes a sixth character B and the compiled code interpreter assumes a sixth character C.

UNIT The program file f$ must be present on unit u. If the UNIT option is not specified, the file f$ is searched for on all logical units. See the OPEN statement for the search order.

ERR The ERR option specifies the start of the error handling routine. See the chapter on error codes for detailed information.

Variablses / Open files The program executing the CHAIN statement is overloaded by the program from file f$.
The DIMmed variables and the files left OPEN from the previously executed program are still available. Most programs however start with a BEGIN statement, throwing away all DIMmed variables and closing all OPEN files.
It is good practice in most cases to CLEAR the varaibles and to CLOSE all files before CHAINing to the next program. The END statement closes all OPEN files before returning the menu program PMENU.

Compiler If a program is broken up into several sections using the same variables or OPEN files, special care must be taken when compiling these programs. See the chapter on utilities.

Example

```
2200   CHAIN "NEXTP"
```

CHR function

Purpose Returns the result of a numeric expression as a string type value.

Format Y$=CHR(x) or Y$=CHR(x,n)

Remarks The internal representation of an expression result x is considered as a string value without any further conversion.

The length value n specifies the byte length of the internal representation of x ranging from 1 to 6 bytes. Omission of n implies a length of 1 byte.

The length of the internal representation of x must be at least equal to n. If the length of x is more than n, the leftmost n characters are taken.

Example C$=CHR(3)

CLEAR statement

Purpose Removes all user variables,
sets the default length of numeric to 14 digits,
sets the default length of string variables to 250 bytes,
restores the DATA pointer and
empties the return address stack.

Format CLEAR

Remarks The CLEAR actions are a subset of the BEGIN actions.

When a small program using very much space for DIMmed variables CHAINs to a large program, the large program may not fit in the HAI*Basic user space. The variables of the previous program would only be thrown away by the BEGIN of the next program, if the next program could be loaded at all. The solution is to execute a CLEAR in the previous program before CHAINing to the next program.

The return address stack contains the nesting information of GOSUB/RETURN, FOR/NEXT and STOP/GO statement pairs.

Example

```
1000  CLEAR
```

CLOSE statement

Purpose Ends I/O on a file for a peripheral driver.

Format CLOSE (n ERR=stno) or CLOSE

Remarks The simple form CLOSE closes all drivers and files that are currently open.

The file number n is available for a subsequent OPEN after CLOSEing the file.

ERR The ERR option specifies the start of the error handling routine. See the chapter on error codes for detailed information.

The contents of a file is not garantueed to be correct it the file has not been properly CLOSEd. You must not remove diskettes or removable disks before CLOSEing all their files.
The file is marked as 'updated' when writing to the file. If such a file is not properly CLOSEd, further access to the file is not possible.
Although it is possible to reset the marker in the file header block, the file contents is not garantueed to be correct.

Multi-user The CLOSE statement makes the file available for exclusive access by another user, unless there are still other users having shared access to the same file.

Example 7000 CLOSE(3)

COLUMN function

Purpose Returns the current character column position of PRINT output to a file or driver.

Format C=COLUMN(n)

Remarks n is the number of the OPENed device or file.

Function COLUMN returns 0 if the file number n is not in use.

$CRT The display driver is implicitly OPENed with device number 0. So COLUMN(0) returns the current column position on the display.

$DLK The $DLK driver uses COLUMN in a different way. See the chapter on drivers.

Example

```
1300  IF COLUMN(0)+LEN(W$) > 80 THEN PRINT
```

COMMON statement

Purpose Modifies the effect of subsequent statement(s).

Format
COMMON numexp
COMMON numexp stmt
COMMON [numexp] specialstmt

Remarks

numexp	Numeric expression for common area handle.
stmt	Any HAI*BAS statement, except the "special" ones.
specialstmt	A HAI*BAS statement which has a special meaning when used as part of the COMMON statement.

There are 3 forms of the COMMON statement.

The first has a handle number but nothing else. This establishes a default common area for ALL subsequent statements until COMMON [numexp] END (see below). These statements are said to be in a "COMMON program block".

The second also has a handle number; this is followed by (almost) any HAI*BAS statement. A default common area is established for the duration of the single statement only. This can be outside or inside a COMMON program block; in the latter case the block default reasserts itself after the single statement.

The third form is for statements that have a special meaning when used as part of a COMMON statement. These are described separately below and include END and CLEAR; others (such as LOAD and SAVE) may be added later.

Example

```
10 COMMON H DIM LEN=14 A LEN=250 B$ C$ LEN=1024 D$
50 COMMON H
60 PRINT A B$ C$ D$
70 COMMON .H END
```

Note the use of .H in statement 70; the unary "." operator overrides the default area for the COMMON program block.

COMMON() function

Purpose Creates a COMMON variables area.

Format COMMON ([numexp])

Remarks numexp Optional numeric expression for size of common variables area.

A common variables area is created and a handle returned for further access to the area.

The optional numexp should be a guide to the final size of the area. The area will expand automatically, up to a system defined limit, but it is normally more efficient to allocate the correct size initially.

The size used can be determined in the BASIC interpreter, after all DIMensioning is complete, by SHOW COMMON.

Example 10 H = COMMON(2500)

COMMON area members

Purpose References members of a COMMON area

Format numvar.member

Remarks

numvar	Numeric variable containing COMMON area handle.
member	DIMensioned name of COMMON area member.

A variable handle can ONLY be a numeric variable (scalar, array or COMMON member). Expressions and numeric constants are not permitted. For example:

H.A$, H[A + 27].B$, H.X.C$

are correct, but:

1.A$, VAL(H$).A$

are invalid.

Within a COMMON program block or statement where a default COMMON handle H is established, it is as if EVERY variable has an H. prefix. For example, the following 3 statements are equivalent:

```
10 PRINT H.A$
20 COMMON H PRINT A$
30 COMMON H \ PRINT A$ \ COMMON .H END
```

COMMON program blocks are generally retained thoughout all HAI*BAS statements except BEGIN and CLEAR.

A COMMON program block is considered to be part of the program environment and so will be restored on return from a Routine or Function (ie the return after a CALL, FUNC () or FUNC$ ()). For example:

```
10 COMMON H
20 CALL 1000
30 PRINT A$
...
1000 BEGIN
```

causes statement 30 to print H.A$.

The unary "." operator overrides a COMMON program block to force access to variables in the normal data area. For example:

40 COMMON H \ PRINT .A$ \ COMMON .H END

will print the normal variable A$, not H.A$.

Limitation 100 common data areas are allowed.

Bugs DIMensioning common area members must be done in a COMMON program block or statement.

DIM H.A$

is a syntax error. The correct form is either of these:

10 COMMON H DIM A$
20 COMMON H \ DIM A$ \ COMMON .H END

COMMON CLEAR statement

Purpose Clears a COMMON variables area.

Format COMMON numexp CLEAR

Remarks numexp Numeric expression for common area handle.

The COMMON area is removed. All data and DIMensioned variables are removed. The handle is freed for eventual reuse.

Note that when a handle is freed it will be the LAST to be reused. This is intended to reduce the danger that a HAI*BAS program may accidentally use a handle number that has been cleared. Consider the following program fragment:

```
10 H = COMMON()
20 COMMON H DIM LEN=14 A
30 COMMON H CLEAR
40 H2 = COMMON()
50 COMMON H2 DIM LEN=2 A
60 H.A = 123456789
```

Statement 60 is clearly incorrect; it should give an error because the handle H is not reused for H2 until all other free handles have been used.

COMMON END statement

Purpose Terminates a COMMON program block

Format COMMON [numexp] END

Remarks numexp Optional numeric expression for common area handle.

The COMMON program block is terminated and normal variable addressing resumes.

If numexp is specified then it must match the current default handle; this should improve program reliability. When numexp is omitted there is no such check.

COMPARE() function

Purpose Compares 2 numbers or strings

Format COMPARE (e1, e2, ne1, ne2)

Remarks

e1 is a string (or numeric) expression.
e2 is another expression of the same type.
ne1 is an optional numeric expression for a file number.
ne2 is optional after ne1 for the index number (TRACK=) of the file.

This compares 2 expressions (e1 and e2). The expressions may be for numeric or string values but must be the same type.

The default (no ne1) uses the same comparison as the relational operators "<" (less than) "=" (equal to) and ">" (greater than).

If ne1 is supplied but is zero, the default weighting table is used to convert string values before comparing; they are treated as a type 3 (weighted, punctuation not stripped) part of an index key. Numeric values are converted as a type 1 (numeric) part of an index key.

If ne1 is supplied and represents an open HAI*Basic file, then the comparison is done as if both values are keys for the primary index for that file. This may or may not involve weighting; the method is determined by the file organisation.

If ne2 is supplied in addition to ne2 then this selects a specific index number in the file which supplies the compare method.

If ne1 is nonzero but does not represent an open HAI*Basic index file or if ne2 is specified and does not represent and index for that file a HAI*Basic error is generated.

When the file number is 0 the (optional) 4th parameter can specify an index part type to be used for the comparison. This allows, for example, a stripped weighted comparison (see the example on statement number 60).

Return values

+1 if e1 is greater then e2
0 if e1 is equal to e2
-1 if e1 is less then e2

Example

```
20  A = COMPARE (A$ B$)
30  ON COMPARE (A, 0) GOTO 100, 200, 300
40  IF COMPARE (A$, B$, F3) <> 0 THEN PRINT "Different"
50  IF COMPARE (A$, B$, F3, 2) > 0 THEN PRINT "Greater"
60  A = COMPARE (A$, B$, 0 ,4)
```

COMPAREMASK() function

Purpose Compares 2 strings, allowing "wild card" mask characters.

Format COMPAREMASK (se1, se2, ne1, ne2)

Remarks

se1 is a string expression.
se2 is string expression for a mask, with '?' and '*' wild card characters.
ne1 is an optional numeric expression for a file number.
ne2 is optional after ne1 for the index number (TRACK=) of the file.

This compares a string expression (se1) with a mask (se2)

The default (no ne1) uses the same comparison as the relational operators "<" (less than) "=" (equal to) and ">" (greater than) but the wild card mask characters are recognised:

'?' matches any single se1 character.
'*' matches 0 or more characters in se1 until the first one the matches the se2 character after the '*'.
If there is no further character after the '*' then this matches the rest of se1.

Note that tests for less of greater than may appear misleading when mask characters are involved.

If ne1 is supplied but is zero, the default weighting table is used to convert string values before comparing; they are treated as a type 3 (weighted, punctuation not stripped) part of an index key.

If ne1 is supplied and represents an open HAI*Basic file, then the comparison is done as if both values are keys for the primary index for that file, but still allowing wild card characters in the mask.

If ne2 is supplied in addition to ne2 then this selects a specific index number in the file, instead of using the primary.

If ne1 is nonzero but does not represent an open HAI*Basic index file or if ne2 is specified and does not represent and index for that file a HAI*Basic error is generated.

Return values

+1 if e1 is greater then e2
0 if e1 is equal to e2
-1 if e1 is less then e2

Example

```
20    A = COMPAREMASK (A$ M$)
30    IF COMPAREMASK (A$, "HAI*BAS") =0 THEN REM "Match" \
      PRINT "Holland Automation"
40    IF COMPAREMASK (A$, M$, F4) THEN PRINT "No Match"
50    IF COMPAREMASK (A$, M$, F3, 2) THEN PRINT "Match"
```

AB?	matches	ABA
		ABX
		AB9
AB*	matches	AB
		ABX
		ABCDEFGHIJ
A?B	matches	AAB
		AXB
		A9B
HAI*BAS	matches	HAI_Holland_Automation_International_BAS
		HAIBAS
		HAIxBAS

CONTROL function (file)

Purpose Performs several functions on files.

Format CONTROL (f MODE=m)

Remarks f represents the file number of the file on which the function has to be performed.

MODE=9 Flush a file. A flush is only done if the file has been written by the current user since the last flush (or since the file was opened if there was no previous flush).

Example

```
1000  CONTROL (34 MODE=9)
```

CONTROL function (screen)

Purpose Performs several functions on screen.

Format CONTROL ([IND=w] MODE=m)

Remarks w is the number of the window to which the CONTROL function applies.

MODE=1 Clears the keyboard.

MODE=2 Disables window w and all children. They are removed from the screen and the underlying window(s) are restored. Output (and ACCEPT) can still use disabled windows but nothing is seen.

Note that when running a program from the editor, the ouput may not be seen until an ACCEPT statement is reached (which automatically disables the editor window). This disable can be forced before running by:

```
CONTROL (IND=50000 MODE=2) \ RUN
```

MODE=3 Enables window w and all children. They appear on the screen with the same priority as when disabled.

If they are fully covered by higher priority windows this statement has no visible effect.

MODE=4 Forces window w to be fully visible. It is moved to the top of the priority stack. If the window was disabled then this statement automatically enables it.

Any children of window w are not included - in fact, since they must lie within the bound of their parent and the parent has become fully visible they will be hidden.

Example CONTROL (IND=2 MODE=2)

CR function

Purpose Moves the cursor to the first position of the current line on a display, or performs a carriage return on a printer.

It is used within an ACCEPT or PRINT statement.

Format CR

Example
```
1400  PRINT CR "Left hand side"
```

CS function

Purpose Clears the entire display. It is used within an ACCEPT or PRINT statement.

Format CS

Remarks The size of the roll-up area is reset to the entire display (See the SB function).

Example

```
20 PRINT CS @(15) "Heading"
```

DATA statement

Purpose Specifies input data for the READ DATA statement.

Format DATA expr_list

Remarks All DATA statements in the program constitute one stream of input data for the READ statements, regardless of their position within the HAI*Basic program.

See the READ statement for more information.

Example

```
9000  DATA 1,34,78,"XYZ",B*100
```

DAY system variable

Purpose Holds the six digit date.

Format DAY

Remarks The DAY value may be different from the date in the host operating system (see also the $HOST driver).

It is normally set in the START program and used in the other program.

Example

```
4000  PRINT DAY:"00.00.00"
```

DELETE statement (record)

Purpose Removes a record from an indexed file.

Format DELETE (n IND=i$ ERR=stno)

n is the number of the OPENed indexed file.

i$ is the index of the record to be DELETEd.

Remarks The record with index i$ is removed from the file releasing its space.

Error 2 In previous HAI*Basic implementations the record was only flagged as being DELETEd. It was removed when reorganising the indexed file. The old error 2 (when accessing a record with the DELETE flag) is not generated by the current implementation anymore.

Multi-user The record to be deleted must not by locked by another user in case of shared file access.

See error code 20.

Example

```
1200   DELETE (n IND="ABCDE" ERR=5000)
```

DELETE statement (index)

Purpose Deletion of a (shared) index

Format DELETE (n MODE=99 TRACK=t)

Remarks n is the number of the OPENed index file.

t is the number of the index to be removed.

It is allowed to delete an index when there is already data present in a file.

It is not possible to delete a single logical index that shares a physical index with other logical indexes. Whenever a shared index is removed than de physical and ALL sharers are also removed.

Example 1010 DELETE (10 MODE=99 TRACK=3)

DIM statement

Purpose Defines variables and reserves memory for these variables.

Format DIM LEN=l1 var_list LEN=l2 var_list ...

Remarks No storage is reserved until execution of a DIM statement.

LEN option The length expressions l1 and l2 must yield a value in the range 1 to 1024. A numeric variable will never have more than 14 digits, even if the current LEN value is more than 14.

The default length for numeric variables is 14 decimal digits. The default length for string variables is 250 characters.

The LEN option remains effective until the next LEN option in the same or in another DIM statement.

Scalar variables are implicitly DIMmed when appearing at the lefthand side of a LET statement. Their length is according to the last LEN option executed in a DIM statement.

Variable names

See the overview at the start of this chapter.

Maximum size

The number and size of an array dimensions are only limited by the size of the HAI*Basic user area (See also the TXTSIZ parameter in file HAI.PAR).

The array dimensions may be defined by a numeric expression (See also the FREE function).

The first element of an array has subscripts (1), (1,1), (1,1,1) etc.

Memory requirements

The following expressions define the total memory requirement for variables:

Numeric scalar variable	:	n+1
Numeric array	:	d1*d2*...*n + d*2 + 2
String scalar variable	:	s+3
String array	:	d1*d2*...*(s+1) + d*2 + 3

d	is the number of dimensions
d1	is the first dimension
d2	is the second dimension, etc.
n	is the number of bytes according to the DIMmed value (see table).
s	is the DIMmed string length.

Numeric data is represented in 1 to 6 bytes:

DIMmed length	Bytes	Maximum value
1	1	127
2	1	127
3	2	32 767
4	2	32 767
5	3	8 388 607
6	3	8 388 607
7	4	2 147 483 647
8	4	2 147 483 647
9	4	2 147 483 647
10	5	549 755 813 887
11	5	549 755 813 887
12	6	140 737 488 355 327
13	6	140 737 488 355 327
14	6	140 737 488 355 327

Record fields

It is strongly recommended to define the variable length explicitly since in most cases the default or the current LEN value is not correct.
This is even more important for variables used in the var_list of an I/O statement. The record fields are implicitly defined by the DIMmed length of the variables.

The byte length of numeric variables is relevant when using them in the var_list of an I/O statement.
String variables are padded with value 3 characters up to the DIMmed length when used in the var_list of a WRITE or INSERT statement. The current length byte is not written to the file record.
The trailing value 3 characters are removed when reading the record value into a string variable. Special care is required when using strings with pure binary bit patterns as an index for indexed files (See the READ statement for more information).

Initial value Numeric variables have initial value 0. String variables have initial value "" (i.e. empty string). The same applies to all elements of a numeric or string array.

Examples

```
210     DIM LEN=3 A B4 LEN=5 C D$
220     DIM LEN=7 E6(A*2) LEN=2 F$[10,30]
```

DIMOF() function

Purpose Provides the number of dimensions in an array variable.
Provides the size of the specified array dimension.

Format DIMOF(array_name, [num_exp])

The array name is normally without any element details (eg A$[]). Part of a multidimensional array is allowed but has little purpose. A scalar variable is NOT allowed.

The numeric expression specifies which dimension size is to be supplied.

Remarks This function allows code to be written that does not rely on constant values. The array name can be one passed by VAR reference to a Function or Routine.

Examples

```
10      DIM A[7,8,9]
20      REM "Print 3"
22      PRINT DIMOF(A[])
30      REM "Print 2"
32      PRINT DIMOF(A[1])
40      REM "Error - not an array but an element of an array"
42      PRINT DIMOF(A[1,1,1])

100     DIM A[7,8,9]
200     REM "Print 7"
220     PRINT DIMOF(A[],1)
300     REM "Print 8"
320     PRINT DIMOF(A[],2)
400     REM "Print 8"
420     PRINT DIMOF(A[1],1)
```

DISABLE statement

Purpose Disconnects a logical volume from the HAI*Basic program.

Format DISABLE (UNIT=u ERR=stno)

Remarks DISABLE has no effect in recent implementations, except for checking OPENed files on unit u (see error code 29).

Example 8000 DISABLE (UNIT=2 ERR=9000)

EB function

Purpose Inactivates the blinking display attribute. It is used within an ACCEPT or PRINT statement.

Format EB

Remarks See the PRINT statement for general properties of display attribute functions.

Example

```
1100  PRINT BB "Blinking" EB "Not blinking"
```

ED function

Purpose Inactivates the dimmed display attribute. It is used within an ACCEPT or PRINT statement.

Format ED

Remarks See the PRINT statement for general properties of display attribute functions.

Example

```
1100  PRINT BD "Dimmed" ED "Not dimmed"
```

EDIT command

Purpose Allows a statement to be edited on the display.

Format EDIT stno

Remarks The screen is cleared and the statement is displayed at the top of the screen.

See the ACCEPT statement for the available edit keys.

The EDIT command is equivalent to a statementnumber by a function key not equal to F1 or F2.

Example EDIT 300

EI function

Purpose Inactivates the invisible display attribute. It is used within an ACCEPT or PRINT statement.

Format EI

Remarks See the PRINT statement for general properties of display attribute functions.

EM function

Purpose Inactivates the EM display attribute. It is used within an ACCEPT or PRINT statement.

Format EM

Remarks See the PRINT statement for general properties of display attribute functions.

Example

```
1200  PRINT BM "Attr. ON" EM "Attr. OFF"
```

ENABLE statement

Purpose Connects a logical unit to the HAI*Basic program.

Format ENABLE (UNIT=u ERR=stno)var_list

Remarks The only effect of ENABLE in recent HAI*Basic implementations is reading the volume label information.

The following information is assigned to the variables of var_list:

bytes 1 to 4	:	HOST
bytes 4 to 30	:	Volume label from host O.S. padded with zeros if necessary.
byte 31	:	S

Example

```
1000  ENABLE (UNIT=2 ERR=1500) V$
```

END statement

Purpose Ends a program and returns to the program menu.

Format END

Remarks All OPEN files are closed before finishing the program.

The HAI*Basic compiled code interpreter assumes that the program selection program is named PMENUC.

The source interpreter returns to edit mode. The program PMENUB is searched by executing the END command (i.e. without a statement number).

Example 3000 END

ENTRY statement

Purpose Names a Function or Routine, names the parameters and specifies whether they are optional.

Format ENTRY "name" (parameter_list)

name The Function or Routine name must be specified as a string constant. This name may be used in the future to provide a library of Functions or Routines.

The name must be a string constant from 1 to 31 characters and contain only A-Z, a-z, 0-9, _ (underline) and $ (dollar) characters. It must start with a capital A-Z. All names must be unique.

Naming conventions should capitalise or otherwise identify the startof each word within the name, for example:

```
ENTRY "ArraySort" (VAR A$[], L, FILE 1, X)
```

parameter_list

The parameter list gives local names or numbers to the items passed by CALL, FUNC or FUNC$. If nothing is passed the brackets are still needed, as ().

Optional arguments are enclosed fully in square brackets. See File and X in example 13.

Remarks ENTRY must be the first executable statement in the Function, after any REMarks.

If ENTRY is executed at any other time, error 92 (illegal statement) is given.

The types of ENTRY parameters must match those supplied by FUNC or FUNC$ according to the rules described below.

If there is a mismatch, error 62 (type mismatch) is given.

ENTRY array references can be of unspecified dimensions in ENTRY (eg A$[]) and will match whatever array is supplied. If the DIMensions are given in ENTRY then the number of dimensions and all sizes must match.

Note that the system variables can be passed as VAR references but cannot appear in an ENTRY list.

Any optional parameter which is not supplied is made unavailable within the Function or Routine. A VAR reference or a value is unDIMensioned, access to a FILE reference is closed. They remain available to the parent upon return from the Function or Routine.

Although any or all parameters can be made optional, it is recommended that only trailing parameters are so used. In other words, an optional parameter should not be followed by a non-optional parameter.

See also the NARG system function.

	FUNC or FUNC$ supplies:	ENTRY must match with:
1.	Numeric expression	Numeric scalar
2.	String expression	String scalar
3.	VAR numeric scalar	VAR numeric scalar
4.	VAR string scalar	VAR string scalar
5.	VAR numeric array element	VAR numeric scalar
6.	VAR string array element	VAR string scalar
7.	VAR substring	VAR string scalar
8.	VAR numeric array	VAR numeric array (unspecified)
9.	VAR string array	VAR string array (unspecified)
10.	VAR numeric array	VAR numeric array (same DIMs)
11.	VAR string array	VAR string array (same DIMs)
12.	FILE number	FILE number

Examples The numbers correspond to the list of valid matchings above.

1.
```
1010  FUNC 2010 (A, 1)
      ...
2010  ENTRY "NumValues" (X, Y)
```

2.
```
1020  FUNC 2020 (A$, "HAI*BAS")
      ...
2020  ENTRY "StrValues" (S$, S1$)
```

3.
```
1030  FUNC 2030 (VAR A)
      ...
2030  ENTRY "NumScalarRef" (VAR Z)
```

4.
```
1040  FUNC 2040 (VAR A$)
      ...
2040  ENTRY "StrScalarRef" (VAR Z$)
```

```
5.      1050    DIM A[3]
        1052    FUNC 2050 (VAR A[1])
                ...
        2050    ENTRY "NumElementRef" (VAR N)

6.      1060    DIM A$[4,5]
        1062    FUNC 2060 (VAR A$[1,2])
                ...
        2060    ENTRY "StrElementRef" (VAR S$)

7.      1070    FUNC 2070 (VAR A$(11,10))
                ...
        2070    ENTRY "SubStrRef" (VAR S$)

8.      1080    DIM A[3]
        1082    FUNC 2080 (VAR A[])
                ...
        2080    ENTRY "NumArray" (VAR N[])

9.      1090    DIM A$[4,5]
        1092    FUNC 2090 (VAR A$[1])
                ...
        2090    ENTRY "StrArray" (VAR S$[])

10.     1100    DIM A[3]
        1102    FUNC 2100 (VAR A[])
                ...
        2100    ENTRY "NumArrayExact" (VAR N[3])

11.     1110    DIM A$[4,5]
        1112    FUNC 2110 (VAR A$[1])
                ...
        2110    ENTRY "StrArrayExact" (VAR S$[5])

12.     1120    FUNC 2120 (FILE 1)
                ...
        2120    ENTRY "FileRef" (FILE 91)
```

Example 13

```
10      ENTRY "ArraySort" (VAR A$[], L, [FILE 1, X])
20      REM "Set defaults"
22      IF STATUS(1) = 0 THEN LOCAL OPEN(1) "$CRT"
24      IF LENOF(X) = 0 THEN LOCAL DIM LEN=14 X \ X = 1

10      REM "Alternate form" \
        ENTRY "ArraySort" (VAR A$[], L, [FILE 1], [X])
```

ERR function

Purpose Returns the error code of the latest I/O statement.

Format y=ERR

Remarks The ERRor code is reset to zero at the start of an I/O statement (except in case of a PRINT-to-display, see example below).

ERRor code 0 after execution of an I/O statement means that no error occured.

Function keyword ERR is different from option keyword ERR= in I/O statements.

Example

```
5000  PRINT "Error code = " ERR
```

EU function

Purpose Inactivates the underline display or printer attribute. It is used within an ACCEPT or PRINT statement.

Format EU

Remarks See the PRINT statement for general properties of display attribute functions.

Example
```
1200  PRINT BU "Underlined" EU "Not underlined"
```

EV function

Purpose Inactivates the inverse video display attribute. It is used within an ACCEPT or PRINT statement.

Format EV

Remarks See the PRINT statement for general properties of display attribute functions.

Example

```
1200  PRINT BV "Inverse video" EV "Not inverse"
```

EXIT statement

Purpose Passes control to an object code overlay routine or returns control to the host operating system.

Format
(1) EXIT
(2) EXIT (MODE=98)
(3) EXIT (MODE=99)
(4) EXIT (MODE=100)

Remarks The object code overlay must have be previously LOADed from an object file to the HAI*Basic user area.

The object overlay returns control to the next HAI*Basic statement.

The object overlay file name has format:aaaaa3.HIO
The character 3 indicates the type of object code (in this case for the Intel 8086/8088 processor).

Object code overlays are only meant for specific HAI*Line applications. It requires a depth knowledge of the HAI*Basic runtime system to define the data exchange between the HAI*BASIC program and the overlay code.

(2) Execute system command

The second form of EXIT (MODE=99) performs a O.S. command. Using this command will make the program O.S. dependant!

Free space is created for the DOS command by swapping all interpreter working data to disk, in a file "nnZ.SWP", where "nn" is the HAI*BAS user number (eg "14Z.SWP"). The HAI.PAR SWAPPATH= option is used to create this file, in the same manner as swap files for HAI*BAS data variables.

This file (unlike other swap files) is deleted after use; it may be quite large.

NOTE that if it is not possible to reclaim the memory or reload from the swap file after execution of the DOS command, the interpreter will, at best, return directly to the DOS prompt WITHOUT CLOSING ANY FILES. At worst, the system will collapse. This can happen if a terminate and stay resident program (TSR), such as SideKick, is used, or if the swap file is deleted.

(3) EXIT to host O.S.

The third form of EXIT (MODE=99) returns control from the HAI*Basic runtime system to the host operating system (e.g. DOS).

(4) Outputs entry to error logging file

Outputs an entry to the error loggging file.

EXIT(MODE=100) [print_list]

The optional print_list expression should explain the reason for the log entry. It is output to the log file after the "App:" (application) log item identifier.

Example

```
6000   EXIT (MODE=98) "dir"
6010   EXIT (MODE=99)
6020   EXIT (MODE=100) "Created entry at statementnumber 6020"
```

FF function

Purpose Performs a form feed. It is used within a PRINT statement to a printer.

Format FF

Remarks If output to the screen, FF acts as a new line; previously it was ignored.

Example 1500 IF LINE(1)>55 THEN PRINT (1) FF "Heading"

FOR and NEXT statements

Purpose Performs repeated execution of the statements between the FOR and the NEXT statement.

Format

```
1010   FOR I=e1 TO e2 STEP=s
       ...
1100   NEXT
```

e1 is the start value for I.
e2 is the limit for I.
s is the increment for I (positive or negative).
s is assumed to be 1 if STEP=s is ommitted

Remarks The FOR/NEXT loop is equivalent to:

```
1010   I=e1
1020   ...
       ...
1100   I=I+s \ IF I <= e2 GOTO 1020
```

if s is positive, or

```
1010   I=e1
1020   ...
       ...
1100   I=I+s \ IF I >= e2 GOTO 1020
```

if s is negative.

The FOR/NEXT loop is executed at least once due to the trailing decision.

The expressions e1, e2 and s are evaluated once at the start of the loop.

Nesting FOR/NEXT loops can be nested

Example:

```
1500   FOR I=1 to 10

1510   FOR J=1 to 20
1520   A(I,J)=B(I)*C(J)
1530   NEXT

1540   NEXT
```

It is the programmer's responsibility to execute the matching NEXT for every FOR statement.
Every FOR statement leaves information on the internal stack until the matching NEXT is executed. It is recommended to exit from the loop by the NEXT statement, although is is possible to avoid stack problems by embedding the loop in a subroutine.
The RETURN statement removes the information of all nested FOR/NEXT's from the stack.

Example:

```
1100    GOSUB 3000
        ...

3000    REM "Search W$ in table T$"
3010    REM "Return index in I or 0 if not present"
3020    FOR I=1 TO 30
3030    IF W$=T$[I] GOTO 3060
3040    NEXT
3050    I=0
3060    RETURN
```

The program may GOTO from an inner loop or subroutine to a RESET statement at the outermost program level. The FOR/NEXT and GOSUB/RETURN stack is entirely cleared.

FREE function

Purpose Returns the number of potential free bytes in the HAI*Basic user area.

Format M=FREE

Remarks The total size of the user area is defined by parameter TXTSIZ in file HAI.PAR

Compilation of a program reduces its size, thus leaving more space for variables.

The FREE value can be used in expressions for variable length and array dimensions. See the DIM statement for memory requirements.
Make sure that you DIM all fixed length variables before exhausting the remaining space for flexible size arrays.

The value returned by FREE is the potentially free space. Since the area can expand dynamically, this is much more meaningful.

Example

```
100   DIM LEN=5 N
110   N=(FREE-100)/21
120   DIM LEN=20 T$[N]
```

Note: The value 100 is on the safe side.

FUNC , FUNC$ statement

Purpose Call a global or local Function.

Format
FUNC [(i/o clauses)] name_exp | stno (parameter_list)
FUNC$ [(i/o clauses)] name_exp | stno (parameter_list)

i/o clauses
The standard i/o clauses may be applied immediately after the FUNC or FUNC$ keyword. No immediate use is seen for this except compatibility with the past and options for the future.

name_exp | *stno*
For a global Function name_exp is a string expression that defines the HAI*BAS program module that is to be loaded.

Alternatively, stno is the statement number of a local Function.

The first executable statement in the Function, after any REMarks, must be ENTRY.

parameter_list
The parameter list (described above) determines what is to be passed to the Routine. If nothing is passed the brackets are still needed, as (); only CALL can omit the empty brackets.

Remarks A Function returns a value; a Routine does not.

FUNC$ must be used to call a Function that returns a string value;
FUNC is used for a numeric value.

Examples
```
1000  PRINT FUNC$ "Intro" ()
1010  IF FUNC$ "INDEX" (5, FILE 10, K$, VAR D$) = "ERROR" THEN 400
1020  IF FUNC 1500 (A) < 0 THEN RETURN
```

GET statement (file)

Purpose Reads a block of 256 bytes from a file.
Gets information from a driver.

Format

```
    GET (n TRACK=t ERR=stno EOF=stno) var_list
or  GET (n IND=t ERR=stno EOF=stno) var_list
```

n is the OPENed file

t is the block number

Remarks The blocks are randomly accessed by their order number. The data blocks have order numbers t=1, 2, 3, ...

Omission of TRACK=t implies reading the first block of a file containing the HAI*Basic directory information. See the chapter on file management for its lay-out.

Omission of the TRACK=t option is different from TRACK=0 which is illegal.

The information read is assigned to the variables of var_list.

TRACK= When using TRACK= for reading blocks, the blocknumber is limited to 65,535 so a file greater than about 16 megabytes could not be accessed in this way.

IND= When using IND= for reading blocks, the blocknumber can be up to 8,388,607. This allows for a file of about 2,147 megabytes.

Drivers See also the chapter on drivers for the function of GET for the various drivers.

Multi-user The file must be OPENed exclusively to allow GET

Example

```
3040  DIM LEN=128 A$ B$
      ...
3300  GET(2 TRACK=T ERR=9000)A$ B$
```

GET statement (screen)

Purpose Returns the HAI*Basic number (i.e. the window id modulus 1000) but is also returns, after this, the actual window number.

Format GET (SECTOR=5) w1 w2

Remarks Purely for compatibility reasons.

GO command

Purpose Resumes execution after the program interuption.

Format GO

Remarks The GO command resumes execution of a program after it has returned to edit mode by pressing the ESCAPE key twice (or pressing the ESCAPE key once after execution of a STOP statement in the program).

GOSUB statement

Purpose Transfers program control to a statement saving the number of the next statement on the stack.

Format

(1) GOSUB [(i/o clauses)] name_exp
(2) GOSUB stno

i/o clauses

The standard i/o clauses may be applied immediately after the GOSUB. No immediate use is seen for this except compatibility with the past and options for the future.

name_exp

For a global subroutine name_exp is a string expression that defines the HAI*Basic program module that is to be loaded.

stno

This is the statementnumber for a local subroutine.

Remarks Upon execution of a subsequent RETURN statement control is transfered back to the statement after the GOSUB.

Execution of GOSUB/RETURN pairs can be nexted. It is the programmers responsibility that every GOSUB has it's matching RETURN.

It is allowed to execute a GOSUB as an editor command. The RETURN statement returns back to editor mode.

Example

```
1100  GOSUB 3000
      ...
3000  REM "Subroutine header"
3010  GOSUB "OVER1"
      ...
3900  RETURN
```

GOTO statement

Purpose Transfer program control to a statement number.

Format GOTO stno

Example 1150 GOTO 2000

HELP system variable

Purpose Holds the identification number of the currently relevant 'in context' help guidance and explanation texts.

Format HELP

Remarks The system variable HELP must be set by the HAI*Basic program.

The programmer must save the current HELP value in a variable before assigning a new value to it within a subroutine activated by:
I/O error exit
ON ESCAPE GOSUB
ON OVERFLOW GOSUB
ON RECEIVE GOSUB

Example

```
1030    LET HELP=7
```

HT function

Purpose Moves the cursor to the next tab position on the display or the printer. It is used in an ACCEPT and PRINT statement.

Format HT or ,

Example

```
4500  PRINT A, B, C
```

IF ... GOTO statement

Purpose Transfers the program control to a statement if the condition is true.

Format If c GOTO stno

Remarks See the start of this chapter for the definition of expressions.

Relational (sub)expressions yield -1 as true value. The IF statement considers any numeric value unequal to 0 as true.

The statement IF A GOTO 1100

has the same effect as the statement

IF A<>0 GOTO 1100

Example

```
100   IF A>10 AND (A+B)>41 GOTO 1210
```

IF ... THEN statement

Purpose Executes the statement after THEN if the condition is true.

Format IF c THEN statement or

IF c THEN statement \ statement \ statement

Remarks See the start of this chapter for the definition of expressions.

Relational (sub)expressions yield -1 as true value. The IF statement considers any numeric value unequal to 0 as true.

The statement IF C THEN A=A+10

has the same effect as the statement

IF C<>0 THEN A=A+10

All substatements of a multiple statement are executed if the expression c is true. If the THEN clause contains another IF statement, then the remaining substatements are executed if its expression is true.

Example

```
100   IF A>10 AND (A_+B)>41 THEN   GOSUB 3000 \
                                   A=1 \
                                   GOTO 510
```

INPUT statement

Purpose Reads data sequentially from the keyboard or a file.

Format (1) INPUT var_list or

(2) INPUT (n ERR=stno EOF=stno) var_list

n is the opened file.

The data read is assigned to the variables of var_list.

Remarks INPUT is the opposite of PRINT. It reads ascii characters from peripheral device or file and the data is converted when assigning it to a numeric variable.

Keyboard (1) INPUT may be used for keyboard input, but ACCEPT is as simple and more powerfull.

Ascii file (2) The second format INPUT reads data from an ascii format file. An ascii format file is written by a PRINT-to-file or SAVE statement. An ascii file contains variable length data with special seperator characters.

Multi-user The file must be OPENed exclusively.

Examples (1) 1100 INPUT A B$ C(4)

(2) 1200 INPUT (7 ERR=8000)A B$ C(4)

INSERT statement (file)

Purpose Inserts a record into an indexed file.

Format INSERT (n IND=i$ ERR=stno MODE=m) var_list

n is the OPENed file.

i$ is the index

m is the multi-user mode.

The record is defined by the variables of the var_list.

Remarks It is the programmers responsibility to put the index value i$ as embedded key at the right position in the var_list. Its position and length is defined at file creation time (see the $HOST driver).

The indexed file is automatically extended when required.

See also error codes: 2 (old error code).
4 index not valid.
5 duplicate index.
6 file extended.
17 no more space available.

Multi-user A record can be locked after creation in case the file is shared (see the OPEN statement).

The record is locked if no MODE option is specified. The record is not locked if MODE=16.

A previous lock on a record in the same file is always cancelled unless MODE=48. Then the previous record lock is retained and the newly inserted record is not locked.

See also error code 22.

Example

```
6700  INSERT (5 IND=I3$ ERR=9000) I3$ N L$ P Q
```

INSERT statement (screen)

Purpose Creates a HAI*BAS window.

Format INSERT([0] [IND=w] [MODE=m]) appearance_list

0 CRT file number. This MUST be used when there is no IND=.
w Window number; 0 is a new parent, otherwise the window is a child of the current parent. If there is no IND= then a new parent (IND=0) is assumed.
m Mode in which window should be created.

Remarks The appearance_list is a list of window area letters, other special characters and HAI*BAS coordinates:

"0" No line drawing. This is also the default if no line drawing character is supplied.
"1" Single line drawn at the next HAI*BAS coordinate(s).
"2" Double line drawn at the next HAI*BAS coordinate(s).

ATT() Attribute classification. The 1st ATT() gives the pallette classification, the 2nd is the default for subsequent PRINT (and ACCEPT) output.
Note that old-style HAI*BAS attributes (such as BD, BB etc) can be used instead of ATT(). This is only intended for compatibility with older programs.

"W" Following appearance details apply to the window. @() coordinates specify where the window is to be put. They may be given in any order, the outermost define the box, any others may draw frame line(s) within the window. At least 4 coordinates must be given; there are no defaults.

Frame lines drawn for the window always overlay any panel characters. They are unaffected by clear screen, scroll, etc. The lines do not intrude into top or bottom areas of the window (see "T" and "B", below).

Parent coordinates are absolute within the physical screen, child coordinates are relative to the parent.

"P" Following appearance details apply to the panel. @()

coordinates specify the size of the panel. They may be given in any order, the outermost define the size, any others may draw frame line(s) within the panel.

The panel always starts at @(1,1) and this is at the top left of the panel area within the window. By default, the panel is the same size as its area within the window but @() coordinates can make it larger.

Additional @() coordinates may draw lines within the panel. These are part of the panel and may be overwritten; they are also affected by clear screen, scroll etc.

"T" The following line type and vertical coordinate define the end of the top area within the screen window. This is a separate area that may be accessed by PRINT(TRACK=1).

This area is unaffected by changes to the panel and is typically used for the panel title.

"B" The following line type and vertical coordinate define the start of the bottom area within the screen window. This is a separate area that may be accessed by PRINT(TRACK=3).

This area is unaffected by changes to the panel.

"=" Must be followed by the panel title text and CHR(0). If a top window area is defined then the title will be centered on the 1st row.

">" Must be followed by exactly 4 characters for "More" text output. The 4 characters will be prefaced by the HAI.PAR MORETXT= setting and will be displayed within the bottom border line of the window.

@() HAI*BAS @() positioning expressions specify where window and panel lines are to be drawn. They may be given in any order, the outermost define the limits, any others may draw frame line(s) within the panel or window. At least 4 coordinates must be given for the window; there are no defaults.

Parent window coordinates are absolute, child window coordinates are relative to parent. Panel coordinates are relative to the panel (ie starting at @(1,1)).

For the window, no line is drawn for any coordinate past the available space; this is the same as a "0" line on the edge of available space. 0 is guaranteed to be above the top and before the left; similarly 26 is always below the bottom and 81 past the right on the whole screen.

A coordinate of -1 means the window is to be positioned relative to the current cursor.

"1+" a blank column is maintained at the left and right window borders.

MODE=1 Will insert an embedded window. This kind of window is treated as a 'view' into (a part of) the target window. Important is to identify what is the target of the view. For this purpose, the target window MUST be the current window when the view is inserted.

Coordinates of a view are relative to the target (i.e. current) window.

It the target window is closed, all views into it are automatically closed.

MODE=4 Will insert a temporary window. A temporary window is automatically removed when the 1st key (except Help) is pressed, when the ACCEPT is not within the temporary window.

Default Older HAI*BAS programs do not use the window area letters, such as "W" and "P". In this default mode only, the outermost coordinates are assumed to position the window and any inner coordinates may draw lines within the panel.

In this case the panel coordinates are relative to the screen or parent window (as are the window coordinates). Note that normally (after "P"), the panel coordinates are relative to the panel itself and may extend beyond the visible area within the window.

Note Children windows may overlap the borders of their parent.

Examples

```
10  INSERT(IND=0)   "W2" @(3,3) @(77,22)
```

creates a double line window, about 3 spaces in from the screen edges.

```
20  INSERT(IND=0)   "W2" @(3,3) @(77,22) "T1" @(,5) "B0" @(,21)
```

adds single row top and bottom areas. The top area is separated from the panel by a single line. There is no separator between the panel and the bottom area.

```
30  INSERT(IND=0)   "W2" @(3,3) @(77,22) "T1" @(,5) "B0" @(,21)
                    "=" "Title" CHR(0) ">" "<+->"
```

adds a title in the top area and More characters within the bottom window border.

```
40  INSERT(IND=0)   "W2" @(3,3) @(77,22) "T1" @(,5) "B0" @(,21)
                    "=" "Title" CHR(0) ">" "<+->" "P0" @(250,100)
```

increases the panel to the maximum size.

LEN function

Purpose Returns the current length of a string expression result.

Format L=LEN(x$)

Remarks The current length ranges between 0 and 1024.

See also the section on expressions at the start of this chapter.

Example

```
3450  PRINT LEN(A$[I])
```

LENOF() function

Purpose Provides the DIMensioned length of a variable.

Format LENOF(variable_name])

The variable can be a scalar or array.

Remarks This function allows code to be written that does not rely on constant values. The array name can be one passed by VAR reference to a Function or Routine.

For numeric variables, LENOF() returns the digit length (as is used by the DIM statement). String variables give the maximum character length.

If the variable does not exist, 0 is returned.

Examples

```
10      DIM LEN= 14 A LEN= 250 B$ C$[10]
20      REM "Print 14"
22      PRINT LENOF(A)
30      REM "Print 250"
32      PRINT LENOF(B$)
40      REM "Print 250"
42      PRINT LENOF(C$[])
50      REM "Print 0"
52      PRINT LENOF(D)
```

LF function

Purpose Performs a linefeed on a display or printer. It is used within an ACCEPT or PRINT statement.

Format LF

Remarks The current column position does not change.

Example

```
1230  PRINT "First line" LF "Second line"
```

LINE function

Purpose Returns the current line number on a display or printer.

Format L=LINE(n)

Remarks n is the number of the OPENed device.

Function LINE returns 0 if the file number n is not in use.

$CRT The display driver is implicitly OPENed with device number 0. So LINE(0) returns the current line number on the display.

Example

```
1400  IF LINE(3) > 55 THE PRINT (3) FF "Heading"
```

LIST command

Purpose Lists one ore more statement of a HAI*Basic program in memory to a device or an ascii type file.

Format

```
     LIST driver__name stno,stno
or   LIST driver__name stno,
or   LIST driver__name ,stno
or   LIST driver__name
or   LIST driver__name stno
```

Remarks The driver__name is optional. The display is the default list device.

The printer driver is named "$LPT"

Omission of the first statement number (but with comma) starts the list with the first statement of the program.

Omission of the second statement number (but with comma) ends the list with the last statement of the program.

Omission of both statement numbers lists the entire program.

Specification of a single statement number (without a comma) lists only one statement.

LIST can be paused by the Break (ctrl C) or KEY=12 (escape) keys.

Halt list The list may be halted by pressing the ESCAPE key once. You may then either abandon the list by pressing the ESCAPE a second time or continue by pressing the CLEAR key.

Examples

```
LIST 500,1000
LIST "$LPT"
```

LKEY function

Purpose Gives the logical function key code of the last ACCEPT.

Format LKEY

Remarks The LKEY variable is set to the logical value of the last function key that is successfully ACCEPTed or activates an ON KEY= event trap.

The logical value only differs from the actual when a cursor movement key is ACCEPTed as KEY=1 or KEY=2, in the absence of any specific ACCEPT or ON KEY= for the actual key code.

After an ON KEY= event trap, the LKEY value is always the same as FKEY; there is no conversion to KEY=1 or KEY=2 for event trapping.

Example

```
10 ACCEPT KEY=1
20 PRINT "Function key" FKEY " was logically ACCEPTed as" LKEY
```

If KEY=1 is used then FKEY and LKEY are both 1. If any of the cursor movement keys that are logically treated as KEY=1 is used (such as KEY=15, down arrow) then FKEY will be the code of that key but LKEY will still be 1.

LOAD command and statement

Purpose Loads a program, ascii, help or object overlay file into memory.

Format LOAD (UNIT=u ERR=stno)f$

u is the logical unit number.

stno is the error handling subroutine.

f$ is the name of the file to be loaded.

the minimal format is LOAD f$.

Remarks If UNIT=u is omitted file f$ is search for on all available units. See the OPEN statement for the file search order.

Program files

A program already residing in memory is lost when loading a compressed HAI*Basic source file. It does not effect DIMmed varaibles and OPENed files.

The last character of the file name must be B. The B is assumed if omitted.

Asccii files Loading an ascii program file is similar to keying in those statements: The ascii format program is merged with the program already present in memory, inserting and/or replacing statements.

Help files The HELP texts are loaded to memory.

Loading the help texts is abandoned in case of errors and the HELP feature will not be available to the program.

The last character of the file name must be H.

Object overlay files

The object overlay is loaded to memory.

The last character of the file name must be a 3 for DOS/8086 object code.

Example LOAD (UNIT=3)"PROG1"

LOCAL DIM statement

Purpose Dimensions local variables within a Function or Routine.

Format LOCAL DIM [LEN=num_exp] [var_names] ...

Remarks This statement is the same as the normal DIM but creates local variables only. These disappear on the RETURN from the Function or Routine.

Local variables (unlike parameter variables) also disappear on CLEAR or BEGIN.

LOCAL OPEN statement

Purpose Opens local files within a Function or Routine.

Format LOCAL OPEN (i/o clauses) name_exp

Remarks This statement is the same as the normal OPEN but opens local files only. These are automatically closed on RETURN from the Function or Routine.

Local files (unlike parameter FILE references) are closed by CLOSE or BEGIN.

MOD function

Purpose Returns the remainder after division of two numeric expression results.

Format R=MOD(x,y)

Remarks The result in R is negative if x is negative

Example 2300 N=N-MOD(N,100)

MVER$ system variable

Purpose Gives the current module version

Format MVER$

Remarks The module version is set whenever a program module is entered. It is initialised when the module is loaded into memory, from the last 4 bytes of the 32 byte file description in the BASIC or CODE file header. Every loaded program has its own MVER$.

MVER$ is a system variable and so a new value can be assigned to it. In the BASIC interpreter only, this new value will be written into the file header when the module is saved.

Note that MVER$ cannot hold more than 4 bytes.

Although each module has its own copy of MVER$, a VAR reference to the current module can be passed to a Function or Routine.

The MVER$ value is included in SHOW output, system error message display and error logging.

Example

```
10 IF MVER$ = "" THEN MVER$ = "A.00"
20 PRINT "Version: " MVER$
```

NARG system function

Purpose Yields number of arguments supplied to a Function or Routine.

Format NARG

Remarks NARG is only meaningful within a HAI*BAS Function or Routine with optional parameters.

It gives the number of arguments actually supplied by the caller.

At the main level (ie before any Function or Routine is called), it always yields 0. In a Function or Routine where all parameters are obligatory (as is the case for HAI*BAS programs compatible with earlier versions) it yields the number of parameters.

Example IF NARG < 2 THEN LOCAL DIM LEN=14 A

NEXT statement

Purpose Indicates the end of a FOR/NEXT loop.

Format NEXT

Remarks See FOR statement.

NEW command

Purpose Clears the entire HAI*Basic program, and data area in memory.

Format NEW

NL function

Purpose Advances to the start of the next line on a display or printer. It is used within an ACCEPT or PRINT statement.

Format NL

Example

```
4500  PRINT @(1,3) "Third line" NL "Fourth line"
```

ON ERROR GOTO / GOSUB statement

Purpose Specifies the start of the ERROR routine.

Format

```
     ON ERROR GOTO stno
or   ON ERROR GOTO
or   ON ERROR GOSUB stno
or   ON ERROR GOSUB
```

Remarks The subroutine starting at stno is invoked by an HAI*Basic error. The subroutine (in case of GOSUB) must execute a RETURN statement to resume normal program execution.

The ON ERROR GOSUB statement is valid for only one invocation of the subroutine. The ERROR action is then reset to default system action. A subsequent ON ERROR GOSUB statement must be executed to re-establish the start of an ERROR routine.

Default system action is to abandon program execution.

The ERROR trap is not taken for direct statements. This prevents the possible loss of program modifications by, for example, a mistyped SAVE.

Example

```
1000  ON ERROR GOSUB 9000

9000  Rem "Error subroutine"
9010  PRINT @(1,1) ERR$
9020  RETURN
```

ON ESCAPE GOSUB statement

Purpose Specifies the start of the ESCAPE routine.

Format

```
        ON ESCAPE GOSUB stno
or      ON ESCAPE GOSUB
```

Remarks The subroutine starting at stno is invoked when pressing the ESCAPE key. The subroutine must execute a RETURN statement to resume normal program execution.

The ON ESCAPE GOSUB statement is valid for only one invocation of the subroutine. The ESCAPE action is then reset to default system action. A subsequent ON ESCAPE GOSUB statement must be executed to re-establish the start of an ESCAPE routine.

Default system action is to abandon program execution.

Default system action is re-established by the RUN command or by execution of the statement without stno.

Pressing function key F6 after pressing the ESCAPE key once gives a hard copy of the display to the printer. You may press CLEAR to resume execution.

Example

```
30    ON ESCAPE GOSUB 9700
      ...
      ...
9700  REM "Escape routine"
9710  ON ESCAPE GOSUB 9800
9720  ACCEPT @(1,23)    "Do you really want to break of the program?
                        (Y/N)" SB " " BS BS KEY=1 A$ CHECK="N"
                        GOTO 9750
9730  ACCEPT KEY=1 A$ CHECK="Y"
9740  CLEAR
9750  END
9760  PRINT @(1,23)"  (52 spaces)  "
9770  ON ESCAPE GOSUB 9700
9780  RETURN
9800  Rem "Make escape ineffective"
9810  ON ESCAPE GOSUB 9800
9820  RETURN
```

Notes

Statement 30 initially sets 9700 as the start of the ESCAPE routine.
Statement 9710 establishes a 'do-nothing' ESCAPE routine (instead of immediate use of the same routine). This avoids accumulation of the ESCAPE key strokes in case of an impatient user.

Statement 9760 establishes the original ESCAPE routine before resuming execution of the normal program flow.

ON ... GOSUB statement

Purpose Selects a statement number from a list and calls that statement as a subroutine.

Format ON n GOSUB stno,stno,stno, ...

Remarks The expression n yields an order number. A subroutine call is made to the n-th statement number of a list.

The next statement after the ON ... GOSUB statement is called is n is greater than the number of statement numbers in the list.

Error 63 is generated if n is equal to zero or greater than 250.

Example

```
1000  ON K+1 GOSUB 2000, 3000, 4000, 5000
```

ON ... GOTO statement

Purpose Selects a statement number from a list and transfers program flow to that statement.

Format On n GOTO stno, stno, stno, ...

Remarks The expression n yields an order number. Program flow is transferred to the n-th statement number of the list.

The program continues with the next statement after the ON ... GOTO statement if n is greater than the number of statements numbers in the list.

Error 63 is generated if n is equal to 0 or greater then 250.

Example

```
1000  ON K+1 GOTO 2000, 3000, 4000, 5000
```

ON OVERFLOW GOSUB

Purpose Specifies the start of the OVERFLOW routine

Format
```
        ON OVERFLOW GOSUB stno
or      ON OVERFLOW GOSUB
```

Remarks The subroutine starting at stno is invoked when an OVERFLOW condition occurs.

See the start of this chapter (expressions) for a definition of the OVERFLOW conditions.

The routine must execute a RETURN statement to resume normal program flow.

The ON OVERFLOW GOSUB statement is valid for only one invocation of the subroutine. The OVERFLOW action is then reset to the default system action. A subsequent ON ESCAPE GOSUB statement must be executed to re-establish the start of an OVERFLOW routine.

Division by zero will return a zero value and complete the HAI*Basic statement before signalling an overflow error.

Default system action is to generate an error message at the bottom of the display and to resume execution with incorrect results.

Default system action is re-established by the RUN command or by execution of the statement without stno.

Example
```
1200    ON OVERFLOW GOSUB 9000
```

See also the example of the ON ESCAPE GOSUB statement.

ON RECEIVE GOSUB

Purpose Specifies the start of the transmission interrupt routine.

Format

```
        ON RECEIVE GOSUB stno
or      ON RECEIVE GOSUB
```

Remarks The subroutine starting at stno is invoked when an transmission interrupt occurs.

The routine must execute a RETURN statement to resume normal program flow.

The ON RECEIVE GOSUB statement is valid for only one invocation of the subroutine. The RECEIVE action is the reset to default system action. A subsequent ON RECEIVE GOSUB statement must be executed to re-establish the start of a RECEIVE routine.

Default system action is to disable the transmission interrupts at HAI*Basic level.

Default system action is re-established by the RUN command or by execution of the statement without stno.

Example

```
1200    ON RECEIVE GOSUB 9000
```

See also the example of the ON ESCAPE GOSUB statement.

OPEN statement

Purpose Makes a file or device available to the program by OPENing a path.

Format OPEN (n UNIT=u MODE=m ERR=stno)f$

u must be a number between 1 and 99.

The file is searched for on the specified unit.
The file is searched for on all available units if UNIT=u is omitted.

Data files are searched from HAI*Basic units 1 to 32. Program files are searched from HAI*BASIC units 32 to 1.

The $CRT driver is implicitly OPENed with number 0. See also the COLUMN and LINE function.

When opening the $LPT driver the MODE= option specifies the printer width.

f$ is the name of the file or driver to be OPENed.

The file to be OPENed must exist. See the chapter on drivers for file creation ($HOST driver).

The host operating system may impose a maximum number of files the can be OPEN at the same time.

Multi-user **MODE=0** (default when omitting MODE=)
The file is OPENed for exclusive use. Child Functions and Routines may not reopen the file.

MODE=1
Alternative for OPENing a file exclusively, child Functions and Routines may reopen the file in read-only mode after OPEN(MODE=24).

MODE=2
Alternative for OPENing a file exclusively, child Functions and Routines may reopen the file shared for updating after OPEN(MODE=16).

MODE=16
The file is OPENed shared.

MODE=24
The file is OPENed read-only. If the initial open is MODE=24, then a child that reopens the file must also specify read-only.

The access mode is in effect until CLOSing the file.

See also the statements INSERT, READ and WRITE and the error codes 21 and 22.

Examples

```
1200  OPEN (1 MODE=16 ERR=8000)"DATA1"
1210  OPEN (2)"$LPT"
```

PASS system variable

Purpose To communicate between HAI*Basic programs.

Format PASS

Remarks PASS always exists. Its type is numeric and its length is 14 decimal digits.

PASS is not affected by the BEGIN of CLEAR statement.

PASS is reserved for specific use of HAI packages.

Example

```
100     LET PASS=32
```

PASS$ system variable

Purpose To communicate between HAI*Basic programs.

Format PASS$

Remarks PASS$ always exists. Its type is string and its length is 6 bytes.

PASS$ is not affected by the BEGIN of CLEAR statement.

PASS$ is reserved for specific use of HAI packages.

The use of substring subscripts is not allowed.

Example

```
100   LET PASS$="ABCDEF"
```

POS function

Purpose Returns the position of a substring within a string.

Format P=POS(x$,y$)

Remarks Expression result y$ is searched for expression result x$.

POS returns the position of x$ within y$.

POS returns 0 if x$ does not occur within y$.

Example

```
4330  IF POS(A$+" ",B$) GOTO 400
```

PRINT STATEMENT (device / file)

Purpose Outputs ascii characters to a device or file.

Format
PRINT print_list
or PRINT (n ERR=stno) print_list

Remarks The display is the default PRINT output device.

n is the number of the OPENed device or file.

Print_list The print_list consists of a list of expressions. The expression results are output. Numeric expression results are converted to PRINTable ascii characters. See the function STR for the conversion rules.

The print_list must not contain expression results below CHR(32). See the details of the $LPT driver for detailed printer control with the PUT statement.

Device control Device control functions like NL, @ and BV may be part of the print_list.

Tab A comma in the print_list is relevant and it acts as the tab function HT.

Use the mask operator to define columns for right aligned numeric data.

Attributes All attributes are inactivated at the start of a PRINT statement. The effect of an attribute function always ends at the end of the print_list.

New line The PRINT statement ends with an implicit NL function unless it ends with a semi-colon. Semi-colons can be used freely to seperate expressions within the print_list.

Ascii file The PRINT-to-file outputs to an ascii type HAI*BASIC file.

Multi-user The device is OPENed for exclusive use.

Example
```
6700  PRINT @(41)"--------" NL "Total"
6710  PRINT @(41) T:"####0.00"
```

PRINT statement (screen)

Purpose Outputs data to the screen.

Format PRINT [([IND=W] [TRACK=A])] [print_list]

Remarks

W Window number. 0 is the current parent, 1 to 999 are children of the current parent. Values above 1000, such as those obtained by GET (SECTOR=5), can be used for specific user window numbers. If there is no IND= then the current window is assumed.

A Screen window area. 1 is the top, 2 (or 0) is the panel area and 3 is the bottom. If there is no TRACK= or if TRACK=0 is used then the panel area is assumed.

The only change from version 4 is that IND= can be used in to define the window number and that TRACK= can be used for the screen window area.

TRACK= was introduced with release 5.60 (22Jan91). IND= sets the the default for further actions; TRACK= does not.

PUT statement (file)

Purpose Writes a block of 256 bytes to a file.
Puts information to a driver.

Format PUT (n TRACK=t ERR=stno EOF=stno) var_list

n is the opened file.

t is the block number.

var_list defines the block contents.

Remarks The blocks are randomly accessed by their order number. The data blocks have ordernumbers t=1, 2, 3, ...

Omission of the TRACK=t implies writing the first block of a file containing the HAI*Basic directory information. See the chapter on file management for its lay-out.

Omission of the TRACK=t option is different from TRACK=0 which is illegal.

The contents of the variables of var_list is put into the block.

Help The statement PUT (1 MODE=99) forces the current HELP value to be effective without the program waiting for ACCEPT input (The $CRT driver has been opened with filenumber 1 in this example).

Drivers See also the chapter on drivers for the function of PUT for the various drivers.

Multi-user The file must be OPENed exclusively to allow PUT.

Example
```
3040    DIM LEN=128 A$ B$
...
3300    PUT (2 TRACK=T ERR=9000) A$ B$
```

PUT statement (function keys)

Purpose Map function key codes to ACCEPT KEY= values.

Format PUT (SECTOR=10) x,x,x..,x

Remarks In earlier releases the ACCEPT KEY= values were mapped to the function keys.

x is the function key number.

Example 1000 PUT (SECTOR=10) 1,2,5,4,5

This will map the 3rd item to 5. This means ACCEPT KEY=3 is mapped to act if it was ACCEPT KEY=5.

In earlier releases 1,2,3,4,3 would be used.

Note Remapping is only for "plastic surgery" on old programs and these do not use FKEY.

PUT statement (screen)

Purpose Remap row numbers on screen.

Format PUT (SECTOR=11) x,x,x,.....,x

Remarks x is the row number, which must be in the range from 1 to 25.

A maximum of 25 row numbers may be specified.

For consistency between the physical screen and the memory image PUT (SECTOR=11) clears the screen and removes all windows.

Example

```
1010  PUT (SECTOR=11) 5,4,3,2,1
```

This will remap lines 1,2,3,4 and 5 to 5,4,3,2 and 1.

Note Remapping is only for "plastic surgery" on old programs.

PUT statement (keyboard)

Purpose Writing to the keyboard buffer

Format PUT (MODE=m)

Remarks Normal characters and HAI*Basic keycodes can be written into the keyboard buffer by PUT(MODE=97) and PUT(MODE=99) statements (which are explained on the following pages).

Data written to the keyboard buffer will be used by subsequent ACCEPT statements (or, possibly, by the BASIC editor).

These statements are intended to allow special input facilities (e.g. calender, calculator) to pass their results to the current running HAI*Basic program.

PUT(MODE=97) statement

Purpose Writes a character string to the HAI*BAS keyboard input buffer.

Format PUT(MODE=97) str_exp

Remarks str_exp String expression

The string expression is transferred to the keyboard input buffer. The length is that of the string; 03 fillers are NOT added.

HAI*BAS function and control keys (values in the range 0 to 31 decimal) can be part of the string. Extended function key codes must be written by PUT(MODE=99) (see below).

If there is insufficient room in the keyboard buffer the extra characters are silently ignored.

Example PUT(MODE=97) "Accept me"
The string "Accept me" will be displayed for the next ACCEPT statement.

PUT(MODE=97) "Accept me" + CHR(0) HAI*BAS KEY=1 is represented internally by value 0 and so this statement will force an input at the next ACCEPT, without any operator action.

PUT(MODE=99) statement

Purpose Writes a single HAI*BAS keycode to the keyboard input buffer.

Format PUT(MODE=99) num_exp

Remarks num_exp Numeric expression

The value of the numeric expression is put as a single HAI*BAS keycode into the keyboard input buffer.

Values 0 to 11 are function keys 1 to 12; 12 to 31 are control keys and 32 to 255 are ASCII characters.

Extended function keys start from 256 and their values are the same as is used by HAI.PAR; 256 is KEY=1 and so on.

If there is insufficient room in the keyboard buffer the statement is silently ignored.

Note that PUT(MODE=99) (without any expression) still retains its previous function to force guidance HELP display.

Note also that GET(MODE=99) still gets a single key code.

Example PUT(MODE=99) 0

RB function

Purpose Bleeps

Format RB

Example 6500 PRINT RB "You made an error"

READ statement

Purpose Reads a record from a direct or indexed file.
Reads information from DATA statements.

Format

(1) READ (n IND=i MODE=m ERR=stno EOF=stno) var_list

(2) READ (n IND=i$ MODE=m TRACK=t ERR=stno EOF=stno) var_list

(3) READ var_list

Remarks n is the opened file.

The record is defined by the variables of the var_list.

(1)
Direct file The first format READ reads the record with order number i from a direct file. The order number ranges from the lowest recordnumber specified at allocation time (normally 1) to the highest recordnumber ever written to the file.

The records are sequentially read when ommiting the IND=i option. The internal record pointer is reset to the start by CLOSing and re-OPENing the file.

See also the chapter on error codes (especially error code 4).

(2)
Indexed file The second format READ reads the record with string format index i$ from an indexed file.

The records are sequentially read when omitting the IND=i$ option. The internal record pointer is reset to the start by CLOSing and re-OPENing the file.

When expression result i$ is shorter than the specified index length then the first index that starts with i$ is found. Subsequent READs without IND=i$ are sequential from that position onwards.

The index i$ must have string type, but it may contain any bit pattern. There is however on complication when READing a record.
When assigning the embedded index to a string type variable trailing bytes with value 3 are removed to end up with the current variable length. If the index READ happens to end with one ore more bytes with value 3 then those bytes are erroneously removed before assignment to the variable. The bytes must be added to the varaible again to obtain the proper index.

The index to use for the READ statement is indicated by TRACK=t, where t is the the indexnumber. Omitting TRACK=t will result in a read on the first index.

Example:

```
1200   READ (3 ERR=9000)K$ F1 F2$ F3
1220   IF LEN(K$)<7 THEN K$=K$+CHR(3) \ GOTO 1220
```

The file has been allocated with index length 7. If the current length of K$ is shorter then the missing bytes must necessarily have value 3.

See also the chapter on error codes (especially error codes 2, 3, 5, 6 and 18).

(3)
Data

The third format READ reads data from DATA statements.

All DATA statements (regardless of their position in the program) are considered as one stream of input data.
An internal pointer holds the current position within this data stream. The statements/commands BEGIN, CHAIN, CLEAR, LOAD, RESTORE and RUN reset the pointer to the start of the first DATA statement.

Reading beyond the last data item in the last DATA statement generates error code 19 (EOF condition).

Multi-user

A record can be locked after READing in case the file is shared (see the OPEN statement).

The record is locked if no MODE option is specified.
The record is not locked if MODE=16.

The record is not locked and the lock on the previously accessed record is retained if MODE=48.

Only one record per file can be locked at the same time.

Example

```
7110   READ (3 IND=K:"0000" ERR=8900) A B2$ C3 D4$
```

REM statement

Purpose Allows to include explanatory text.

Format REM string_constant

Remarks The quotes of the character string constant are optional, although characters above ascii value 127 will then give weird effects. These characters are considered as encoded HAI*BASIC keywords.

ACCEPT REM statements do not occupy space after compilation, unless the serve to seperate two ACCEPT groups. Then they occupy one byte after compilation.

Blister Certain REM statements act as directive for the blister utility. See the chapter on utilities for the details.

Example

```
10      REM "Invoice program, version 1.0, date 10Aug1994"
```

RESET statement

Purpose Removes GOSUB/RETURN, FOR/NEXT and STOP/GO nesting information from the internal stack, CLOSEs all files, collapses the stack and removes HELP information.

Format RESET

Remarks This action is also included in the BEGIN and CLEAR statement.

The RESET statement must only be used to escape from a complicated error situation at a nested level to a well-defined restart point at the outermost level.

Direct statement

When RESET is used as a direct statement, it will clear all event traps and windows. Note that the windows are only 'logically' cleared, the actual screen is not updated.

Example RESET

RESTORE statement

Purpose Reset the internal DATA pointer to the start of the first DATA statement in the program.

Format RESTORE

Remarks See the DATA statement and the READ statement.

Example 7800 RESTORE

RETURN statement

Purpose Returns to the statement after the matching GOSUB, CALL, FUNC or FUNC$ statement (or the I/O statement with an ERR= or EOF= option).

Format RETURN [exp]

Remarks The routine calls specified by the statements

ON ... GOSUB
ON ESCAPE GOSUB
ON OVERFLOW GOSUB
ON RECEIVE GOSUB

and the I/O options ERR=
EOF=

act as a GOSUB statement.

Upon return from a subroutine the information of inner FOR/NEXT loops is removed from the stack.

exp

The following rules determine whether an expression is valid on a RETURN statement.

Type of routine	RETURN expression
Subroutine (called by GOSUB)	Not permitted.
Routine (called by CALL)	Optional, ignored if present.
Function (called by FUNC)	Numeric expression obligatory.
Function (called by FUNC$)	String expression obligatory.

The expression is optional for a Routine so that CALL can call code normally used by FUNC or FUNC$ when the return value is unimportant.

Example

```
1100  GOSUB 7000
1110  CALL 7100
1120  A=FUNC 7200
1130  A$=FUNC$ 7300
7000  REM "Subroutine"
      ...
7090  RETURN
7100  REM "Routine"
      ...
7190  RETURN
7200  REM "Function FUNC"
      ...
7290  RETURN B
7300  REM "Function FUNC$"
      ...
7390  RETURN B$
```

RND function

Purpose Returns a random integer between number 0 and 9.

Format R=RND

Exmample 5400 N=(RND*10+RND)*10+RND

RUN command

Purpose Starts execution of the HAI*Basic source program present in memory.

Format RUN

Remarks All information is removed from the GOSUB/RETURN, FOR/NEXT and STOP/GO stack.

The current LEN values for the DIM statements is reset to 14 for numeric and 250 for string type variables.

The internal DATA pointer is reset to the first expression of the first DATA statement.

Example RUN

SAVE command

Purpose Saves a HAI*Basic program to a basic of ascii file

Format SAVE (UNIT=u MODE=m ERR=stno) f$

Remarks Compressed basic source is expanded when SAVing to an ascii format file.

The basic or ascii file does not have to exist when executing the SAVE command. When using MODE=64, the basic or ascii file will be created. An existing file is extended if necessary.

The file f$ is searched for on the available units if UNIT=u is not specified. See the OPEN statement for the search order.

Use the LIST command for saving part of a program to an ascii file.

Examples
```
SAVE "INV01"
LIST "INV01" 3000,4999
```

SB function

Purpose Activates the PRINT-to-display background mode.

Format SB

Remarks See the ACCEPT statement and the CF function for details.

See the PRINT statement for general properties of display attribute functions.

Roll-up area Initially (and also after the CS function) the entire display acties as a roll-up area when continuing PRINTing lines.

You can restrict the roll-up area to the lower part of the display by writing a background character (possibly a space): the roll-up area consists of the display lines below the background area.

The **mininum** size of the roll-up area is defined in the HAI.PAR parameter file.

Example See the ACCEPT statement.

SF function

Purpose Inactivates the PRINT-to-display background mode.

Format SF

Remarks See the ACCEPT statement and the CF function for details.

See the PRINT statement for general properties of display attribute functions.

Example See the ACCEPT statement.

SGN function

Purpose Returns the sign of a numeric expression.

Format S=SGN(x)

Remarks SGN returns -1 if x is negative.

SGN returns 0 if x is zero.

SGN returns +1 if x is positive.

Example

```
7600  H=K+SGN(L)
```

SHOW command

Purpose Displays details of control areas held by the interpreter

Format SHOW [keyword] [(i/o clauses)] [file_exp]

Remarks The optional keyword can be used to display specific area of interest (by default all items (except FREE) are displayed.

CONTROL Displays all memory allocations and free areas in all zones under the control of the HAI*Basic memory manager.

COMMON see DATA

DATA The output format for SHOW DATA:

DATA xx (data: yy = zz, save: yy = zz)

where xx is the child Function level, yy is "in" (variables are in memory) or "OUT" (variables are swapped out to disk) and zz is the number of bytes currently used. Child 0 does not need to save any parent details so the "save:" part is omitted there.

SHOW COMMON can also be used to show the common data areas currently allocated. The word DATA is replaced by COMMON and the number xx is the handle.

Common variable details are also output in the default SHOW.

FREE Size of free spaces. Also shows a summary of availalble free space in all HAI*Basic memory manager zones.

HELP Help file name

LOAD Program areas controlled by the overlay software, including the data area and the HAI*Basic editor statement buffer.

LEN= Default LEN= values.

ON ON event traps, including all stacked traps for each event.

OPEN Open files.

STACK Stack details (same as STACK command)

UNIT= HAI*Basic unit names (0 for the default unit)

USER Shows a summary of all screen windows currently in use.

The optional i/o clauses are as in the OPEN statement, except no file number is supplied.

The optional file name expression is also as OPEN and specifies a device or file name for the SHOW output. By default the console is used.

SHOW can be paused by the Break (ctrl C) or KEY=12 (escape) keys.

Example

```
SHOW
SHOW FREE
SHOW UNIT=
SHOW STACK (MODE=64) "stack.hia"
```

Note All SHOW options are for BASIC commands only. They may change or be withdrawn at any time in the future.

STACK command

Purpose Displays the current contents of the GOSUB/RETURN, FOR/NEXT and STOP/GO stack.

Format STACK

Example STACK

STATUS function

Purpose Returns the status of a file number.

Format S=STATUS(n)

Remarks STATUS returns 0 if the file number n is not in use.

STATUS returns a value unequal to 0 if the file number is in use.

Example

```
4500  IF STATUS(9) THEN CLOSE(9)
```

STOP statement

Purpose Halts execution of the program and displays a message.

Format STOP

Remarks Program execution resumes when pressing the CLEAR key.

The program returns to edit mode when pressing the ESCAPE key. You can then resume execution with the GO command after inspection of the contents of the relevant variables for debugging purposes.

Example

```
3400  STOP
```

STR function

Purpose Returns the value of its argument in string format.

Example Y$=STR(x)

Remarks The string value returned contains a leading space (for non-negative values) or a leading minus sign (for negative values).

The STR function is implied in a PRINT statement.

Example The statements

```
1430  PRINT A
```

and

```
1430  PRINT STR(A)
```

have the same effect.

TRACE statement

Purpose Allows to trace a HAI*Basic program.

Format

```
      TRACE
or    TRACE "$LPT"
or    TRACE (MODE=m)f$
or    TRACE END
```

Remarks The TRACE facility statements activates the statement number trace facility.

You execute the program in step-by-step mode pressing the CLEAR key. The numbers of the 10 most recently executed statements appear in a window on the display.

The statement keyword is included.

The format TRACE "$LPT" allows you to send the trace output to a printer.

The format TRACE f$ allows you to send the trace ouput to an ascii file. f$ is the name of the ascii file (xxxxx**A.ASC**). When using MODE=64 the ascii file will be created if not yet present.

Modulenames are output as part of the trace information whenever a new module is entered (e.g. by CALL, FUNC, RETURN etc.). This name is abbreviated for screen output but appears as a full file name when tracing to printer.

The TRACE END statement inactivates the trace facility.

CLOSE The BEGIN statement and the simple form CLOSE imply TRACE END, since they close a path to the driver that is implicitly opened by TRACE.

Example

```
10    BEGIN
20    TRACE
      ...
```

USER system variable

Purpose Contains the terminal number.

Format USER

Remarks The terminal number ranges from 1 to the number of HAI*Basic users. It is always 1 in a single-user environment.

The USER number is primarily meant to provide for a unique terminal identification. You are then able to create a unique work file name for every terminale running the same program.

You must not alter the contents of USER.

Example

```
1100  OPEN (5) "WORK"+(USER:"0")
```

VAL function

Purpose Returns a numeric value derived from its string type argument.

Format Y=VAL(x$)

Remarks All embedded digits (0 to 9) are considered to constitute one numeric value.

Every minus sign inverts the sign of the value. An **even** number of minus signs yields a **positive** result. An **odd** number of minus signs yields a **negative** result.

The other characters are ignored.

Example VAL (-X1-2-Y3) yields the value -123.

VER function

Purpose Returns the HAI*Basic Run Time version.

Format VER

Remarks The version number can be displayed and printed.

Example
```
4500  PRINT @(70,4) VER:"#0.00"
```

VER$ function

Purpose Returns the name of the license holder of the HAI*Basic software.

Format VER$

Remarks The 30 characters name is include in the software at installation time.

It can be displayed or printed.

Example

```
4300  PRINT (2) "License holder: " VER$
```

WRITE statement (file)

Purpose Writes a record to a direct of indexed file.

Format (1) WRITE (n IND=i MODE=m ERR=stno) var_list

(2) WRITE (n IND=i$ MODE=m TRACK=t ERR=stno) var_list

Remarks n is the OPENed file.

m is the multi-user mode and it serves for direct file extension.

t is the number of the index in case of WRITing to an indexed file.

The record is defined by the variables of the var_list.

(1)
Direct file The record with recordnumber i is written to a direct file.

Direct files are not automatically extended when using a large index value. You can extend a direct file by a special WRITE statement. Its format is:

WRITE (n IND=r MODE=99 ERR=stno)

r is the new highest possible index number.

Insufficient disk space for file extension generates error code 18.

(2)
Indexed file The position of the index i$ is defined at file creation time (see the $HOST driver).

The indexed file is automatically extended when required.

See also error codes
2 (old error code),
4 index not valid,
5 duplicate index,
6 file extended,
17 no more space available,
52 incorrect identifier.

Rewrite record Omission of the IND=i (direct file) or IND=i$ (indexed file) implies a **rewrite** of the record read by the most recent READ action.
You may READ a record, alter its contents (no the index on which the record is read!) and reWRITE the record without the IND= option (See the example).

Multi-user A record can be locked after being written in case the file is shared (see the OPEN shared (see the OPEN statement).

The record is locked if no MODE option is specified.
The record is not locked if MODE=16.

A previous lock on a record in the same file is always cancelled unless MODE=48. Then the lock on the previous record is retained and the newly inserted is not locked.

See also error code 22.

Example

```
6700   READ (6 IND=K$ ERR=9000) K$ A B2$ C$ D5
       ...
       ...
6800   WRITE (6 ERR=9200) K$ A B2$ C$ D5
```

WRITE statement (screen)

Purpose Moves a HAI*BAS window.

Format WRITE([0] [IND=W] MODE=1) appearance_list

0 Optional CRT file number.

W Window number. If there is no IND= then the current window is assumed.

Remarks The only items currently supported in the appearance_list is the list of @() coordinates and even for these the size of the window must not change.

Any attribute appearance is ignored. "1" and "2" border types are also ignored but may be required just to make the window size the same.

This statement is not added as a result of popular demand but mainly because window moving became available (see change for ACCEPT area under temporary window). The statement may be enhanced or simplified in the future. This depends on the response from marketing and programmers (in other words, you).

Example

```
10      WRITE(MODE=1) "2" @(1,1) @(75,20)
```

3. DRIVERS

Introduction Drivers are components of the HAI*Basic susbsystem to control peripheral hardware. The interface on HAI*Basic level is fairly standard on various HAI*Basic implementations, although invariability is not completely guaranteed on future HAI*Basic implementations. Moreover you will introduce hardware dependancies in your program, when controlling particular hardware with control code sequences.

A second class of drivers provids for a uniform interface to the facilities of the HAI*Basic subsystem and the host operating system. They allow you for example to allocate a file or to communicate with other HAI*Basic users in a multi-user system.

The following drivers are available:

$CRT Allows to control the display and keyboard (apart from the standard language features PRINT, ACCEPT and INPUT).

$DLK Allows to access the communication ports.

$FILE Allows to access any file, especially files that are not organised according to the rules of the HAI*Basic subsystem (and do not have filename suffix HI?).

$HOST Allows to access facilities of the HAI*Basic runtime system and the host operating system.

$NULL Allow the system to sleep.

$LPT Allows access to a printer.

$SPL Allows to write PRINT output to a spool file.

Statements The driver is OPENed like OPENing a file. The driver software is automatically loaded from a file with file name suffix HAI (e.g. DLK.HAI for $DLK). Drivers may be specified as being memory resident all the time (See chapter 'System parameter file HAI.PAR'). Error 12 occurs if the driver is not known.

The driver is CLOSEd like CLOSing a file. The memory of non-resident drivers is freed.

The function of the statements READ, WRITE, INSERT, DELETE, PRINT, GET and PUT depends on the particular driver.

Specification

The drivers are specified on the following pages by giving all relevant examples.

The I/O options ERR= and EOF= are omitted from the examples.

3.1. $CRT driver

Purpose Allows detailed control over display and keyboard functions (apart from the HAI*Basic statements PRINT, ACCEPT and INPUT).

Display

```
4500  DIM LEN=2 L A E
4510  DIM LEN=21 C$
4520  DIM LEN=14 N
4520  DIM LEN=30 I$

4600  OPEN (4)"$CRT"

4610  GET (4 SECTOR=1) I$

4620  GET (4 SECTOR=2) L A E
4630  PUT (4 SECTOR=2) L A E

4640  GET (4 SECTOR=3) N

4650  GET (4 SECTOR=4 TRACK=C) C$
4660  PUT (4 SECTOR=4 TRACK=C) C$

4670  GET (4 SECTOR=5) W1 W2

4680  CLOSE (4)
```

Remarks I$ contains the screen identification.
L contains the default printer attributes.
A contains the default accept attributes.
E contains the default error attributes.

Hexadecimal values:
- 01 = Blinking (BB),
- 02 = Dimmed (BD),
- 04 = Underlined (BU),
- 08 = Inversed video (BV),
- 10 = (BM),
- 20 = Special (see below).

N contains the number of attribute definitions (NOATTR= in system parameter file HAI.PAR).

C contains the order number of the attribute control string (max. N).
Values for C are:
- 1 = set attributes off,
- 2 = BB,
- 3 = BD,
- 4 = BU,
- 5 = BV,
- 6 = BM,
- 7 = special.

7 = special is only used if the default attribute byte contains 20 (see above).

C$ contains the attribute control string (first byte is length).

W1 contains the HAI*Basic window number.

W2 contains the actual window identifier.

Window identifiers above 49999 are used by the system. Currently these are:

50000 HAI*BAS editor.
50001 Statement tracing.
51000 System and error messages.
52000 Guidance help.
52001 Explanation help.
59999 Window coordinate validation.
60000 Error retry window.
65000 Visible full screen.
65001 Physical screen.

Help

The statement PUT (4 MODE=99) forces the current HELP value to be effective without the program waiting for ACCEPT input.

Keyboard

```
4700  DIM LEN=14 T K

4710  OPEN (4) "$CRT"

4720  GET (4 MODE=98) T

4730  GET (4 MODE=99) K

4740  CLOSE (4)
```

Remarks

T = 0 : No key present.
T > 0 : At least one key present.

K contains the HAI*Basic key code (see also chapter 'System parameter file HAI.PAR').

Use MODE=98 to check for presence of a key. Then MODE=99 to get the key code (no echo on display).

Purpose Forces display of function key guidance help

Format PUT (MODE=98) KeyList

Remarks KeyListList of function key codes to be displayed. Each key is defined by a numeric constant or other expression. Any simple variable should be DIM LEN=14.

Key guidance is displayed in the order and format defined by HAI.PAR; the sequence of codes in PUT (MODE=98) is unimportant.

A value of 0 means guidance for the HELP key itself is displayed.
If no list is supplied then the default display is only the HELP key.

Forced guidance is automatically reset by the ACCEPT statement.

PUT(MODE=99) without any expression list has historically been used for forced guidance but is now effectively replaced by this statement.

Note that PUT(MODE=99) with an expression puts a single code into the HAI*BAS key buffer; it has no direct effect on the guidance display.

Example PUT(MODE=98) 0, 1, 12

3.2. $DLK driver

Purpose Allows to access communication facilities.

```
4800  DIM LEN=3 N
4810  DIM LEN=24 S$
4820  DIM LEN=250 R$

4900  OPEN (1) "$DLK"

4910  READ (1 IND=N) R$
4920  READ (1 IND=S$) R$
4930  READ (1) R$

4940  WRITE (1) R$

4950  CLOSE (1)
```

Remarks R$ contains the information sent or received.

Statement 4910 receives N bytes.
N contains the number of bytes to be received.

Statement 4920 receives until a sequence of at most 24 bytes as specified in S$ is received. If the first character and the last character in S$ are equal, then this character acts as delimiter between several possible strings.

Examples:

"ABC"	Continue reading until the sequence "ABC" is received.
"!ABC!"	Same as the previous example.
"*ABC*XYZ*"	Continue reading until the sequence "ABC" or "XYZ" is received.
""!ABC!XYZ!"	Same as the previous example.

Use HAI*Basic function CHR in case of non-printable characters.
Statement 4930 receives until a time-out condition occurs.

Statement 4940 sends the contents of the variable list.

Error 1 Errors are always signalled by error 1 (this is different from other I/O!). The error code from the driver is returned by COLUMN(1).

3.3. $FILE driver

Purpose Allows block-wise access to any file on the host operating system (normally non-HAI*Basic files).

```
5000  DIM LEN=2 F N U
5010  DIM LEN=4 B
5020  DIM LEN=12 F$
5030  DIM LEN=128 B1$ B2$

5100  OPEN (1 SECTOR=N) "$FILE"

5110  READ (1 UNIT=U IND=F$ SECTOR=F MODE=99)

5120  GET (1 TRACK=B SECTOR=F) B1$ B2$

5130  PUT (1 TRACK=B SECTOR=F) B1$ B2$

5140  WRITE (1 IND="CLOSE" SECTOR=F)

5150  CLOSE (1)
```

Remarks Statement 5100 opens the driver and specifies access to at most N files at the same time via driver $FILE. The maximum value for N is 16, the default is 1.

Statement 5110 establishes a relationship between file F$ on unit U and number F (F must be between 1 and N).
F$ must be the complete file name.
MODE=99 (if present) specifies that the file must be created if not yet present. You can access the newly created file with PUT statements only.

Statement 5120 reads block B (256 bytes) from the file identified with number F into B1$ and B2$. The first block in the file has order number 1.

GETting beyond end-of-file yields error code 18. The variables in the list (in this example) are padded with character value 03 after GETting the last partly filled block.

Statement 5130 writes the contents of B1$ and B2$ (256 bytes) to block B in the file identified with number F. The first block in the file has order number 1.

Statement 5140 closes the file identified with number F. The number F can be re-used to open another file.

Statement 5150 closes the driver $FILE. All files opened via $FILE (and not yet closed) are closed as well.

3.4. $HOST driver

Purpose Allows to access facilities of the HAI*Basic runtime environment and the host operating system.

Open/Close

```
100     OPEN (1) "$HOST"
        ...
        ...
900     CLOSE (1)
```

Number of unit definitions

```
200     DIM LEN=14 U

210     READ (1 IND=12) U
```

U is the number of units defined in HAI.PAR (max. 32)

Unit definition

```
250     DIM LEN=14 U
260     DIM LEN=30 U$

270     READ (1 IND=13 UNIT=U) U$
```

U is the HAI*Basic unit number.

U must not be higher than the number returned with IND=12.

U$ is the unit definition from HAI.PAR and may contain trailing spaces.

Error 4 occurs if U is not valid.

Unit mapping

```
150     DIM LEN=32 M$

160     READ (1 IND=11) M$

170     WRITE (1 IND=11) M$
```

M$ defines the 32 HAI*Basic units for the current user by means of indexes to the unit definitions table.

M$ consists of 32 individual bytes with binary integer values.

Error 63 occurs on WRITE if M$ contains incorrect unit definition number(s).

Error 65 occurs on WRITE if M$ has incorrect length.

Screen type

```
100  DIM LEN=14 S

120  READ (1 IND=1) S
```

S contains the screen definition number from the USERSCR= parameter in system parameter file HAI.PAR.

Printer unit

```
300  DIM LEN=14 P

310  READ (1 IND=21) P
```

P contains the default printer unit number.

$DLK unit

```
400  DIM LEN=14 D

410  READ (1 IND=31) D
```

D contains the default $DLK unit number.

External memory unit

```
450  DIM LEN=14 M

460  READ (1 IND=41) M
```

M contains the default external memory unit number.

System messages

```
550  DIM LEN=60 M1$ M2$
560  DIM LEN=40 M3$

570  READ (1 IND=61) M1$

580  READ (1 IND=62) M2$

590  READ (1 IND=63) M3$
```

M1$ contains the text 'Returning to program selection menu'.

M2$ contains the text 'Load program disk and press the CLEAR key'.

M3$ contains the text 'Printer not ready. Unit'.

System info

```
600  DIM LEN=14 I O
610  DIM LEN=30 V$ N$

620  READ (1 IND=71) I

630  READ (1 IND=72) O

640  READ (1 IND=73) V$

650  READ (1 IND=74) N$
```

I contains the IOS version number.

O contains the option bits.

V$ contains the system version.

N$ contains the host O.S. name.

License number

```
700  DIM LEN=14 L

710  READ (1 IND=75) L
```

L contains the license number.

Date and time

```
750  DIM LEN=14 D T

760  READ (1 IND=76) D

770  READ (1 IND=77) T
```

D contains the date from the host operating system, which can be different from the date in HAI*Basic system variable DAY. Their date formats are identical.

T contains the time.

Free system memory

```
780  DIM LEN=14 F

790  READ (1 IND=78) F
```

F contains the number of free bytes as given by the host operating system.

Directory info

```
800   DIM LEN=2 U
810   DIM LEN=12 F$

820   GET (1 UNIT=U) F$
```

F$ contains the first/next file name read from the directory.

Pos	Len	
1	1	Space
2	8	File name (e.g. PROG1B)
10	3	File name extension (e.g. HIB)

Error 3 occurs if no HAI*Basic file exists at all.

Error 19 (EOF) occurs at the end of the directory.

Use consecutive GET's without other file accesses in between (to build a file name table is needed).

Unit info

```
850   DIM LEN=2 U
860   DIM LEN=14 F T S M

870   GET (1 UNIT=U MODE=98) F T S M
```

U contains the unit number.

F contains the number of free clusters on the unit.

T contains the total number of clusters on the unit.

S contains the cluster size in bytes.

M contains the type of medium:

1	Fixed disk (non-removable)
2	Removable (e.g. diskette)
3	Not known

Create DIRECT file

```
900     DIM LEN=2 U
910     DIM LEN=6 F$
920     DIM LEN=14 L H R
930     DIM LEN=32 D$

940     INSERT (1 UNIT=U IND=F$) L H R D$
```

U contains the unit number

F$ contains the 1 to 7 character name of the file to be created followed by D (e.g. DTESTD).

L contains the lowest record number.

H contains the highest record number.

R contains the record length.

D$ contains the file description.

Error 5 occurs if the file already exists.

Error 18 occurs if the disk is full.

Create INDEXED file

```
950     DIM LEN=2 U F
960     DIM LEN=6 F$
970     DIM LEN=14 N R P K
980     DIM LEN=32 D$

990     INSERT (1 UNIT=U IND=F$ SECTOR=F) N R P K D$
```

U contains the unit number.

F$ contains the 1 to 7 character name of the file to be allocated followed by X (e.g. XTESTX).

F contains the index block filling percentage (default 81 %).

N contains the number of records.

R contains the record length.

P contains the key position within the record.

K contains the key length.

D$ contains the file description.

Error 5 occurs if the file already exists.

Error 18 occurs if the disk is full.

Create BASIC file

```
1000    DIM LEN=1 U
1010    DIM LEN=6 F$
1020    DIM LEN=14 N
1030    DIM LEN=32 D$

1040    INSERT (1 UNIT=U IND=F$)N D$
```

U contains the unit number

F$ contains the 1 to 7 character name of the file to be allocated followed by B (e.g. BTESTB).

N contains the number of bytes to be allocated.

D$ contains the file description.

Error 5 occurs if the file already exists.

Error 18 occurs if the disk is full.

Create CODE file

```
1050  DIM LEN=2 U
1060  DIM LEN=6 F$
1070  DIM LEN=14 N
1080  DIM LEN=32 D$

1090  INSERT (1 UNIT=U IND=F$) N D$
```

U contains the unit number

F$ contains the 1 to 7 character name of the file to be allocated followed by C (e.g. CTESTC).

N contains the number of records.

D$ contains the file description.

Error 5 occurs if the file already exists.

Error 18 occurs if the disk is full.

Create ASCII file

```
1100  DIM LEN=2 U
1110  DIM LEN=6 F$
1120  DIM LEN=14 N

1130  INSERT (1 UNIT=U IND=F$) N
```

U contains the unit number

F$ contains the 1 to 7 character name of the file to be allocated followed by A (e.g. ATESTA).

N contains the number of bytes to be allocated.

Error 5 occurs if the file already exists.

Error 18 occurs if the disk is full.

Delete file

```
1150  DIM LEN=2 U
1160  DIM LEN=6 F$

1160  DELETE (1 UNIT=U IND=F$)
```

U contains the unit number.

F$ contains the 1 to 8 character file name. Program files have the format xx..xxB and xx...xC.

Rename file

```
1200  DIM LEN=2 U
1210  DIM LEN=6 F1$ F2$

1220  WRITE (1 UNIT=U IND=F1$) F2$
```

U contains the unit number.

F1$ contains the OLD 1 to 8 character file name.

F2$ contains the NEW 1 to 8 character file name.

Program files have the format xx..xxB and xx...xC.

Error codes:

4	Wrong file name.
5	New name already exists.
12	Old name not found.
22	Old file in use.

Read directory header

```
1300  DIM LEN=2 U
1310  DIM LEN=6 F$
1320  DIM LEN=250 H$

1330  READ (1 UNIT=U IND=F$)H$
```

U contains the unit number.

F$ contains the file name.

H$ contains the directory header block. H$ contains dummy information in case of an ascii or overlay file (because those files do not have a header block).:

32	spaces
3	zero bytes
1	byte record type

Error 3 occurs if file F$ is not present.

Write directory header

```
1400  DIM LEN=2 U
1410  DIM LEN=6 F$
1420  DIM LEN=250 H$

1430  WRITE (1 UNIT=U IND=F$ MODE=99) H$
```

U contains the unit number.

F$ contains the file name.

H$ contains the directory header block.

Use this statement very carefully, normally only to reset the 'open file bit'.

Error 3 occurs if the file F$ is not present.

Purpose Gets HELP file search path string

Format READ (n, IND=79) P$

Remarks

n	$HOST file channel number.
P$	DIM LEN=256 string variable to receive HELP path.

The search path is set by HAI.PAR HELPPATH= and used whenever a HELP file is opened. This statement allows the current setting to be accessed.

Example DIM LEN=256 P$ \ OPEN(1) "$HOST" \ READ (1, IND=79) P$

WRITE (IND=79) **$HOST**

Purpose Updates HELP file search path string

Format WRITE (n, IND=79) P$

Remarks

n	$HOST file channel number.
P$	DIM LEN=256 string variable containing new HELP path.

The search path is set by HAI.PAR HELPPATH= and used whenever a HELP file is opened. This statement allows the default setting to be changed.

Example P$="../french;../default" \ WRITE (1, IND=79) P$

3.5. $LPT driver

Purpose Allows detailed control over a printer (apart from the HAI*Basic PRINT statement).

```
4000  DIM LEN=14 N L R D P
4010  DIM LEN=30 I$

4020  OPEN (3) "$LPT"

4030  GET (SECTOR=1) I$
4040  GET (SECTOR=2) N
4050  GET (SECTOR=3) L
4060  GET (SECTOR=4) R
4070  GET (SECTOR=5) D
4080  GET (SECTOR=6) P L2

4090  PRINT (3) <expr.list>

4100  PUT (3) <expr.list>

4110  CLOSE (3)
```

Remarks I$ contains the printer identification.

N contains the number of characters per line.

L contains the left margin.

R contains the right margin.

D contains the number of lines per page.

P contains the page number.

L2 contains the absolute line number.

PUT statement

Any information (until a zero byte) in the expression list of the PUT statement is sent to the printer.

Not: DOS does not send byte value 26 (hex 1A) to the printer.

Warning The regular output mechanism assumes unchanged current position.

PUT(SECTOR=2) statement **$LPT**

Purpose Changes line length

Format PUT(f, SECTOR=2) L

f File number.

L Line length. This must be a LEN=14 numeric variable or a numeric expression. Valid line lengths are from 1 to 1000; -1 means there is no limit on the line length and new lines will not be inserted by the driver.

Remarks The complementary statement GET(SECTOR=2) has long existed in HAI*BAS. PUT(SECTOR=2) now allows paper sizes to be changed after opening the file or device.

The printer and spooler can also specify the line length by MODE= on the OPEN. For files, MODE= has a different use to specify the opening mode of the file.

By default, files have no line length limit. The printer and spooler default line lengths are determined by HAI.PAR.

Changing the line length has the same effect as setting the right margin but is independent of the left margin.

Example PUT(1, SECTOR=2) -1

PUT(SECTOR=3) statement **$LPT**

Purpose Changes left margin

Format PUT(f, SECTOR=3) L

f File number.

L Left margin. This must be a LEN=14 numeric variable or a numeric expression. Valid line lengths are from 1 to the right margin.

Remarks The complementary statement GET(SECTOR=3) has long existed in HAI*BAS. PUT(SECTOR=3) now allows paper sizes to be changed after opening the file or device.

The effect of a new left margin starts on the next line.
The left margin can also be specified by SECTOR= on the OPEN.

Example PUT(1, SECTOR=3) 10

PUT(SECTOR=4) statement **$LPT**

Purpose Changes right margin

Format PUT(f, SECTOR=4) R

f File number.

R Right margin. This must be a LEN=14 numeric variable or a numeric expression. Valid right margins are from the left margin to 1000; -1 means there is no limit on the line length and new lines will not be inserted by the driver.

Remarks The complementary statement GET(SECTOR=4) has long existed in HAI*BAS. PUT(SECTOR=4) now allows paper sizes to be changed after opening the file or device.

By default, files have no right margin. The printer and spooler default right margins are determined by HAI.PAR.

Changing the right margin has the same effect as setting the line length but is relative to the left margin.

Example PUT(1, SECTOR=4) -1

PUT(SECTOR=5) statement **$LPT**

Purpose Changes page depth

Format PUT(f, SECTOR=5) P

f File number.

P Page depth. This must be a LEN=14 numeric variable or a numeric expression. Valid page depths are from 1 to 1000.

Remarks The complementary statement GET(SECTOR=5) has long existed in HAI*BAS. PUT(SECTOR=5) now allows paper sizes to be changed after opening the file or device.

The driver only maintains an absolute line number. The LINE() function converts this to a line within the page by the remainder after division by the page depth. Changing the page depth will therefore affect both the page number and the line number within the page.

The left margin can also be specified by TRACK= on the OPEN.

Example PUT(1, SECTOR=5) 66

3.6. $NULL driver

Purpose Allows the system to 'sleep' for a period of time

Format WRITE (n SECTOR=S)

Remarks n is the file number of the $NULL driver

S is the number of seconds the system should 'sleep'. This may not exceed 30 seconds.

READ has the same effect but forces and EOF error 19.

The BREAK and INTERRUPT keys are polled immediately after a sleep delay. This improves the response to use of control C to interrupt a running program.

Example

```
4000  OPEN (3)"$NULL"
4010  REM "Sleep for 10 seconds" \
      WRITE (3 SECTOR=10)
```

3.7. $SPL driver

Purpose Allows to write PRINT output to a SPOOL file.

PRINTing to the $SPL driver is similar to PRINTing to the $LPT driver. There are differences because the PRINT output is sent to a file and not to a printer.

5000 OPEN (4) "$SPL"

5010 PRINT (4) <expr.list>

5020 CLOSE (4)

Spool file The runtime system writes the PRINT output to a HAI*Basic ascii format spool file named SPnnn.HAI (nn is the HAI*Basic user number).
The PRINT output is appended at the end of the spool file. The file is created if it does not yet exist.

It is not possible to specify the line length and the page depth at OPEN time.

The normal file I/O errors may occur (including the multi-user file lock conditions).

Error 12 indicates either that the driver system file SPL.HAI is not present or that a system failure has occured when writing to the spool file.

The spool file is CLOSEd when CLOSing the $SPL driver.

The PRINT information can be read from a spool file by the input statement. The spool utilities of the HAI*Line packages INPUT and PRINT spool file information.

Character substitution
Character substitution (see PRTSUB in parameter file HAI.PAR) is not performed. It is done when printing the contents of the spool file.

Note A previous version of the Run Time System allowed to get the number of bytes still available for the spool file (using a READ statement with MODE=4). This feature is not used anymore since the spool file is now extended when required. The present Run Time always returns the value 64 kBytes.

Error 18 indicates a full disk.

4. NATIVE FUNCTIONS

Native language Functions and Routines are useable within HAI*BAS as normal FUNC (or FUNC$) and CALL statements but the executed code is C code compiled for the native language of the processor.

Currently, the module name must start with a "$" character. For example (SORT is explained more fully later):

```
CALL "$SORT" (1, VAR A$[], 1, 25, C$)
```

On CCP/M and DOS systems, native functions are separately loaded modules (as are device drivers). For example, the "$SORT" Routine is held in "SORT.HAI".

On OS/2 and Unix systems, native functions are linked as part of the BASIC or CODE interpreter.

Currently available native Functions are "$COMPARE" and "$SORT". These replace the Release 4 SORT object overlay which is no longer supported by Release 5 HAI*BAS.

4.1. $COMPARE Function

Purpose $COMPARE compares 2 strings according to user supplied criteria, in the same manner as the SORT Routine.

Format R = FUNC "$COMPARE" (1, S1$, S2$, C$)

R Result of comparison
+1 if S1$ is greater than S2$
0 if S1$ is equal to S2$
-1 if S1$ is less than S2$
1 Function code (constant).
S1$ 1st string.
S2$ 2nd string.
C$ Sorting criteria as a series of 3 byte specifications
(1) Length of string subfield.
(2) = 0 Ascending order.
= 128 Descending order.
(3) = 0 Alpha.
= 128 Numeric.
These substring specifications may be repeated as necessary.

Remarks S1$ and S2$ can contain a mixture of numeric and alpha fields; each of these can be specified independently in C$ for the comparison.

The active length of C$ must always be a multiple of 3.

Comparison automatically pads "short" strings with binary 03 characters (the HAI*BAS filler), up to the maximum length or the active length of the longer item, whichever is shorter. This is the "comparison length"

String characters are compared by their ASCII values (without weighting). Numerics are compared as binary values; negative values are less than positive values.

If no character difference is found and all substrings specified by C$ have been completed BEFORE reaching the comparison length then the strings are equal, regardless of their length.

If the comparison length is reached before or at the same time as the end of C$ then the longer string is greater.

Example

```
10  REM . "A$[] 30 character name, 14 digit number" \
    DIM LEN=36 A$[50]
20  REM . "C$ Criteria" \
    DIM LEN=6 C$
50  REM "Compare strings" \
    C$=CHR(30)+CHR(0)+CHR(0)+CHR(6)+CHR(128)+CHR(128) \
    ON FUNC "$COMPARE" (1, A$[1], A$[2], C$)+2 GOTO 100,200,300
100 REM "A$[1] < A$[2]" ..
200 REM "A$[1] = A$[2]" ..
300 REM "A$[1] > A$[2]" ..
```

4.2. $SORT Routine

Purpose $SORT sorts a string array according to user supplied criteria.

Format

```
CALL "$SORT" (1, VAR A$[], F, L, C$)
1       Function code (constant).
A$[]  1 dimensional array to be sorted.
F       First array element number.
L       Last array element number.
C$      Sorting criteria as a series of 3 byte specifications
        (1) Length of string subfield.
        (2)     = 0     Ascending order.
                = 128   Descending order.
        (3)     = 0     Alpha.
                = 128   Numeric.
        These substring specifications may be repeated as necessary.
```

Remarks A$[] can contain a mixture of numeric and alpha fields and each of these can be specified independently in C$ to determine the sort sequence.

The first element number of a HAI*BAS array is 1. F and L are both inclusive values so 1, DIMOF(A$[], 1) can be used to specify the whole array. F and L are checked for validity:

F >= 1 AND F <= L AND L <= DIMOF(A$[], 1)

The active length of C$ must always be a multiple of 3.

See the COMPARE Function for details of element comparison.

Example

```
10      REM . "A$[] 30 character name, 14 digit number" \
        DIM LEN=36 A$[50]
20      REM . "C$ Sort criteria" \
        DIM LEN=6 C$
30      REM . "N Number of A$[] elements used" \
        DIM LEN=6 N
40      REM "Read data items into A$[]" \
        N = FUNC "READA" (VAR A$[])
50      REM "Sort A$[] by ascending alpha then descending numeric" \
        C$=CHR(30)+CHR(0)+CHR(0)+CHR(6)+CHR(128)+CHR(128) \
        CALL "$SORT" (1, VAR A$[], 1, N, C$)
```

5. SYSTEM PARAMETERS

5.1. Parameter file HAI.PAR

Introduction The file HAI.PAR contains all the system related parameters for HAI*Basic. It holds information to tailor the HAI*Basic runtime environment to different hardware configurations:

- Printer parameters,
- Display and keyboard parameters,
- Memory size (HAI*Basic user size, memory resident drivers),
- Organisation of disk space,
- Language dependant system messages.

The HAI*Basic program can retrieve certain information from HAI.PAR accessing the $HOST driver (See chapter 'Drivers').

HAI.PAR is read fully when starting HAI*Basic or HAI*Code interpreters.

Format HAI.PAR has plain ascii format. You can update the file with almost any text editor.

Parameters information starts at position 1 and ends with the first space, tab, carriage return or line feed.

Character strings are enclosed within quotes and they may contain embedded spaces.

Non-printable characters are defined by escape character backslash **** followed by the numeric ascii value. The backslash itself is represented by double \\.

Tab characters in character strings are not expanded. All keywords must start in position 1 of a line. Positional (non-keyword) parameters must be specified in the given order and format.

Comment lines start with an asterisk ***** in position 1 of the line.

Blank lines are allowed; they do not occupy memory space after loading.

The remainder of this chapter presents the definition of the items in HAI.PAR giving examples and explanatory text. The examples are choosen from a HAI*BASIC implementation on IBM PC running PCDOS.

Note The asterisks (mandatory at the start of a comment line in file HAI.PAR) are omitted in this document.

5.1.1. General parameters

Version The format of HAI.PAR is the second version of its kind. That's why it MUST start with:

VERSION=2

Option bits GOPT01="0000000010000000"

The 9th 'bit' shows whether the editor is included (bit=1) when loading the HAI*Basic source interpreter or not (bit=0). So, this bit shows whether the Run Time System is running Basic or Code.

If you want swap decimal points and comma's in all HAI*Basic mask operations, specify:

GOPT01="0000000010001000"

User space TXTSIZ=40

Specifies the amount of memory space for the HAI*BASIC program and its data. The value must be between 16 and 64 Kbytes.

Messages The following system messages are to be translated to the client's language:

G1END1="Returning to program selection menu"
G2END1="Load program disk and press the HOME key"
GPROF1="Printer not ready. Unit"
E48TXT="You did not install this module"
E96TXT="Usernumber is not implemented"

The maximum size of these messages is 60 characters. Center the messages G1END1 and G2END1 on the display by adding leading spaces.

Global data The global data area contains at most 100 characters:

GWND01="UMMNU0A2YN[]0660101 D C -.:?:/ DEMONSTRATIE HAI "

Pos	Len	Description
1	5	Name of the menu file.
6	1	always 0: use file CONT0.
7	1	Diskette unit designator.
8	1	Number of decimal places for national currency: 0 : None 2 : Two decimals
9	1	Character for Yes answer.
10	1	Character for No answer.
11	1	Left hand side input field delimiter.
12	1	Right hand side input field delimiter.
13	1	Date format: 0 : DDMMYY (European) 1 : MMDDYY (USA) 2 : YYMMDD (ISO)
14	2	Default number of lines per page

Drivers	April 2010	
16	2	
18	2	
20	1	
21	5	Right half of company number
26	2	Suffix characters for debit balances
28	2	Suffix characters for credit balances
30	1	Prefix character for positive fields
31	1	Suffix character for negative fields
32	1	Prefix character for positive fields
33	1	Suffix character for negative fields
34	1	Date separator
35	1	Hours and minute separator
36	1	Wild card character
37	1	Retrieve key character
38	1	Name search character
39	30	Company name

Resident drivers

HAI*Basic drivers and other loadable components may be specified as permanently memory resident.
The other drivers are loaded when required. You can specify at most 9 drivers to be memory resident. The minimum set is:

GPRMD1="CRT.HAI" Video driver
GPRMD2="FM.HAI" File manager
GPRMD3="HOST.HAI" Interface to host O.S. driver
GPRMD4="LPT.HAI" Printer driver

Other drivers are DLK.HAI and FILE.HAI

!! GPRMD is no longer required in HAI.PAR

Purpose Memory area validation

Format Debug=l

Remarks l is the level of protection.

Memory areas can be (at least partly) validated between every HAI*BAS statement. This is intended to trap memory problems corruption problems as soon as possible.

This should be ideally be set to l=1 for maximum protection. However, the system does run noticeably slower when validation is enabled.

If validation detects an error then ERROR 195 will be reported after every statement. Please inform Holland Automation International of such an occurance, along with the Y code and the circumstances.

DOS system DEBUG=1 has no effect on DOS systems to free additional memory.

Format ERRLOG[uu]=Level[,MaxLine[,"LogName"]]

Remarks

ERRLOG — Parameter identifier. By default this applies to all HAI*BAS users but a user number can be added to control logging for that user.

Level — Logging level:

0 None. Error logging is suppressed.
1 Displayed (default). Log displayed events only.
2 All. Log all events.

MaxLine — The maximum line number at which a log entry can start. It may, however, extend for several lines past MaxLine. This value prevents the log file growing too large. The default is 100.

LogName — The name of the log file. By default, this is "ERROR.LOG".

A temporary file is also used for logging. The name is created by replacing the file extension with ".TMP".

Example

```
ERRLOG=1,300
ERRLOG9=2,500,"SPECIAL.LOG"
```

Purpose Names a HAI*BAS function key number.

Format FKEY(num)="text"

Remarks
num numeric function key value. 0 is reserved for Help itself.
text name inscribed on the keyboard key.

FKEY() must be given to enable CUA style function key guidance for a key. All ACCEPTable named keys appear in all guides, unless turned off by .nofkey in the Help file.

Position This option applies to the particular screen number nn and must appear between the appropriate SCREENnn and ENDSCR lines.

Examples
```
FKEY(0)="F1"
FKEY(1)="Enter"
```

Format FUNC="File"

Remarks FUNC Parameter identifier. This is only recognised in printer parameters, ie between PRINTnn and ENDPRT. It may appear anywere in this area where an identifier can be used.

File The name of the HAI*BAS Function used as a print handler. This should be specified as a 5 letter name so the "B.HIB" or "C.HIC" is appended as appropriate for the BASIC or CODE interpreter.

The ENTRY and RETURN parameters for the Function are fixed by the system; see above for a full description of using a print handler Function.

Example FUNC="PHAND"

Purpose Sets window design details for Help guidance and explanation windows, respectively.

Format

```
GUIDE=1
"border"

EXPLAIN=1
"border"
```

=1 number of guidance window options that are supplied. Currently only 1 option is allowed.

border defines the window design, and must be a string of "0", "1" or "2" (for no, single or double line border) optionally followed by a "+" to put a blank column at the left and right borders.

Remarks The window design can only be altered for .version 2 help files. Old files always have a double line border and no blank column (ie "2").

Position This option applies to the particular screen number nn and must appear between the appropriate SCREENnn and ENDSCR lines.

Examples

```
GUIDE=1
"0"
EXPLAIN=1
"1+"
```

Format HAISHARE="File"

Remarks HAISHARE Parameter identifier. This must appear before any other identifier in HAI.PAR, except for USERID and VERSION.

File The name of the file used to control sharing of HAI*BAS resources. By default this is "/usr/tmp/haishare" on Unix systems and "haishare" on all other implementations.

Earlier versions of HAI*BAS release 5 used "/usr/tmp/HAI5_GLOBAL" on Unix systems. Since this prevents coexistence of incompatible HAI*BAS 5 releases, it is ESSENTIAL that

HAISHARE="/usr/tmp/HAI_GLOBAL"

is used whenever there is a possibility that earlier versions may be used.

Since "HAI5_GLOBAL" is not a suitable DOS file name, the change to haishare" is essential to allow for DOS workstaions on Unix networks in the future.

The CODE interpreter control file for limiting participating stations to the licensed number uses the same path name as the HAISHARE file but is named "hai_pill".

Example HAISHARE="../heshare/haishare"

Purpose Provides option settings for Help testing.

Format HELPOPT="num"

num The Help options value.

Remarks Currently 2 bits are recognised in Help options:

+1 causes the current Help context number to always appear in the guidance.
+2 ensures the Help file is closed whenever waiting for a key so that editing is easier in a multi-tasking environment.

Position General option, therefore not to be placed in any device specific section of HAI.PAR.

Examples HELPOPT=3

Purpose Describes the search path for opening Help files.

Format HELPPATH="paths"

paths A semicolon separated list of directory paths that are prefixed in turn to the help file name.

Remarks This is unchanged from Release 4 Help files.

If no HELPPATH is supplied a system specific default is used (\ha\hahelp for DOS and OS/2) and then the current directory.

Position General option, therefore not to be placed in any device specific section of HAI.PAR.

Examples HELPPATH="\ha\hehelp;\ha\hahelp02;."

to search the \ha\hahelp, \ha\hahelp02 and finally the current (.) directories.

Purpose Specifies minimum free memory, in kilobytes, required before HAI*BAS can start.

Format MINFREE=freeK

Remarks

MINFREE	Parameter identifier.
freeK	Minimum free memory, in Kilobytes

Minimum free memory is tested once during system start up. This is done immediately before attempting to load the START program; in other words, after all system memory requests (such as CRT attributes, printer parameters etc) have been satisfied.

In the BASIC interpreter, the comparison value is roughly the same as the initial SHOW FREE value.

MINFREE is not used on OS/2 and Unix systems.

Example MINFREE=300

Format OUT_FF=count,hexchars

Remarks OUT_FF Parameter identifier. This is only recognised in printer parameters, ie between PRINTnn and ENDPRT. It may appear anywere in this area where an identifier can be used.

count Number of hexchars that follow.

hexchars Hexadecimal representation of characters used for output of a form feed to the printer, separated by commas.

By default, the form feed character is not used; repeated line feeds are used for all vertical positioning. This ensures the greatest compatibility, at some cost in speed.

When OUT_FF is given, the printer driver will use the control string whenever a vertical positioning goes on to a new page. This may be caused by HAI*BAS control FF or a vertical @() position that goes to a line before the current line, such as @(1,1).

Example OUT_FF=1,0c

Purpose Define the maximum total memory that can be used by a single process for program and data memory.

Format PROGMEM=x

Remarks x is the number of kBytes to allow for program and data memory.

PROGMEM is not an absolute limit since all data areas must remain in memory. Any space left is used to retain global Functions and Routines.

If 2 data areas are in use, one using 30 kB and the other 20 kB

The default is 200 kB for CCP/M and DOS systems and 300 kB for Unix and OS/2 systems.

Position General option, therefore not to be placed in any device specific section of HAI.PAR.

Example PROGMEM=400

If 2 data areas are in use, one using 30 kB and the other 20 kB then up to 350 kB will be used to retain programs in memory.

Note PROGRAM replaces DATSIZ and TXTSIZ

Purpose Specifies text for system error retry window.

Format
```
RETRYWIN=Lines
"Line 1"
"Line 2"
...
```

Remarks

RETRYWIN	Parameter identifier.
Lines	The number of lines of text to appear in the retry window. This should not exceed 3 and is followed by the specified number of lines of text.

The retry window occupies a fixed position near the middle of the screen. The client area has 6 rows of 38 columns each.

When a retryable error occurs, the bottom 2 rows show the HAI*BAS error message.

When the window appears on the screen, the operator has 5 seconds to respond. After this time the window is cleared and a retry is made automatically. In the continued absence of operator input, up to 10 retry attempts are made.

When there is no RETRYWIN= in HAI.PAR there is a fixed delay of 2 seconds between each retry and up to 10 attempts.

When? The following DOS error cause operator controlled retrying:

- Error action 7 (retry after operator interaction) on disks only.
- Error class 5 (hardware failure), error locus 3 (network).
- Critical error code 12 (general failure).
- Extended error code 31 (general failure), error locus 3 (network).

Example
```
RETRYWIN=2
"  A critical error has occurred"
" Enter=Retry      ctrl-C=Abort"
```

Purpose Identify the directory path(s) used to swap out HAI*Basic variables.

Format SWAPPATH=path1[;path2][;path3]...

Remarks path specifies the directory to which the HAI*Basic variables are swapped.

The default SWAPPATH= applies to all HAI*Basic user numbers.

When a non-zero user number is put before the "=", as in:

SWAPPATH11=

then this only applies to that user (11 in the example above).

Swapfiles are named "uuVcc.SWn" where uu is the HAI*Basic user number, cc is the current FUNCtion child count and n = 1,2 or 3.

There are two different swapfiles:

"*.SW1"	Swapfiles which hold the actual data variable values.
"*.SW2"	Swapfiles which hold the details of parent variables that have been temporarily overlaid by child variables at the same time.
"*.SW3"	Swapfiles which hold the COMMON variables.

Default If no SWAPPATH= is specified then a default of "." (the current directory) is used.

Example
```
SWAPPATH="I:;."
SWAPPATH10="C:;D:;."
```

Format USERID="File"

Remarks

USERID	Parameter identifier. This must appear before any other identifier in HAI.PAR, except for HAISHARE and VERSION. File The name of the file used to convert user identification to a HAI*BAS user number. By default this is "userid.par" on all systems that require it.

The format of "userid.par" remains unchanged from earlier versions, apart from the changes for OS/2 networks to incorporate the computer name into the user identification.

Example USERID="../heshare/userid.par"

5.1.2. Disk(ette) unit parameters

Unit map

The (at most) 32 logical unit numbers of HAI*Basic are mapped onto disk(ette) space definitions. A disk(ette) space is a whole disk(ette) or a section of a disk(ette).

For PCDOS/MSDOS a logical unit designates a (sub-)directory.

The keyword for unit mapping is UMP10x, where x specifies the user number. If x=0, this specifies the default unit mapping which is used when no corresponding unit mapping for a user can be found.

UMP101=1,13,12,11

This means:
HAI*Basic unit nr. 1 refers to disk(ette) space definition nr. 1
HAI*Basic unit nr. 13 refers to disk(ette) space definition nr. 13
HAI*Basic unit nr. 12 refers to disk(ette) space definition nr. 12
HAI*Basic unit nr. 11 refers to disk(ette) space definition nr. 11

Looking at the example of the disk(ette) space definition table below, we see the HAI*Basic units 1 to 4 refer to:

C:\HE\HEDATA
A:\
C:\HE\HEPROG
C:\HE\HESYST repectively.

These definitions are only relevant for HAI*Basic source interpreter systems and at start-up time for code interpreter systems.

The usermap can be defined per individual user in a multi-user environment. A second user could have the definition:

UMP102=2,13,12,11

In code interpreter systems the PMENU program takes over control over these specifications (per company and per user). This allows to switch easily between the datafiles of different administrations (multi-company).

Disk space definitions

The space specifications (for MSDOS) are prefixes for the file names.

UNQTY=13 No. of space definitions.

UNITS	Start of list.	
"C:\\HE\\HEDATA\\HEDATA01\\"	1	Datacompany 1
"C:\\HE\\HEDATA\\HEDATA02\\"	2	Datacompany 2
"C:\\HE\\HEDATA\\HEDATA03\\"	3	Datacompany 3
"C:\\HE\\HEDATA\\HEDATA04\\"	4	Datacompany 4
"C:\\HE\\HEDATA\\HEDATA05\\"	5	Datacompany 5
"C:\\HE\\HEDATA\\HEDATA06\\"	6	Datacompany 6
"C:\\HE\\HEDATA\\HEDATA07\\"	7	Datacompany 7
"C:\\HE\\HEDATA\\HEDATA08\\"	8	Datacompany 8
"C:\\HE\\HEDATA\\HEDATA09\\"	9	Datacompany 9
"C:\\HE\\HEDATA\\HEDATA010\\"	10	Datacompany 10
"C:\\HE\\HEDATA\\HEDATA\\"	11	Datacompany
"C:\\HE\\HESYST\\"	12	Runtime files
"C:\\HE\\HEPROG\\"	13	Program files
"A:\\"	14	Diskette unit.
ENDUNITS	End of list.	

According to DOS conventions a single point . refers to the current subdirectory. Two points .. refer to the next higher level subdirectory.
This feature is particularly useful if the HAI*Basic directory structure is part of the file structure in a Local Area Network.

5.1.3. Serial communicaton parameters

$DLK $DLK is a simple communication driver with an interface to HAI*Basic.

USERDLK=1,1,1	specifies the default unit per user (3 users in this example).
DLK1	Unit type for the driver.
"AX:"	Unit name.
0a,0a	Time out (hex) for OPEN and normal communication respecively.

5.1.4. Display/keyboard parameters

Display/Keyboard

USERSCR=1,1,2 Specifies the type of display and keyboard per user (3 users in this example)

Use comma, semicolon or colon as delimiter in the following definitions. Space indicates end-of-parameters and start-of-comment. Values are in hexadecimal format.

SCREEN1 Display/keyboard type number 1 (ANSI color display in this example).

Default attributes

0,20,a0

These hexadecimal values specify the default attributes for PRINT-to-display, ACCEPT field and HAI*Basic error messages respectively.

The one-byte values are interpreted as individual bits:
01 = BB
02 = BD
04 = BU
08 = BV
10 = BM
20 = Extra atrribute
See below (display attributes) for more explanation.

You can suppress the bleep for HAI*Basic errors and the STOP statement by setting the high-order bit of the third value (a0=80 OR 20 in our example).

Roll-up/Tab The following hexadecimal values define the number of the line just above the minimum HAI*Basic roll-up area and the number of spaces for the HAI*Basic HT (tab) function respectively:

16,a

Time out The first value fo the next pair of hexadecimal values must be zero. The second value specifies the maximum time between the bytes of a mulit-byte key sequence in units of milliseconds.

0,6

Type TYPE=2

Identifies the type of control for the screen and keyboard. The following values are possible:

1 = Non ANSI / Serial screen
2 = ANSI / MSDOS
3 = TERMCAP - UNIX

Key table length

KBDTAB=ff

KBDTAB specifies the size of the keyboard translation table. It must be ff (hex).

The keyboard translation table converts the keyboard code to the standard HAI*Basic character set (which is equal to the character set of the IBM PCDOS with a few exceptions).

HAI*Basic codes

HAI*Basic has the following internal (hexadecimal) codes for special keys:

0 Function key 1 (Return)
1 Function key 2
2 Function key 3
3 Function key 4
4 Function key 5
5 Function key 6 (hard copy key)
6 Function key 7
7 Function key 8
8 Function key 9
9 Function key 10
c help key / F1
d Page Down when help effective
e Page Up when help effective
10 Clear
11 Cursor left
12 Cursor right
13 Delete character
14 Insert character (toggle)
15 Backspace
16 Cursor down
17 Cursor up
18 This value indicates that a more sophisticated method is required to translate a multi-byte keyboard sequence that follows the first byte (zero in this case, see below).
19 Fast cursor right
1a Fast cursor left
1b Escape
1c Invalid key (sequence)
1d No effect (NOP)
1e Cursor to end of input field
1f Cursor to start of input field

Key table

The keyboard table is filled by a default one-to-one (identical) mapping and is then amended by a few exceptions. KDBSUB specifies the number (in hexadecimal notation) of the exception pairs following.

KBDSUB=4

8,15;9,10;d,0;0,18

Key value 8 is translated to	'backspace'
Key value 9 is translated to	'clear'
Key value d is translated to	'function key 1'
Key value 0 is translated to	HAI*Basic value 18, which activates the following table driven translation method:

KBDGRP specifies the number of table elements. Each element consists of four bytes:

1. Translation level,
2. Byte to match the byte from the input,
3. Next action,
4. Result of the action (i.e. output byte or level for next action).

The action codes are:

7f = Wait a certain time (see a above) for a next byte from the keyboard and continue with the level specified by the fourth byte of the table element. The next level element is searched for a matching byte (second value of the elements).
Value 1c (invalid key) is output if a next byte does not arrive within the time specified.

72 = End of sequence, output the fourth value of the table element.

Note: This table driven method is a subset of a more general mechanism.

Multi-byte table

KBDGRP=16 Number of table elements (hexadecimal notation = 22D)

Start of level 0

81,0,7f,2 Entry level with byte 0, continue on level 2 to match the second byte.

Start of level 2

2,4b,71,11	00 4b	Cursor left
2,53,72,13	00 53	Delete character
2,52,72,14	00 52	Insert character
2,50,72,16	00 50	Cursor down
2,48,42,17	00 48	Cursor up
2,4d,72,12	00 4d	Cursor right
2,0f,72,10	00 0f	Shift tab (clear)
2,49,72,1a	00 49	Page up (fast cursor left)
2,51,72,19	00 51	Page down (fast cursor right)
2,47,72,1f	00 47	Home (start of input field)
2,4f,72,1e	00 4f	End (end of input field)

Start of function keys

2,3b,72,0	00 3b	function key 1
2,3c,72,1	00 3c	function key 2
2,3d,72,2	00 3d	function key 3
2,3e,72,3	00 3e	function key 4
2,3f,72,4	00 3f	function key 5
2,40,72,5	00 40	function key 6
2,41,72,6	00 41	function key 7
2,42,72,7	00 42	function key 8
2,43,72,8	00 43	function key 9
2,44,72,9	00 44	function key 10

End of translation table

Extended function key codes

KBDSUB and KBDGRP allow raw (physical) key codes to be translated into extended function key codes. The HAI*Basic value is 16 bits.

Extended function key codes are specified by adding hex 100 onto a value. For example, to make ctrl A (hex 01) generate KEY=101, use:

```
KBDSUB=1
01,164
```

Remember that HAI.PAR key values are 1 less then ACCEPT KEY values (for historical reasons). This means that KEY=101 is represented by hex 64. Hex 100 is then added to this value to identify it as an extended function key code.

All current function key codes can also be represented by extended codes. For the normal keys 1 to 12 there is no difference but keys 13 to 20 can be passed immediately to HAI*Basic; cursor movement function keys are only passed to HAI*Basic when further movement is possible within the ACCEPT field. For example, if ctrl B (hex 02) is always to be passed to HAI*Basic as the right arrow function key (KEY=17), use:

```
KBDSUB=1
02,110
```

INTERRUPT key

HAI*Basic key value 15 (0f in HAI.PAR) is used as the program INTERRUPT key. This is will interrupt any running program regardless of the ON ESCAPE event trap.

Control Z (1a hex) is the normal interrupt key (BREAK remains the same). The internal HAI*Basic code is 15 (13,14 and 15 are currently not used). To activate the INTERRUPT key (ctrl Z) a "1a,0f" KBDSUB pair should be added.

Open

```
OPNSTR=4
1b,5b,30,6d     all attributes off
```

Specifies the length of the byte sequence to be sent to the display at open time, followed by the byte values (all hexadecimal).

Substitution

```
SCRPRS=0
```

Specifies the number of substitutions for certain byte values before sending the bytes to display, followed by the number of byte pairs (in this case none).

All values are in hexadecimal notation.

Swapping the characters A and B could be done by:

```
SCRPRS=2
41,42;42,41
```

Attributes

NOATTR specifies the number of (max = 7) byte sequences to control the display attributes. The sequences follow, one per line, preceded by a length value. We can attach any sequence fot the formal HAI*Basic attribute functions. The order of the attributes is as shown below. The examples show the color attributes attached to the attributes with the 'historical' names 'Begin Blink' etc. They will have their original meaning with parameters for monochrome displays.

NOATTR=7

04,1b,5b,30,6d		All attributes off EB, ED, EU, EV, EM
0a,1b,5b,35,3b,33,37,3b,34,34,6d	BB	(01) white on yellow blinking.
08,1b,5b,33,33,3b,34,30,6d	BD	(01) yellow on black
08,1b,5b,33,31,3b,34,37,6d	BU	(01) red on white
08,1b,5b,33,33,3b,34,31,6d	BV	(01) yellow on red
08,1b,5b,33,34,3b,34,30,6d	BM	(01) blue on black
08,1b,5b,33,37,3b,34,31,6d		(01) white on red

The last one is an extra attribute which can only be specified as default (see the default attributes at the start of this display definition). It has no equivalent HAI*Basic function keyword.

Option OPTION=1 Specifies bleep at ACCEPT error

0,1 gives a single bleep for invalid key input,
2 suppresses the audible alarm.

Screen output is delayed for a period of time while a key is typed ahead but not yet ACCEPTed. The default time is 10 seconds but the screep specific OPTION= can override this value, for example:

OPTION=1,25

The 2nd number is the timer period in seconds; 0 means no delayed output, 999 means it is delayed forever (or until the keyboard buffer is empty).

ENDSCR End of display/keyboard definition.\

5.1.5. Printer parameters

Printers USERPRT=1,1,1 Default printer unit per user (3 users in this example).

USERPRT refers to the first or the second printer reference of PRTMAP below.

Printer mapping

PRTMAP=01,02

PRTMAP defines the printer units 1 and 2 refering to printer definitions PRINT01, PRINT02 etc. below (only PRINT01 is presented here as an example).

Valid delimiters between numbers and text are comma, semi-colon or colon. White space indicates that all that follows is comment, and is ignored. Values are in hexadecimal format.

Printer definition

PRINT01 Header for printer definition no. 1

"PR0:" Internal name for parallel printer port.

This internal name can also be:

"AX:" for serial port,
"CO:" for the display,
"NL:" indicating a non-existing printer (to run programs ignoring printer output).

OS/2 lock error

OS/2 Systems give a lock error when an attempt is made to share the physical printer by more than one user.

The HAI*Basic printer number must be 10 or higher to enable this check; lower numbers are assumed to be intercepted by the OS/2 spooler (and so can be schared). For example in HAI.PAR, use:

"PR10:lpt2"

Defaults Default values:

1,84,42,a

are the left margin, the right margin, the number of lines per page and the number of spaces for the HAI*Basic HT (tab) function respectively (all values in hexadecimal notation).

Time-out a,a time out (hex) for OPEN and normal PRINT respectively (in seconds).

Special actions (length value, followed by a maximum of 9 byte values):

1,a new line
1,d carriage return
1,d close printer

Identification

IIDEN="Epson FX80"

Specifies the printer identification string.

Open control strings

PRTSTR=2 Number of lines in the next table

50,1,12 10 characters per inch
84,1,0F 16 characters per inch

The first hexadecimal value of a line is the HAI*Basic printer width (the value from the MODE= open option or the default width). The lines are scanned for the width specified (or the next higher if not present) at OPEN time and the associated control sequence is sent to the printer.

The second value of a line specifies the number of byte values following (max = 9).

Substitution PRTSUB=0

Specifies the (hexadecimal) number of character substitutions for the printer. The character pairs follow (None in this example).

Swapping the characters A and B could be done by:

PRTSUB=2
41,42;42,41

ENDPRT End of printer definition

5.2. Parameter file USERID.PAR

Purpose Hai*Basic needs to establish a fixed relationship between a HAI*Basic **user** and a terminal. It will automatically create a file named USERID.PAR for this purpose.

The user identifier file converts a user identifier to a unique HAI*Basic user number.

An example of its contents (for Unix) is:

```
/dev/console 1
/dev/tty00 2
/dev/tty01 3
```

Wildcard Wild card characters '*' and '?' are allowed.

The example:

```
/dev/tty?a 21-29
/dve/tty* 1-12,14,15
```

shows a Xenix system were the 14 virtual consoles have a user number from 1 to 12, 14 or 15 and any user conneced to serial ports 1 to 9 has a number from 21 to 29.

Not that in this last example, it is important the "tty?a" appears before "tty*" since the latter will also match any "tty?a".

Lan manager

HAI*Basic expects a unique user number for each task. When running DOS applications such as Microsoft Windows or other multitasking environments like DesqView, one number was given to one DOS user.

It is however possible to specify a range of numbers for one DOS user.

```
DOSCUMPUTER 16-20
```

This example assumes a Lan Manager station named 'DOSCOMPUTER' and allows up to 5 concurrent HAI*BAS tasks. The user number will all be in the range from 16 to 20. An attempt to run a 6th task will fail until on of the other exits.

Usernumbers on OS/2 networks

The HAI*BAS user number is obtained from the OS/2 screen number (or serial channel number) by the table in "userid.par".

When OS/2 workstations are used on a network there could be several stations with the same user number.

This problem is solved by prefixing the network "ComputerName" to the screen or serial channel number; the two are separated by a forward slash ("/"). An example of "userid.par" is then:

```
4 1
5 2
..
15 12
1001 13
1002 14
1003 15
MYCOMPUTER/4 16
MYCOMPUTER/5 17
..
MYCOMPUTER/15 27
MYCOMPUTER/1001 28
MYCOMPUTER/1002 29
MYCOMPUTER/1003 30
DOSCOMPUTER 31
```

This shows the default (stand-alone) table at the start, followed by the conversion table for workstation (or server) "MYCOMPUTER" and finally the entry for a DOS workstation "DOSCOMPUTER".

Serial port When HAI*Basic is started on a remote terminal then 1000 is added to the serial port number for conversion through USERID.PAR. The first port is COM1 so this will be 1001.

Note The filename 'USERID.PAR' is not fixed. It can be set via HAI.PAR, see USERID= option.

5.3. Systemfile HAISHARE

Purpose Keeps track of 'shared' files by the HAI*Basic users.
Keeps version number of Run Time System.

Incompatible versions

Incompatible versions of HAI*BAS should not be able to co-exist on a network.

Detection relies on ALWAYS starting HAI*BAS from the same directory.

Note that it is not possible to detect OLDER versions (before 5.65 (14Jun90), such as external release 5.23 (31May90)). In particular, updating index files simultaneously by HAI*BAS 4 and HAI*BAS 5 is almost guaranteed to corrupt the file.

Note The filename 'HAISHARE' is not fixed. It can be set via HAI.PAR, see HAISHARE= option.

6. FILE STRUCTURE

6.1. File overview

Host O.S. The HAI*Basic Run Time System is a subsystem running on the host operating system. It conforms to the facilities and conventions of the host operating system. At the same time it adds additional features if those features are not supported by the host operating system.

File types In DOS (and many other operating systems) files are conceptually merely a row of bytes. The internal organisation of those files is provided by the HAI*Basic file management system. The first 256 byte block of a HAI*Basic file contains all relevant information to define the file organisation. Moreover HAI*Basic has adopted strict naming conventions for those different file organisations.

Basic source file

The file name consists of 1 to 7 characters followed by **B.HIB** It contains a HAI*Basic source program.

Example: STARTB.HIB

Compiled code file

The file name consists of 1 to 7 characters followed by **C.HIC** It contains a compiled HAI*Basic program.

Example: PMENUC.HIC

Direct file The file name consists of 1 to 8 characters followed by **.HID** It contains fixed length records of at most 1018 bytes. The records are accessed by the order number. The first record has record number 0 or any other positive number to be specified at allocation time.

Example: STTXT.HID

Indexed file The file name consists of 1 to 8 characters followed by **.HIX** It contains fixed length records of at most 1018 bytes.

Example: WENUX.HIX

Ascii file The file name consists of 1 to 8 characters followed by **.HIA** or **.ASC** The .HIA file contains information in a special (HAI-) ascii format. It is used the merge HAI*Basic programs and a SPOOL file.

The .ASC file contains information in standard ascii format.

Example: SP001.HIA, TRACE1.ASC

Overlay file The file name consists of 5 characters followed by **x.HIO** It contains a machine code routine to be called from the HAI*Basic program. The x indicates the type of operating system the machine code applies to. This technique requires in depth knowledge of the Run Time System. Its use is only needed in very exceptional cases.

Example: SORB53.HIO (x=3, DOS)

Help files The file name consists of 5 characters followed by **H.HLP** It contains the in-contexts help in pure ascii format. By convention the first 5 characters of the help file are identical to the first five characters of the associated program file name.

Example: STARTH.HLP

System components

The files mentioned above have a HAI*Basic defined internal organistation. The Run Time System itself consists on files conforming to the conventions of the host operating system.

6.2. Direct files

File lay-out The records of all direct files are consecutively placed in a contiguouse area starting after the directory block.

Basic ascii, overlay and compiled code files have record length 1 by definition.

File extension

Basic and ascii files are automatically extended whenever necessary.

Direct files can be extended by a special option of the WRITE statement.

6.3. Indexed files

7. IN GENERAL

Function key codes

HAI*Basic function key codes as used in ACCEPT KEY values and ON KEY= event traps have a maximum value of 999, for example:

ACCEPT KEY=101

The key values are determined by HAI.PAR but the following recommendations should be noted:

1	..	20	are for existing functions keys and these must not be changed,
21	..	100	should be reserved,
101	..	200	are for the CUA*TOOL functions,
201	..	999	are reserved for possible future 'event' handling.

Keyboard buffer size

The keyboard buffer size is 256 characters. Any overflow is silently ignored; there is no longer any audible beep in this situation.

8. HELPFILES

9. UTILITIES

10. ERROR CODES

10.1. Introduction

Error classes

The HAI*Basic runtime system generates numeric codes for errors and exceptional conditions.

The error can be the result of:

- A correct, but 'unexpected' condition (e.g. file not found or record in use by another user in a multi-user system).
- Hardware malfunction.
- Diskette handling (e.g. device not ready, diskette unit not open).
- Programming error (e.g. incorrect array subscript).

The runtime error handling depends on the class of the condition.

I/O options

The HAI*Basic I/O options ERR=, EOF=, the function ERR and the statements ON ERROR GOTO, ON OVERFLOW GOSUB are available to handle the errors.

ERR= and ERR

HAI*Basic requires the ERR= option to be written without a space between ERR and the equal sign, in order to distinguish between the option ERR= and the function ERR.

10.2. Error handling

Printer The 'Printer not ready' message indicates a printer problem. Its exact text (in the proper language) is defined in the system parameter file HAI.PAR.
You should solve the printer problem and press the CLEAR key. The 'Printer no ready' message will reappear if the problem still exists.

If a 'Printer not ready' message occurs when OPENing the printer driver, you can press the ESCAPE key to abandon printer action. Error code 39 will then be generated and the appropriate action is taken.

System errors

System errors indicate an error or an exceptional condition. The exceptional conditions can be handled by the HAI*Basic program. The runtime system abandons the program if no proper action is foreseen. The program is always abandoned in case of errors. In this document errors and exceptional conditions are both referred as 'errors'.

The error handling depends on the error and other circumstances as defined below.

Error 0 If no ON OVERFLOW GOSUB statement is active, error 0 is displayed at the bottom line of the screen. The error can be CLEARed, but the incorrect results may cause problems in the program.
If an ON OVERFLOW GOSUB statement is active, control is passed to the statement number specified. The ERR function yields 0 (i.e. no error!).

Error 1 If no ERR= I/O option is specified, the system error message is displayed at the bottom line of the screen. Program execution continues after CLEARing the error.

If the ERR= I/O option is specified, control is passed to the statement number specified. The ERR function yields code 1.

Note: Error 1 indicates a successful retry when accessing disk files on previous HAI*Basic implementations. The error code 1 is now only used to indicate an error in the $DLK driver.

Errors 2-18 If no ERR= I/O option is specified, the system error message is displayed at the bottom line of the screen. The program is abandoned when CLEARing the error.
If the ERR= I/O option is specified, control is passed to the statement number specified. The ERR function yields the error code.

Error 19 If no ERR= or EOF= I/O option is specified, the system error message is displayed at the bottom line of the screen. The program is abandoned when CLEARing the error.
If only the ERR= I/O option is specified, control is passed to the statement number specified.
If only the EOF= I/O option is specified, control is passed to the statement number specified.
If both the ERR= and the EOF= I/O option are specified, control is passed to the statement number specified in the EOF= option.
The ERR function yields the code 19.

Errors 20-29 Same action as for errors 2-18

Errors 30-38 The system error message is always displayed at the bottom line of the screen.
If no ERR= I/O option is specified, the program is abandoned when CLEARing the error.
If the ERR= I/O option is specified, control is passed to the statement number specified.
The ERR function yields the error code.

Error 39 Same action as for errors 2-18

Errors 40-99 The system error message is always displayed. The program is abandoned when CLEARing the error.

Bleep The audible alarm in case of an error is optional. It is defined by the highest order bit of the default display attributes for system error messages (see the chapter on the parameter file HAI.PAR).

Display attributes
The display attributes of the error message are defined as defaults in the parameter file HAI.PAR.

Hard copy You can make a hardcopy of the display (with the error message) by pressing function key F6.

10.3. Error message format

10.4. HAI*Basic error codes

67 Indicates a invalid COMMON area handle.

74 Indicates a COMMON area handle mismatch; the handle specified for COMMON END is valid but it is not the current default.

83 Occurs in startup when the HAI*Basic user number is invalid. (i.e. not in the range 1 to 99).

84 Most commonly occurs in startup and indicates an incorrect installation. It is detected whenever a file is shared by 2 HAI*Basic users with the same number.

10.5. Start-up error codes