XBasic

Program Development Environment
(PDE)

Programming Language

Programmer Guide
Programmer Reference

Revision 0.0017
February 1, 1996

Table of Contents

Conventions

TRUE

Numeric variables and values not equal to zero are **`TRUE`** .

String variables with contents are **`TRUE`**, regardless of how many bytes they contain, and regardless of the values of the bytes. For example, the string variable **`a$`** is **`TRUE`** after any the following:

```
a$ = "0"
a$ = " "
a$ = "\0"
```

The last example illustrates that strings can contain *null characters*, characters with a value of **`0x00`**. Even a string containing one null character is not empty.

Arrays having any elements are **`TRUE`**, regardless of how many dimensions they have, or how many elements in any given dimension. Even an array with only one element is not empty and therefore **`TRUE`**. All of the array variables are made **`TRUE`** by the following statements:

```
DIM a[63]
DIM a[0]
DIM a$[63]
DIM a$[0]
```

FALSE

Numeric variables and values that equal zero are **`FALSE`**.

String variables with no contents are called *empty strings* and are **`FALSE`**. Strings start out empty, so until they are assigned values, they are **`FALSE`**. Strings become empty and **`FALSE`** when empty strings are assigned to them, as in **`a$ = ""`**.

Arrays with no elements are called *empty arrays* and are **`FALSE`**. Arrays start out empty, so they are **`FALSE`** until dimensioned. Arrays become empty when dimensioned or redimensioned to empty arrays, as in:

```
DIM a[]                ' a[] becomes empty
REDIM a[]              ' a[] becomes empty
```

or when *attached* elsewhere, as in

```
ATTACH a[] TO b[c,]    ' a[] becomes empty
```

TRUE *vs* $$TRUE

Don't confuse **`TRUE`** with the pre-defined constant **`$$TRUE`**, which has the numeric value `-1`.

```
  IF a THEN PRINT "a"
```

is not the same as

```
  IF (a = $$TRUE) THEN PRINT "a"
```

The first statement prints `"a"` as long as `a` is not zero. The second prints `"a"` only if `a` equals `-1`.

FALSE *vs* $$FALSE

Don't confuse **`FALSE`** with the pre-defined constant **`$$FALSE`**, which has a numeric value `0`.

```
  IFZ a$ THEN PRINT "a$ is empty"
```

is not the same as

```
  IF (a$ = $$FALSE) THEN PRINT "a$ is empty"
```

The second statement is a *type mismatch* since `a$` is a string while **`$$FALSE`** is a number.

Implicit **TRUE** *and* **FALSE**

The notion of **TRUE** and **FALSE** are implicit in many statements, including:

```
  IF, IFZ, CASE, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL.
```

The following examples illustrate the nature of **TRUE** and **FALSE** and how convenient they can be.

```
a  = 0                      ' a is FALSE because it's value is zero
b  = -3                     ' b is TRUE because it's value is non-zero
c  = 2                      ' c is TRUE because it's value is non-zero
a$ = ""                     ' a$ is FALSE because it has no contents
b$ = "0"                    ' a$ is TRUE because it has contents ("0")
c$ = " "                    ' a$ is TRUE because it has contents (" ")
d$ = "\0"                   ' a$ is TRUE because it has contents (1 byte)
DIM a[]                     ' a[] is FALSE because it has no contents
DIM b[0]                    ' b[] is TRUE because it has contents
DIM c[1]                    ' c[] is TRUE because it has contents
DIM d$[7]                   ' d$[] is TRUE because it has contents
'                           ' d$[0] is FALSE because it has no contents
'
IF a  THEN PRINT a          ' Nothing will print because a is FALSE
IF b  THEN PRINT b          ' "-1" will print because b is TRUE
IF c  THEN PRINT c          ' "2" will print because c is TRUE
IF a$ THEN PRINT "a$"       ' Nothing will print because a$ is FALSE
IF b$ THEN PRINT  b$        ' "0" will print because b$ is TRUE
IF c$ THEN PRINT "c$"       ' "c$" will print because c$ is TRUE
IF d$ THEN PRINT "d$"       ' "d$" will print because d$ is TRUE
IF a[] THEN PRINT "a[]"     ' Nothing will print because a[] is FALSE
IF b[] THEN PRINT "b[]"     ' "b[]" will print because b[] is TRUE
IF c[] THEN PRINT "c[]"     ' "c[]" will print because c[] is TRUE
IF d$[] THEN PRINT "d$[]"   ' "d$[]" will print because d$[] is TRUE
IF d$[1] THEN PRINT "YES"   ' Nothing will print because d$[1] is FALSE
'
DO WHILE b                  ' The loop will execute because b is TRUE
  INC b                     ' b = b + 1
  IF c THEN PRINT "hi"      ' "hi" will print because c is TRUE
LOOP WHILE a$               ' The loop will end because a$ is FALSE
'
SELECT CASE ALL TRUE        ' Do all CASEs that are true
  CASE a: PRINT "a"         ' Nothing will print because a is FALSE
  CASE b: PRINT "b"         ' "b" will print because b is TRUE
  CASE a, b: PRINT "a, b"   ' "a, b" will print because b is TRUE
  CASE a, a$: PRINT a, a$   ' Nothing will print because a and a$ are FALSE
  CASE a[]: PRINT "a[]"     ' Nothing will print because a[] is FALSE
  CASE b[]: PRINT "b[]"     ' "b[]" will print because b is TRUE
  CASE d$[]: PRINT "d$[]"   ' "d$[]" will print because d$ is TRUE
  CASE d$[e]: PRINT d$[e]   ' Nothing will print because d$[e] is FALSE
END SELECT
```

Language Elements

Character Set

Standard character sets contain 128 *standard characters*, numbered 0 to 127, encoded in one 8-bit byte. *Extended character sets* contain an additional 128 *extended characters*, numbered 128 to 255, encoded in the same 8-bit byte.

Source programs contain only the standard character set characters. Extended characters are discarded. All language elements are composed of printable standard characters, plus three *whitespace* characters, *space*, *tab*, *newline*. - which are have values **`0x20`**, **`0x09`**, **`0x0A`**.

To programs, however, characters are unsigned bytes. How these bytes are interpreted depends on the programs, though many built-in intrinsic functions assume the standard character set.

Certain groups of characters are referred to by the following names:

```
Alphabetic          "A - Z",  "a - z"
Alphanumeric        "A - Z",  "a - z",  "0 - 9"
Numeric             "0 - 9"
Binary              "0 - 1"
Octal               "0 - 7"
Hexadecimal         "0 - 9",  "A - F",  "a - f"
Symbol Characters   "A - Z",  "a - z",  "0 - 9"
Type Suffixes       @  @@  %  %%  &  &&  ~  !  #  $$  $
Scope Prefixes      #  ##
```

Parse Method

To *parse* means to break program text into language elements. For example, **`thisVar=thatVar`** parses into three language elements:

```
thisVar   - variable
=         - assignment operator
thatVar   - variable
```

The following process is performed to find each language element. First, leading whitespace is ignored. Then, successive characters are collected until adding the next character would produce an invalid language element.

Whitespace separates language elements that would otherwise be inappropriately combined into one. For example, **`FORK=ATOM`** means "assign variable **`ATOM`** to variable **`FORK`**". In contrast, most conventional BASIC languages interpret **`FORK=ATOM`** to mean **`FOR K = A TO M`**. To write the **`FOR`** statement requires the **`FOR`** and **`TO`** keywords be separated from the adjacent variables.

Case Sensitive

All keyword characters are upper case, and the language is case sensitive. Thus **`FOR`** is a keyword while **`for`** , **`foR`**, **`For`**, **`FOr`**, **`FoR`**, **`fOr`**, **`fOR`** are seven valid independent symbols, but not keywords. Thus the following is valid code, though no sane programmer would ever write it.

```
FOR For = foR TO to STEP Step      ' FOR, TO, STEP are keywords
  PRINT For                        ' For, foR, to, Step are variables
NEXT For                           ' Don't ever write code like this!
```

Names and Symbols

Names or *Symbols* are strings of one or more characters, beginning with an alphabetic character and including all subsequent characters up to the first non-symbol character. Characters that immediately follow symbols and constitute valid type-suffixes are considered part of the symbol and determine its data type. Characters that immediately precede symbols and constitute valid scope-prefixes are considered part of the symbol and determine its scope.

Type Suffixes

The type suffixes that can be appended to variables to explicitly specify their date type are:

@	SBYTE	8-bit signed byte integer
@@	UBYTE	8-bit unsigned byte integer
%	SSHORT	16-bit signed short integer
%%	USHORT	16-bit unsigned short integer
&	SLONG	32-bit signed long integer
&&	ULONG	32-bit unsigned long integer
~	XLONG	32/64-bit signed machine integer
$$	GIANT	64-bit signed giant (financial) integer
!	SINGLE	IEEE single precision floating point
#	DOUBLE	IEEE double precision floating point
$	STRING	String of unsigned bytes

Scope Prefixes

The scope prefixes that can be prepended to variables to explicitly specify their scope are:

#	SHARED	variable shared within a program
##	EXTERNAL	variable shared between multiple programs
$	Local Constant	constant visible within one function
$$	Shared Constant	constant visible throughout a program

Symbols

Most language elements are symbols. Language keywords symbols contain only upper-case characters, never a lower case character. Symbols begin with an alphabetic character followed by zero or more symbol characters, possibly terminated by a type-suffix or begun by a prefix, both of which become part of the symbol. Local and shared constants are symbols prefixed by **`$`** and **`$$`**, as in **`$PI`** and **`$$PIE`**.

Array names are always followed by square-brackets, though whitespace between the symbol and square brackets is permissible, so **`a$[j]`** and **`a$ [j]`** are equivalent. Function names are always followed by parentheses, though whitespace between the symbol and square brackets is permissible, so **`Func()`** and **`Func ()`** are equivalent.

Name Conventions

Symbols represent many kinds of language elements, yet each kind is always visually distinguishable. Arrays always have square brackets, functions always have parentheses, etc. Programs are easy to read because you can always tell what the elements of the program are.

Consistent, naming conventions also promote program readability. The following naming conventions are generally observed, hopefully by your programs too.

Keywords

All characters of keywords are upper case.
Examples : `IF DO FOR GOSUB FUNCTION DOUBLE STRING`

Type Names

Built-in data type names are upper case.
Examples : `SBYTE USHORT STRING DOUBLE DCOMPLEX`

User Defined aka Composite Type Names

User-defined data type names are upper case in most instances.
Examples : `COLOR WINDOW LENS SURFACE GLASS XWindowAttributes`

Variable Names

The first character is lower case and the 1st character of each imbedded word is upper case.
Examples : `value value$ thisValue thoseValues[]`

GOTO Label Names

The first character is lower case and the first character of imbedded words are upper case.
Examples : `label goHere goEveryWhichWay`

Subroutine Names

The first character is upper case and the 1st character of each imbedded word is upper case.
Examples : `Create CreateWindow CreateTrouble NukeWashingtonDC`

Function Names

The first character is upper case and the first character of each imbedded word is upper case.
Examples : `FuncName() Rotate() SingeTheUniverse()`

Reserved Names

Function and sharename symbols that begin with three consecutive **`x`** characters (**`Xxx`**, **`xxx`**, **`XXX`** etc) are reserved symbols. Do not create, call, or reference any such variables or functions. Also don't create functions with prefixes that conflict with known function libraries, like **`Xst`**, **`Xma`**, **`Xcm`**, **`Xgr`**, **`Xui`**, etc.

Source Lines
Source lines are separated by *newline* characters, labeled **`Enter`** on most keyboards. Statements and expressions are terminated by the end of source lines.

Line Names
Source lines can be given names, called *labels*. Line labels begin in the first character position on a line and are terminated by a **`:`** character, as in **`thisLabel:`**. Program execution can be transferred to line labels by **`GOTO`** statements, as in **`GOTO thisLabel`**.

Subroutine Names
Subroutines are named sections of functions that can be called from elsewhere in the function. Subroutines begin with **`SUB`** ***`SubName`*** and include all lines to the next **`END SUB`** statement. Subroutines are called by **`GOSUB`** statements, as in **`GOSUB SubName`**.

Comments
The **`'`** character begins a *comment*, except when it forms a valid character constant or appears in a literal string. The **`'`** and the rest of the source line are taken as a comment and have no affect on program operation, size, or speed. See *character literals* and *literal strings* for the exceptions.

Assignment
Statements that begin with a variable or array names are *assignment* statements, and must be followed by an **`=`** assignment operator. ***`typename`*****`AT()`** statements may also begin assignment statements, and must also be followed by an **`=`** assignment operator. See *Direct Memory Access* for **`typenameAT()`**.

Statements

Statements are keywords that specify an action to be performed, followed by language elements appropriate to the statement. In general, it is preferable to implement programming language capabilities with functions rather than statements. But statements have two advantages that are sometimes very important.

Since they do not involve function call overhead, statements execute quicker. Where the action performed is limited, the speed advantage of statements is significant. Decisions and execution control, like `IF`, `DO`, `FOR`, and `SELECT CASE`, execute much faster than functions.

Since statements are not constrained by function syntax, each statement can define its own syntax to make the action performed as clear and readable as possible. The flexibility and readability of the `FOR` statement is a good example.

Multiple statements on the same line are separated by `:` characters, as in `INC x : INC y : INC z`.

Certain statements may be preceded on lines only by whitespace. In general, these are statements that declare, define, begin, or end block structures. They include the following:

`DECLARE FUNCTION`	`Declare a module-shared function`
`INTERNAL FUNCTION`	`Declare a private function`
`EXTERNAL FUNCTION`	`Declare an external function`
`FUNCTION`	`Begin a function block`
`END FUNCTION`	`End a function block`
`EXTERNAL`	`Declare external variables`
`SHARED`	`Declare shared variables`
`STATIC`	`Declare static variables`
`AUTOX`	`Declare autox variables`
`AUTO`	`Declare auto variables`
`DO`	`Begin a DO loop`
`LOOP`	`End a DO loop`
`FOR`	`Begin a FOR loop`
`NEXT`	`End a FOR loop`
`SELECT CASE`	`Begin a SELECT CASE block`
`CASE`	`Check another CASE`
`END SELECT`	`End a SELECT CASE block`
`SUB`	`Begin a subroutine`
`END SUB`	`End a subroutine`
`TYPE`	`Begin a type definition`
`UNION`	`Begin a union definition`
`END TYPE`	`End a TYPE definition`
`END UNION`	`End a UNION definition`

Intrinsics

Intrinsics, short for *intrinsic functions*, are often-called functions like `ABS()`, `INT()`, `LEFT$()` that are built into the language and thus always callable without declaration or importing a function library.

Intrinsics take one or more arguments which are not changed, and return a value. The names of intrinsics are fully capitalized keyword symbols. Intrinsics execute quickly, and some handle variable number of arguments. The intrinsics are described in detail in the reference manual.

Operators

Like algebraic operators, programming language *operators* perform common operations on one or more *operands*. Arithmetic, logical, bitwise, and address operators are provided. Each has a precedence, and each is a member of a class that determines its valid operand types, type of result, and conversions rules.

Unary Operators

Unary operators operate on a single data object or expression to the right of the unary operator. For example, `-` is the common negative operator, used to negate arithmetic sign, as in `-x` or `-ABS(x+y)`. Unary operators have the highest precedence, so unary operators are always executed before adjacent binary operators. Adjacent unary operators execute from right to left.

Binary Operators

Binary operators combine two operands into a single value. When the operands are of different data type, the operand with the lower data type is *promoted* aka *converted* to the higher data type before the operation is performed.

Operator Precedence

As in algebra, operators have *precedence*. Operators with higher precedence are executed before adjacent operators, even when they appear later in an expression.

In `a+b*c`, the `b*c` is performed first, then added to `a`. Precedence limits the need for parentheses to group sub-expressions. Parentheses can make natural execution order more visible, and override natural execution order when desired.

For example, `a+(b*c)` operates the same as the previous example, while `(a+b)*c` forces `a+b` to occur first, the result of which is then multiplied by `c`.

Operator Kind

Arithmetic operators are the kind usually encountered in algebra. They combine numeric operands and produce a numeric result.

Bitwise operators combine numeric integer operands and produce a numeric integer result, but operate on a bit-by-bit basis, without carry/borrow propagation from bit to bit.

Logical operators combine numeric or string operands, and produce a logical result, meaning `$$TRUE (-1)`, or `$$FALSE (0)`.

Operator Class

Operator class determines certain aspects of how operators behave.

Class 1

Class 1 operators include binary logical operators **&&**, **^^**, **||**.

The operands must be integer or floating point variables or expressions. The result is always **XLONG $ $TRUE** or **$$FALSE**.

Class 2

Class 2 operators include the binary relational operators **=**, **<>**, **<**, **<=**, **>=**, **>** and their equivalents **==**, **!=**, **!>=**, **!>**, **!<**, **!<=**.

The operands must be integer, floating point, or string. If one operand is a string, so must the other. String characters are compared until a byte differs or the end of one string is reached, so the effect is alphabetic comparison. The result is always **XLONG $$TRUE** or **$$FALSE**.

Class 3

Class 3 operators include binary bitwise operators **AND**, **XOR**, **OR**, and symbolic equivalents **&**, **^**, **|**.

The operands must be integer variables or expressions. The result is the operand data type.

Class 4

Class 4 operators include binary arithmetic operators **+**, **-**, *****, **/**, ******.

The operands must be integer or floating point variables or expressions. The result is the data type of the highest type operand.

Class 5

Class 5 operators include the binary arithmetic operator **+** and the string concatenate operator **+**.

The operands must both be integer or floating point variables or expressions, or both be string variables or expressions. The result is the operand data type.

Class 6

Class 6 operators include the binary arithmetic operators **MOD**, **/**.

The operands must be integer or floating point variables or expressions. If either operand is **GIANT**, the other operand is converted to **GIANT** and the result is **GIANT**. Otherwise, if either operand is **XLONG**, the other operand is converted to **XLONG** and the result is **XLONG**. Otherwise, if either operand is **ULONG**, the other operand is converted to **ULONG** and the result is **ULONG**. Otherwise, both operands are converted to **SLONG** and the result is **SLONG**. Note that this means that **SINGLE** and **DOUBLE** operands are converted to one integer type or another before the operation takes place. *Class 6* operators are *integer* operators.

Class 7

Class 7 operators include the binary arithmetic and bitwise shift operators **<<**, **>>**, **<<<**, **>>>>**.

The operands must be integer variables or expressions. The result is the data type of the left operand.

Class 8

Class 8 operators include the *unary* arithmetic operators **`+, -`**.

The operand must be an integer or floating point expression. The result is the data type of the operand.

Class 9

Class 9 operators include the unary logical operators **`!`**, **`!!`**.

The operand must be an integer or floating point expression. The result is always **`$$TRUE`** or **`$$FALSE`**.

Class 10

Class 10 operators include the unary bitwise operators **`~`** and **`NOT`**.

The operand must be an integer variable or expression, or a function name. When applied to **`GIANT`** operands, the result data type is **`GIANT`**. Otherwise the result data type is **`XLONG`**.

Class 11

Class 11 operators include the unary address operators **`&`** and **`&&`**.

The operand of **`&`** must be a variable, string, composite, whole array, array node, or array data element. The operand of **`&&`** must be a string, whole array, or string in a string array. The result data type is always **`XLONG`**.

Operator Summary

The following table is a summary of the characteristics of all operators recognized by the language.

OP	ALT	KIND	CLASS	OPERANDS	RETURNS	PREC	COMMENTS
&		unary	11	AnyType	Address	12	Address of Object Data
&&		unary	11	AnyType	Address	12	Address of Object Handle
NOT	~	unary	10	Integer	SameType	12	Bitwise NOT
!		unary	9	Numeric	T/F	12	Logical Not (TRUE if 0, else FALSE)
!!		unary	9	Numeric	T/F	12	Logical Test (FALSE if 0, else TRUE)
+		unary	8	Numeric	SameType	12	Plus
-		unary	8	Numeric	SameType	12	Minus
>>>		binary	7	Integer	LeftType	11	Arithmetic Up Shift
<<<		binary	7	Integer	LeftType	11	Arithmetic Down Shift
<<		binary	7	Integer	LeftType	11	Bitwise Left Shift
>>		binary	7	Integer	LeftType	11	Bitwise Right Shift
**		binary	4	Numeric	HighType	10	Power
/		binary	4	Numeric	HighType	9	Divide
*		binary	4	Numeric	HighType	9	Multiply
\\		binary	6	Numeric	Integer	9	Integer Divide
MOD		binary	6	Numeric	Integer	9	Modulus (Integer Remainder)
+		binary	5	Numeric	HighType	8	Add
+		binary	5	String	String	8	Concatenate
-		binary	4	Numeric	HighType	8	Subtract
AND	&	binary	3	Integer	HighType	7	Bitwise AND
XOR	^	binary	3	Integer	HighType	6	Bitwise XOR
OR	|	binary	3	Integer	HighType	6	Bitwise OR
>	!<=	binary	2	NumStr	T/F	5	Greater-Than
>=	!<	binary	2	NumStr	T/F	5	Greater-Or-Equal
<=	!>	binary	2	NumStr	T/F	5	Less-Or-Equal
<	!>=	binary	2	NumStr	T/F	5	Less-Than
<>	!=	binary	2	NumStr	T/F	4	Not-Equal
=	==	binary	2	NumStr	T/F	4	Equal (also "!<>")
&&		binary	1	Integer	T/F	3	Logical AND
^^		binary	1	Integer	T/F	2	Logical XOR
||		binary	1	Integer	T/F	2	Logical OR
=		binary		NumStr	RightType	1	Assignment
					T/F		T/F always returned as XLONG

Unary Address Operators

The unary address operators are **`&`** and **`&&`**.

`&`

`&` returns the *memory address* of the following variable, array, array node, array data element, or composite element. Applying **`&`** to numeric **`AUTO`** variables may produce compile-time "Bad Scope" errors because **`AUTO`** variables may be assigned space in CPU registers, which do not have addresses. String and composite variables are always located in memory, so applying **`&`** to strings and composites is always valid. The valid forms of **`&`** are:

```
&numeric-variable         &count
&string-variable          &name$
&whole-array              &token[]
&whole-string-array       &symbols$[]
&array-node               &token[func, ]
&array-data               &token[func, line, element]
&string-array-node        &name$[dept, ]
&string-array-data        &name$[dept, stationNumber]
```

`&&`

`&&` returns the *handle address* of the following string variable , composite variable, whole array, or string array element. Numeric variables and components of composite variables do not have handles, so applying **`&&`** to them produces compile-time errors. Applying **`&&`** to **`AUTO`** strings, arrays, and composites produces compile-time errors because they may be assigned space in CPU registers, which do not have addresses. The valid forms of **`&&`** are:

```
&&string-variable (non-AUTO)          &&name$
&&whole-array (non-AUTO)              &&token[]
&&whole-string-array (non-AUTO)       &&symbols$[]
&&string-array-data                   &&name$[dept, stationNumber]
...(same result as above)             &name$[dept, stationNumber,]
```

Unary Arithmetic Operators

The unary arithmetic operators are **`+`** and **`-`**.

`+`

`+`, when used as a unary operator, is the *unary positive* operator, which performs no operation on the following operand. It makes sign visible and explicit where appropriate for program clarity.

`-`

`-`, when used as a unary operator, is the *unary negative* operator which changes the sign of the following operand. The sign bit of the operand value is inverted, or the operand value is subtracted from zero, whichever is appropriate for the operand data type.

Unary Bitwise Operators
The unary bitwise operators are `~` aka `NOT`.

`~` *aka* **`NOT`**

`~` and `NOT` return the *bitwise inversion* of the following operand. All bits are flipped, so every zero bit is made a one, and every one bit is made a zero.

Unary Logical Operators
The unary logical operators are `!` and `!!`.

`!`

`!` returns the *logical NOT* of the following operand. If the value of the operand is zero, `!` returns `$$TRUE (-1)`. If the value is any non-zero value, `!` returns `$$FALSE (0)`.

`!!`

`!!` returns the *logical TEST* of the following operand. If the value of the operand is zero, `!!` returns `$$FALSE (0)`. If the value is any non-zero value, `!!` returns `$$TRUE (-1)`.

Binary Shift Operators
The binary shift operators are >>>, <<<, >>, <<.

>>>

>>>, the *arithmetic shift right* operator, shifts the value of the left integer operand n bits to the right, where n is the value of the integer operand following the >>> operator. When the left operand is unsigned, all vacated upper bits are filled with zeros. When the left operand is signed, all vacated upper bits are filled with the most significant bit of the original value. >>> is explicitly arithmetic.

<<<

<<<, the *arithmetic shift left* operator, shifts the value of the left integer operand n bits to the left, where n is the value of the integer operand to the right of the <<< operator. All vacated lower bits are filled with zeros, regardless of data-type. <<< is explicitly arithmetic. The data type of the result is the same as the shifted integer; if significant bits are shifted out of the integer, they are lost and no error occurs.

>>

>>, the *bitwise shift right* operator, shifts the left integer operand n bits to the right, where n is the value of the integer operand to the right of the >> operator. All the vacated upper bits are filled with zeros. >> is explicitly bitwise.

<<

>>, the *bitwise shift left* operator, shifts the left integer operand n bits to the left, where n is the value of the integer operand to the right of the << operator. All the vacated lower bits are filled with zero bits. << is explicitly bitwise. The data type of the result is the same as the shifted integer; if significant bits are shifted out of the integer, they are lost and no error occurs.

Binary Arithmetic Operators

The binary arithmetic operators are `**`, `/`, `*`, `-`, `+`, `\`, `MOD`.

`**`

`**`, the *raise to power* operator, raises the left operand to the power of the right operand. `**` replaces the `^` in conventional BASIC because `^` is bitwise `XOR`.

`/`

`/`, the *floating point divide* operator, divides the left operand by the right operand. `/` is a floating point divide, so if either operand is an integer type, it is converted to `DOUBLE` before the operation is performed. The result of `/` is `SINGLE` if both operands are `SINGLE`, otherwise it is `DOUBLE`. `/` will also divide `SCOMPLEX` and `DCOMPLEX` numbers.

`*`

`*`, the *multiply* operator, multiplies the left operand by the right operand. `*` will also multiply `SCOMPLEX` and `DCOMPLEX` numbers.

`-`

`-`, the *subtract* operator, subtracts the right operand from the left operand. `-` will also subtract `SCOMPLEX` and `DCOMPLEX` numbers.

`+`

`+`, the *add* operator, adds the right numeric operand to the left numeric operand. `+` will also add `SCOMPLEX` and `DCOMPLEX` numbers. `+` will also concatenate strings - see *Binary String Operators*.

`\`

`\`, the *integer divide* operator, divides the left operand by the right operand. `\` is an integer divide, so if either operand is a floating point type, it is converted to `SLONG` before the operation is performed. The result of `\` is always `SLONG` or `XLONG` or `GIANT`.

`MOD`

`MOD`, the *integer modulus* operator, divides the left operand by the right operand, but returns the remainder left over from the divide instead of the result of the divide. `MOD` is defined to use an integer divide, so if either operand is a floating point type, it is converted to `SLONG` before the operation is performed. The result of `MOD` is always `SLONG` or `XLONG` or `GIANT`.

Binary Bitwise Operators

The binary bitwise operators are `&`, `|`, `^`, aka **`AND`**, **`OR`**, **`XOR`**.

`&` *aka* **`AND`**

`AND` and `&`, the *bitwise AND* operators, **`AND`** the left and right operands on a bit by bit basis.

`|` *aka* **`OR`**

`XOR` and `^`, the *bitsize XOR* operators, **`XOR`** the left and right operands on a bit by bit basis.

`^` *aka* **`XOR`**

`OR` and `|`, the *bitwise OR* operators, **`OR`** the left and right operands on a bit by bit basis.

Binary Logical Operators

The binary logical operators are `&&`, `^^`, `||`.

`&&`

`&&`, the *logical AND* operator, returns **`XLONG $$TRUE`** if *both* the left and right operand is non-zero; otherwise it returns **`$$FALSE`**.

`^^`

`^^`, the *logical XOR* operator, returns **`XLONG $$TRUE`** if *either one but not both* of the left and right operands is non-zero; otherwise it returns **`$$FALSE`**.

`||`

`||`, the *logical OR* operator, returns **`XLONG $$TRUE`** if *either one or both* of the left and right operands is non-zero; otherwise it returns **`$$FALSE`**.

Binary String Operators

The only binary string operator is `+`.

`+`

`+`, the *string concatenate* operator, appends the right string operand to the end of the left string operand. `+` also adds numeric and complex values; see *Binary Arithmetic Operators*.

Binary Relational Operators

The binary relational operators are `>`, `>=`, `<=`, `<`, `<>`, `=`, aka `!<=`, `!<`, `!>=`, `!=`, `==`.

> *aka* **!<=**

`>` and `!<=` return `$$TRUE` if the left operand is greater than the right operand; otherwise they return `$$FALSE`.

>= *aka* **!<**

`>=` and `!<` return `$$TRUE` if the left operand is greater or equal to the right operand; otherwise they return `$$FALSE`.

<= *aka* **!>**

`<=` and `!>` return `$$TRUE` if the left operand is less than or equal to the right operand; otherwise they return `$$FALSE`.

< *aka* **!>=**

`<` and `!>=` return `$$TRUE` if the left operand is less than the right operand; otherwise they return `$$FALSE`.

<> *aka* **!=**

`<>` and `!=` return `$$TRUE` if the left operand is not equal to the right operand; otherwise they return `$$FALSE`.

= *aka* **==**

= and == return `$$TRUE` if the left operand is equal to the right operand; otherwise they return `$$FALSE`.

Operator Considerations

Conventional BASIC does not have logical operators: `&&` `^^` `||` `!` `!!`

In conventional BASIC, the bitwise operators `AND`, `XOR`, `OR`, `NOT` are used in place of the corresponding logical operators, sometimes with undesirable results. For example, if `x=1` and `y=2`, then `x AND y` returns zero, which is `FALSE`, while `x&&y` returns `TRUE`. Furthermore, `x XOR y` returns `3`, which is `TRUE`, while `x^^y` returns `FALSE`.

Subroutines

Subroutines are blocks of code that can be called from elsewhere in the same function. Subroutines do not take arguments, but have access to all variables and constants available in that function.

Subroutines begin with a **`SUB`** ***`SubName`*** statement, end with an **`END SUB`** statement, and are called with a **`GOSUB`** ***`SubName`*** statement. It is an error to attempt to **`GOTO`** labels within a subroutine from outside the subroutine. However, it is possible to **`GOTO`** a label outside a subroutine from within.

Subroutines cannot be nested. Each must end before another begins.

Subroutines are not recursive. They may not call themselves directly or indirectly. A compile-time error occurs if a subroutine tries to call itself directly, that is, from within itself. However a subroutine can call itself indirectly by calling another subroutine that calls it. This error is not detected at compile-time or run-time. Programs may hang up when subroutines call themselves indirectly.

But functions are recursive, so a function can call itself from within a subroutine without restrictions. For example, **`SUB Blivit`** in function **`ThisFunc()`** may not contain **`GOSUB Blivit`**. But function **`ThisFunc()`** may be called from within **`SUB Blivit`**, and the newly called **`ThisFunc()`** may execute **`SUB Blivit`**. Stated differently, the prohibition against recursive execution of subroutines applies only within each call of the function that contains it.

Functions

Functions are blocks of code that can be called from anywhere in a program, receive 0 to 16 input values called arguments, carry out actions based upon the input values, and finally return control to the point in the program immediately following the call, sometimes returning a value.

Arguments can be passed to functions by value, by reference, or by address (for calling C functions). Numeric values, string values, arrays, and composite values can be passed to functions.

Block Structure

Programs are composed of *blocks* which look and act like units. Most block structures can be *nested*, which means located inside other blocks. To promote clarity, readability, and visibility of program structure, source lines in block structures should be indented by a tab or two spaces.

Functions are the outermost blocks. Within functions, five block structures are common:

- `IF ... END IF`
- `SELECT CASE ... END SELECT`
- `DO ... LOOP`
- `FOR ... NEXT`
- `SUB ... END SUB`

Except for functions and subroutines, blocks can be nested. It is common to find these blocks nested several levels deep in many programs.

Each block must fully enclose all blocks within it. In other words, a block must end before any block in which it is imbedded. This does not mean that execution necessarily proceeds through blocks in the order of the source program. Statements exist to control execution within and between blocks, and for multi-level exits.

The reference section describes statements that control execution in and between blocks, including:

- `DO DO`
- `DO LOOP`
- `DO FOR`
- `DO NEXT`
- `EXIT DO`
- `EXIT IF`
- `EXIT FOR`
- `EXIT FUNCTION`
- `EXIT SUB`
- `EXIT SELECT`
- `NEXT CASE`

Execution Order

Execution need not proceed linearly through blocks. **`DO`** ***`xxx`*** and **`EXIT`** ***`xxx`*** statements control the execution path through blocks. For example, it is perfectly acceptable to do the next iteration of a `DO` loop by executing a `DO DO` or `DO LOOP` statement, even if there are other blocks between the ends of the `DO` block and the `DO DO` or `DO LOOP`.

Multi-Level Control

Multi-level **`DO`** ***`xxx #`*** and **`EXIT`** ***`xxx #`*** statements are provided. For example, `EXIT DO 2` escapes two levels of `DO` loops instead of just one, `DO DO 3` jumps to the `DO` statement three levels back, and `DO LOOP 1` is the same as `DO LOOP`.

If you modify code near multi-level statements, be careful to adjust the `#` level counts as necessary.

Data

Data Type

A number of *data types* are built in, including signed and unsigned integers, single and double precision floating point numbers, single and double precision floating point complex numbers, and strings. Additional *user-defined types* can be defined in programs. These *composite types* are fixed-size collections of built-in and other user-defined types.

The type suffix, type name, and format of the built-in data types are given in the following table:

`@`	**SBYTE**	**Signed byte (8-bits)**
`@@`	**UBYTE**	**Unsigned byte (8-bits)**
`%`	**SSHORT**	**Signed short (16-bits)**
`%%`	**USHORT**	**Unsigned short (16-bits)**
`&`	**SLONG**	**Signed long (32-bits)**
`&&`	**ULONG**	**Unsigned long (32-bits)**
	XLONG	**Natural long (32/64-bits)**
	GOADDR	**GOTO address (32/64-bits)**
	SUBADDR	**GOSUB address (32/64-bits)**
	FUNCADDR	**FUNCTION address (32/64-bits)**
`$$`	**GIANT**	**Signed giant (64-bits)**
`!`	**SINGLE**	**IEEE Single Precision Floating Point (32-bits)**
`#`	**DOUBLE**	**IEEE Double Precision Floating Point (64-bits)**
`$`	**STRING**	**String of unsigned bytes (characters)**
	SCOMPLEX	**Single Precision Complex (Two SINGLEs)**
	DCOMPLEX	**Double Precision Complex (Two DOUBLEs)**

Type Suffix

Type suffixes make the data types of variables instantly visible, but are not required. They can be appended to variables and arrays to specify data type when type visibility is important.

Simple Type

Integer and floating point types are called *simple types*, because they contain a single element, a number.

String

Strings are sequences of unsigned bytes normally used to hold characters. Strings are very common in most programs, so special capabilities make string programming faster and more convenient.

Composite Type

Composite types are collections of simple types, composite types, fixed-size strings, and fixed-size one dimensional arrays of any of these. Two complex number data types, **`SCOMPLEX`** and **`DCOMPLEX`**, are built in composite types. Additional composite types defined in the prolog of programs, and are called *user-defined types*.

Built In Types

The following table gives the name, size, format, type suffix, and the minimum and maximum value of every built in data type.

SUFFIX	BITS	NAME	FORMAT	MIN VALUE	MAX VALUE
@	8	SBYTE	Signed Byte Integer	-128	+127
@@	8	UBYTE	Unsigned Byte Integer	0	+255
%	16	SSHORT	Signed Short Integer	-32768	+32767
%%	16	USHORT	Unsigned Short Integer	0	+65535
&	32	SLONG	Signed Long Integer	-2147483648	+2147483647
&&	32	ULONG	Unsigned Long Integer	0	+4294967395
	32/64	XLONG	Natural Long Integer	MIN SLONG / GIANT	MAX SLONG / GIANT
	32/64	GOADDR	Computed GOTO address	MIN XLONG	MAX XLONG
	32/64	SUBADDR	Computed GOSUB address	MIN XLONG	MAX XLONG
	32/64	FUNCADDR	Computed function address	MIN XLONG	MAX XLONG
$$	64	GIANT	Signed Giant Integer	-9223372036854775808	+9223372036854775807
!	32	SINGLE	IEEE Single Floating Point	-1e38	+1e38
#	64	DOUBLE	IEEE Double Floating Point	-1d308	+1d308
$		STRING	Unsigned Byte String	zero characters	2147483647 characters
	64	SCOMPLEX	IEEE Single Complex	1e38 : -1e38	+1e38 : +1e38
	128	DCOMPLEX	IEEE Double Complex	1d308 : -1d308	+1d308 : +1d308

Coersion aka Type Conversion

Type conversion, also called *coersion*, is performed automatically when appropriate. For example, when an operator combines two operands in expression evaluation, the smaller type is *promoted* to the larger type before the operation is performed.

A consistent set of intuitive, efficient intrinsics for explicit type conversion are also provided. For all explicit type conversions, the name of the intrinsic is the name of the data type to convert to, as the following table illustrates. The argument can be any built-in simple (numeric) or string type.

SBYTE()	**Convert to SBYTE**
UBYTE()	**Convert to UBYTE**
SSHORT()	**Convert to SSHORT**
USHORT()	**Convert to USHORT**
SLONG()	**Convert to SLONG**
ULONG()	**Convert to ULONG**
XLONG()	**Convert to XLONG**
GOADDR()	**Convert to GOADDR**
SUBADDR()	**Convert to SUBADDR**
FUNCADDR()	**Convert to FUNCADDR**
GIANT()	**Convert to GIANT**
SINGLE()	**Convert to SINGLE**
DOUBLE()	**Convert to DOUBLE**
STRING()	**Convert to STRING**
STRING$()	**Ditto**

Type Sizes

XLONG, **GOADDR**, **SUBADDR**, and **FUNCADDR** data types are 32-bit on some CPUs and 64-bit on others. These *natural integer* and address types are 64-bit if the *logical* address of the computer CPU is 64-bits. This does not alter how properly written programs operate. To assure portability, avoid writing programs that depend on the size of these data types being 32-bits or 64-bits.

Storage Type

The values of individual integer variables smaller than **SLONG** are held in memory as **SLONG** values. Therefore overflow does not occur for integer types until an overflow of the **SLONG** data type occurs. Array elements, on the other hand, are always held as their specified data type.

Range checking is *not* performed when values are assigned to array elements because the overhead is considerable and in most cases the nature of the program avoids truncation or overflow. In cases where assigning an integer variable to an array might result in undesired truncation, apply a type conversion intrinsic to the result before assigning to the array.

The conversion intrinsics range check their arguments and cause overflow errors if the value is out of range for the specified result type. For example, the following two lines assign an expression value to **SSHORT** array **k[]** without range checking, then with range checking:

```
k%[n] = a * b + c * d              ' no range checking
k%[n] = SSHORT (a * b + c * d)     ' do range checking
```

Kind

There are three principal *kinds* of data objects, *literals*, *constants*, and *variables*.

Literal

A *literals* is a specific numeric or string value. **`23`** is a numeric literal, while **`"hello"`** is a string literal. The many literal formats supported are described in the following pages.

Numeric Literal

A *numeric literal* is a specific numeric value represented in one of the following formats:

Character Literal	**`'x'`**	ASCII character between single quotes.
Integer Literal	**`88110`**	Decimal digits.
Decimal Literal	**`88.110`**	Decimal digits plus and decimal point.
Scientific Literal	**`.88110d+5`**	Decimal number with power of 10 exponent
Hexadecimal Literal	**`0xDEADC0DE`**	"**`0x`**" plus 0 to 16 hexadecimal digits.
Octal Literal	**`0o37777777777`**	"**`0o`**" plus 0 to 22 octal digits.
Binary Literal	**`0b0010100010101011`**	"**`0b`**" plus 0 to 64 binary digits.
SINGLE Image	**`0s3F880000`**	"**`0s`**" plus exactly 8 hexadecimal digits.
DOUBLE Image	**`0d4018000080000000`**	"**`0d`**" plus exactly 16 hexadecimal digits.

Character Literal

Character literals are single ASCII characters enclosed in single quotes, like **`'x'`** or **`'!'`**. Simple *backslash* codes like **`'\a'`**, **`'\n'`**, **`'\V'`**, **`'\\'`**, and **`'\"'`** are also valid representations of non-printing characters. The valid backslash codes are the *single character* backslash codes defined for string literals. Character literals are unsigned bytes **`UBYTE`**.

Character literals are a convenient, efficient way to specify the numeric value of any single ASCII character. **`'a'`**, **`'!'`**, and **`'"'`** are equivalent to **`ASC("a")`**, **`ASC("!")`**, and **`ASC(CHR$(34))`**, but execute much faster. Remember, character literals represent the ASCII value of characters, so **`'5'`** does not represent the number 5, but 0x35 or 53. See Appendix A for character values of ASCII characters. The following code segment illustrates character literals:

```
FUNCTION CheckChars (n$)                 '
  FOR i = 1 TO LEN(n$)                   ' for each character in n$
    v = ASC(n$, i)                       ' v = value of character #i
    SELECT CASE TRUE                     '
      CASE (v >= 'A') AND (v <= 'Z')     : PRINT "Upper case letter."
      CASE (v >= 'a') AND (v <= 'z')     : PRINT "Lower case letter."
      CASE (v >= '0') AND (v <= '9')     : PRINT "Decimal digit."
      CASE (v  = '.')                    : PRINT "Decimal point."
      CASE (v  = '$')                    : PRINT "Dollar sign."
      CASE (v  = '\t')                   : PRINT "Tab character."
      CASE (v  = '\\')                   : PRINT "Backslash character."
      CASE ELSE                          : PRINT "Nothing interesting."
    END SELECT
  NEXT i
END FUNCTION
```

Integer Literal

Integer literals are series of decimal digits without decimal point or scientific exponent. All but huge integers (more than 19 digits) can be stored in one of the integer data types **`SBYTE`** to **`GIANT`**.

Values from **`-9,223,372,036,854,775,808`** to **`+9,223,372,036,854,775,807`** can be represented in at least one of the integer formats. Numeric literals are stored in the smallest type that can hold them. The ranges of the integer data types are as follows:

SBYTE	`-128` to `+127`
UBYTE	`0` to `+255`
SSHORT	`-32,768` to `+32,767`
USHORT	`0` to `+65,535`
SLONG	`-2,147,483,648` to `+2,147,483,647`
ULONG	`0` to `+4,294,967,295`
XLONG	Same as **`SLONG / GIANT`** on 32-bit / 64-bit systems
GIANT	`-9,223,372,036,854,775,808` to `+9,223,372,036,854,775,807`

Decimal Literal

Decimal literals are series of decimal digits with a decimal point, but no scientific exponent. Integer data types cannot store the fractional part of numbers, so decimal literals become one of the two floating point data types, **`SINGLE`** or **`DOUBLE`**.

Positive or negative numbers with exponents larger than about 10^{38} or smaller than about 10^{-38} cannot be held in **`SINGLE`**, so they are typed **`DOUBLE`**.

Numbers that are more precisely represented in **`DOUBLE`** than **`SINGLE`** are typed **`DOUBLE`**. Numbers with more than 7 significant decimal digits can always be represented more precisely in **`DOUBLE`**, as can many shorter numbers. For example, `.1` is typed **`DOUBLE`** because floating point representations of `.1` are binary continuing fractions (like 1/3 in decimal), and **`DOUBLE`** stores more of the fraction.

Decimal literals can be explicitly typed **`SINGLE`** or **`DOUBLE`** appending a type-suffix character, as in **`.100!`** or **`.100#`**.

Scientific Literal

Scientific literals are decimal digits, with or without a decimal point, followed by a scientific exponent. The format of a valid scientific exponent is **`{d|e|D|E}[+/-]XXX}`** where **`XXX`** is 1 to 3 decimal digits. **`e`** and **`E`** exponent specifiers identify numbers as **`SINGLE`**, while **`d`** and **`D`** specify **`DOUBLE`**.

`-3e-2` is **`-0.03`** in **`SINGLE`**, while **`88.110d3`** is **`88110`** in **`DOUBLE`**. The range of **`SINGLE`** and **`DOUBLE`** data types are:

```
SINGLE   < 0    -3.402823e+38 to -1.175494e-38
SINGLE   = 0     0
SINGLE   > 0    +1.175494e-38 to +3.402823e+38

DOUBLE   < 0    -1.79769313486232d+308 to -2.22507385850719d-308
DOUBLE   = 0     0
DOUBLE   > 0    +2.22507385850719d-308 to +1.79769313486232d+308
```

Hexadecimal Literal

Hexadecimal literals begin with `0x` and are followed by 0 to 16 hexadecimal digits `(0-9,A-F,a-f)`. Hexadecimal literals are *images* of 32-bit or 64-bit integers in 4-bit chunks.

0 to 8 hex digit literals are typed `XLONG`, while 9 to 16 hex digit literals are typed `GIANT`. Hexadecimal literals with less than 8 hex digits have implicit zeros in the more significant digit positions, so `0xFEED` is equivalent to `0x0000FEED`. Therefore, 8 hex digits are required to specify negative numbers. For example, the value `-1` is `0xFFFFFFFF`, not `0xFFFF` (+65535) or `0xFF` (+255).

Similarly, 16 hex digits are required to specify negative numbers in the `GIANT` hexadecimal format. When 16 hex digits have been collected, the hexadecimal literal is complete and any further characters, whether hex digits or not, are not counted as being part of the hexadecimal literal.

`0x` followed by an invalid hexadecimal character has a value of zero. Type suffixes are invalid following hexadecimal literals.

Octal Literal

Octal literals begin with `0o` and are followed by 0 to 22 octal digits `(0-7)`. Octal literals are *images* of 32-bit or 64-bit integers in 3-bit chunks.

0 to 11 octal digit literals are typed `XLONG`, while 12 to 22 octal digit literals are typed `GIANT`. The one exception is an 11 octal digit number with the most significant bit set, which is type `GIANT`. Octal literals with less than 11 octal digits have implicit zeros in the more significant digit positions, so `0o7777` is equivalent to `0o00007777`. This means that 11 octal digits are required to specify negative numbers. For example, the value `-1` is `0o37777777777`.

Similarly, 22 octal digits are required to specify negative numbers in the `GIANT` octal format. When adding another octal digit would cause significant bits to be lost (shifted out most significant end), further characters, whether valid octal digits or not, are not counted as being part of the octal literal.

`0o` followed by an invalid octal character has a value of zero. Type suffixes are invalid following octal literals.

Binary Literal

Binary literals begin with `0b` and are followed by 0 to 64 binary digits `(0-1)`. Binary literals are *images* of 32-bit or 64-bit integers in 1-bit chunks.

0 to 32 binary digit literals are typed `XLONG`, while 33 to 64 binary digit literals are typed `GIANT`. Binary literals with less than 32 significant binary digits have implicit zeros in the more significant digit positions, so `0b1111` is equivalent to `0b0000000000001111`. This means that 32 binary digits are required to specify negative numbers. For example, `-1` is `0b11111111111111111111111111111111`.

Similarly, 64 binary digits are required to specify negative numbers in the `GIANT` binary format. When adding another binary digit would cause significant a bit to be lost (shifted out most significant end), further characters, whether valid binary digits or not, are not counted as being part of the binary literal.

`0b` followed by an invalid binary character has a value of zero. Type suffixes are invalid following binary literals.

SINGLE *Image*

Single image literals begin with **0s** and are followed by 8 hexadecimal digits. Single image literals are *images* of the 32-bit **SINGLE** data type in 4-bit chunks. Exactly 8 hex digits must follow the **0s** prefix. Any fewer will result in an erroneous value, while hex digits beyond 8 are not included in the single image value.

0s40800000	4.00000	1.000 x 2^{2}
0s40400000	3.00000	1.500 x 2^{1}
0s40000000	2.00000	1.000 x 2^{1}
0s3F800000	1.00000	1.000 x 2^{0}
0s3F400000	0.75000	1.500 x 2^{-1}
0s3F000000	0.50000	1.000 x 2^{-1}
0s3E800000	0.25000	1.000 x 2^{-2}
0s3E000000	0.12500	1.000 x 2^{-3}

Single image format is useful when exact specification of **SINGLE** numbers is necessary, as when creating values for math functions where the highest possible precision is important.

DOUBLE *Image*

Double image literals begin with **0d** and are followed by exactly 16 significant hexadecimal digits (any beyond 16 are not part of the number). Double Image literals are *images* of the 64-bit **DOUBLE** data type in 4-bit chunks.

0d4010000000000000	4.00000	1.000 x 2^{2}
0d4008000000000000	3.00000	1.500 x 2^{1}
0d4000000000000000	2.00000	1.000 x 2^{1}
0d3FF0000000000000	1.00000	1.000 x 2^{0}
0d3FE8000000000000	0.75000	1.500 x 2^{-1}
0d3FE0000000000000	0.50000	1.000 x 2^{-1}
0d3FD0000000000000	0.25000	1.000 x 2^{-2}
0d3FC0000000000000	0.12500	1.000 x 2^{-3}

Double image format is useful when exact specification of **DOUBLE** numbers is necessary, as when creating values for math functions where the highest possible precision is important.

String Literal

String literals are sequences of zero or more ASCII standard characters between `"double quotes"`. For example, in **`x$ = "mark"`**, **`x$`** is a string variable, while **`"mark"`** is a string literal.

Backslash characters are defined for imbedding non-printable characters in literal strings. The **`\`** and the following character are converted to one character, as summarized in the following table.

`\0`	**`0x00`**	**`null`**
`\a`	**`0x07`**	**`alarm (bell)`**
`\b`	**`0x08`**	**`backspace`**
`\d`	**`0x7F`**	**`delete`**
`\e`	**`0x1B`**	**`escape`**
`\f`	**`0x0C`**	**`form-feed`**
`\n`	**`0x0A`**	**`newline`**
`\r`	**`0x0D`**	**`return`**
`\t`	**`0x09`**	**`tab`**
`\v`	**`0x0B`**	**`vertical-tab`**
`\\`	**`0x5C`**	**`backslash`**
`\'`	**`0x27`**	**`single-quote`**
`\"`	**`0x22`**	**`double-quote`**
`\OOO`	**`0oOOO`**	**`octal value`**
`\xHH`	**`0xHH`**	**`hex value 0xHH`**

In **`\OOO`** (octal format), values from **`\000`** to **`\377`** are valid. **`\OOO`** format is initiated by **`\`** followed by an octal digit **`0-7`**, and continues to the first non-octal digit or until three octal digits have been collected. The most significant bit of **`\400`** through **`\777`** is lost.

In **`\xHH`** (hex format), values from **`\x00`** to **`\xFF`** are valid. **`xHH`** format is initiated by **`\x`** and continues to the first non-hex digit or until two hex digits have been collected.

When **`\`** is followed by any non-alphanumeric character, the **`\`** is ignored and the character is included in the string.

When **`\`** is followed by an alphabetic character not shown the preceding table, the backslash is ignored and the character is included in the string. New backslash characters may be defined in subsequent versions, so undefined backslash characters may result in inconsistent behavior.

Constant

Constants are symbols used as descriptive replacements for literal values. They represent fixed numeric or string values that do not change as programs run. Attempts to redefine constants cause compile time errors. The data type of constants is determined by the data type of the literal or constant assigned to it. Type suffixes are not valid on constants, except for **`$`** on string constants. Only literals, constants, and bitfield specs can be assigned to constants.

Local Constant

Local constants are distinguished by **`$`** prefixes, as in **`$SIZE`** and **`$SHAPE`**. They are declared, defined, and typed within functions when values are assigned to them. Local constants are visible only within the function that defines them.

Shared Constant

Shared constants have **`$$`** prefixes, as in **`$$YES`** and **`$$NO`**. They are declared, defined, and typed when values are assigned to them. Shared constants must be defined in the prolog after all function and shared data declarations. Shared constants that are "exported" by a library become defined in a program when it "imports" the library.

System Constant

System constants are shared constants predefined for all programs. The only ones currently defined are **`$$FALSE`** and **`$$TRUE`**.

Variable

Variables are symbols representing values, or groups of values, that can change as a program executes. Simple variables, often just called *variables*, are single numeric values in one of several data types. String variables, or *strings*, are elastic series of unsigned bytes. Array variables, or *arrays* are one or more dimensional arrays of any type of data. Composite variables, or *composites* are fixed-size collections of simple variables, fixed-size arrays, fixed-length strings, and other composites.

Variables, strings, arrays, and composites are all are called variables. When distinction between the various kinds are important, the individual names are used.

Simple Variable

Simple variables are analogous to algebraic variables and represent single numeric values. Simple variables range from 1 byte **`(SBYTE,UBYTE)`** to 8 bytes **`(GIANT,DOUBLE)`**.

Signed integers, *unsigned integers*, and *floating point* numbers are simple data types. The built-in simple data types include:

```
SBYTE       8-bit signed integer
UBYTE       8-bit unsigned integer
SSHORT      16-bit signed integer
USHORT      16-bit unsigned integer
SLONG       32-bit signed integer
ULONG       32-bit unsigned integer
XLONG       32/64 bit generic integer
GOADDR      32/64 bit address integer
SUBADDR     32/64 bit address integer
FUNCADDR    32/64 bit address integer
GIANT       64-bit signed integer
SINGLE      32-bit floating point
DOUBLE      64-bit floating point
```

Bit Field

Bit fields are arbitrary length fields of bits in integer variables. Bit field operations work with 1 - 32 bit wide fields starting at any bit position from 0 *(LSb)* to 31 *(MSb).*

Bit Field Intrinsics

`CLR()`	clear arbitrary bit field to zeros
`SET()`	set arbitrary bit field to ones
`MAKE()`	make an arbitrarily bit field
`EXTS()`	extract arbitrary bit field signed
`EXTU()`	extract arbitrary bit field unsigned

Brace Notation for Bitfields

Brace notation can be used to extract signed and unsigned bit fields, as in **`aa=token{{3,24}}`** (signed) and **`aa = token{3,24}`** (unsigned).

In **`aa = token{3,24}`**, the three bit field starting at bit 24 is extracted from **`token`** and assigned to **`aa`**. The upper 29 bits of **`aa`** become zero since *single braces* specify *unsigned bitfields*. *Double braces* specify *signed bitfields*, so **`aa = token{{3,24}}`** fills the upper bits of **`aa`** with the most significant bit of the extracted 3-bits.

BITFIELD()

The **`BITFIELD()`** intrinsic creates descriptive bitfield constants and variables in a portable, machine independent, way.

The following examples show four variables and constants being given values by the **`BITFIELD()`** intrinsic, then extracted as signed and unsigned bitfields from variable **`token`**.

```
$$TYPE      = BITFIELD (5, 16)      ' 5 bits 16-20
$SCOPE      = BITFIELD (3, 21)      ' 3 bits 21-23
white       = BITFIELD (3, 29)      ' 3 bits 29-31
kind        = BITFIELD (5, 24)      ' 5 bits 24-28
...
tokenType   = token{$$TYPE}         ' 5 bits 16-20
tokenScope  = token{$SCOPE}         ' 3 bits 21-23
tokenWhite  = token{{white}}        ' 3 bits 29-31
tokenKind   = token{kind}           ' 5 bits 24-28
```

String Variable

String variables are sequences of **`UBYTE`** characters. Strings generally contain ASCII text, but can hold arbitrary byte sequences. Strings are *automatically elastic*, meaning they automatically resize to contain whatever number of bytes are put into them. When a string resizes, its location in memory may change, as when a longer string is assigned and there is insufficient room after the string to store the extra bytes. For this reason, every string has a *handle*, an **`XLONG`** value at a fixed location in memory (or CPU register) that always contains the current address of the string data. If the string is empty, the handle contains zero.

Empty strings are **`FALSE`**, while strings with any contents are **`TRUE`**. This makes testing for empty strings simple and efficient, as illustrated by the following examples:

```
IF a$ THEN PRINT "a$ has contents"
IFZ a$ THEN PRINT "a$ is empty"
```

Brace Notation Extract

Brace notation can be used to *extract* individual bytes from strings. **`thisByte = var${byteOffset}`** extracts the **`UBYTE`** from **`var$`** that is **`byteOffset`** characters from the beginning of **`var$`**, and assigns it to numeric variable **`thisByte`**. The first character is at **`byteOffset = 0`**.

Brace notation extract is hundreds of times faster than **`ASC(MID$(var$,n,1))`**, which is the conventional BASIC way to extract one byte from a string. It is faster because intrinsic function call overhead is eliminated, string space management is avoided, and erroneous access outside the string is not restricted (unless bounds checking is enabled).

The two argument **`ASC()`** intrinsic (as in **`ASC(var$,pos)`**) as a safe, intermediate efficiency alternative to brace notation extract. Brace notation extract is an advanced feature to be employed with care.

Brace Notation Assign

Brace notation can also be used to *assign* individual bytes into strings. **`var${byteOffset} = newByte`** replaces the existing character at **`byteOffset`** in **`var$`** with the low byte of **`newByte`**.

For the same reasons as brace notation extract, brace notation assign is much faster than conventional alternatives, and is an advanced feature to be used with care.

Brace notation assign has an additional danger. Unlike brace notation extract, it writes to memory. Since bounds checking is not automatically performed, brace notation assign can write outside the memory allocated for a string, modifying other variables and memory allocation headers, either of which can crash the program and possibly even the development environment. It is particularly important, therefore, to avoid invalid offset values in brace notation assign.

Brace Notation Warning

Brace notation extract and assign are much faster than conventional alternatives for reading and writing single bytes in a string. But either will cause memory faults if used on empty strings, and brace notation assign will cause obscure and disastrous results if used to write beyond the current end of a string. The byte written will overwrite the zero terminator, another variable, or, even worse, a memory allocation header. In most cases, the program and probably the program development environment will crash.

Therefore, *brace notation assign* in particular should be considered an advanced feature for experienced programmers when the fastest possible string processing is needed.

When programs run in the development environment with bounds checking enabled, dangerous brace notation accesses are prevented. Instead, recoverable out-of-bounds errors occur. If bounds checking is disabled, the program and the environment will probably crash. Programs that use brace notation must be designed so improper brace notation accesses are impossible.

Internally, every string is represented by a *handle*, an **`XLONG`** value at a fixed location in memory. It contains the address of the string data, which is elsewhere in memory. Empty strings are represented by a handle containing 0. Any attempt to access an empty string element is therefore an error, usually one that will cause a segment violation or memory fault.

Accesses beyond the end of strings return invalid values, and may cause segment violations. **`LEN(theString$)`** returns the offset of the null character that terminates strings, so as long as the length of the string is non-zero, offsets up to **`LEN(theString$)`** are valid. Be careful not to write over the zero terminator at offset **`LEN(theString$)`** !

To prevent access of empty strings, test for contents first, as in the following example:

```
  IF a$ THEN firstChar = a${0} ELSE firstChar = -1
```

To prevent an attempted read or write from the null terminator or beyond, the brace notation offset should be limited to the length of the string minus one, as in either of the following two-line examples:

```
  stringLength = LEN (a$)
  IF (n < stringLength) THEN thisChar = a${n} ELSE thisChar = -1

  maxOffset = UBOUND (a$)
  IF (n <= maxOffset) THEN thisChar = a${n} ELSE thisChar = -1
```

The following examples illustrate a similar strategy to prevent improper brace notation accesses in loops. The first calculates a hash value for a symbol, and the second converts a symbol to lower case with a **`UBYTE`** conversion array. **`UBOUND()`** returns **`-1`** for empty strings, so these examples will not accidentally attempt to access empty strings - which cause run memory access violations.

```
  hash = 0
  FOR i = 0 TO UBOUND (symbol$)
    hash = hash + symbol${i}
  NEXT i
'
  i = 0
  z = UBOUND (symbol$)
  DO WHILE (i <= z)
    symbol${i} = charsetUpperToLower[symbol${i}]
    INC i
  LOOP
```

String Support

Strings are important in most programs, so many efficient string intrinsics are built in, including:

ASC()	Intrinsic	ASCII value of 1st character of string
CHR$()	Intrinsic	Convert number to 1 char ASCII string
CJUST$()	Intrinsic	Center justify string in field of spaces
CSIZE()	Intrinsic	Number of bytes before null character
CSIZE$()	Intrinsic	Clip string off at 1st null character
CSTRING$()	Intrinsic	Convert C string into native string
FORMAT$()	Intrinsic	Create formatted string from format spec and value
HEX$()	Intrinsic	Hexadecimal string representation of number
HEXX$()	Intrinsic	HEX$ with "0x" prefix
INCHR()	Intrinsic	Find first search-set character in string
INCHRI()	Intrinsic	Same as INCHR() except case insensitive
INFILE$()	Intrinsic	Input line of text from disk file
INLINE$()	Intrinsic	Input line of text from keyboard
INSTR()	Intrinsic	Find first substring in string
INSTRI()	Intrinsic	Same as INSTR() except case insensitive
LCASE$()	Intrinsic	Lower case of string
LCLIP$()	Intrinsic	Clip n bytes from left end of string
LEFT$()	Intrinsic	Leftmost n characters of string
LEN()	Intrinsic	# of elements in string
LJUST$()	Intrinsic	Left justify string in field of spaces
LTRIM$()	Intrinsic	Trim spaces and tabs from left of string
MID$()	Intrinsic	Extract arbitrary part of string
NULL$()	Intrinsic	Create string of n null characters
RCLIP$()	Intrinsic	Clip n bytes from right end of string
RIGHT$()	Intrinsic	Extract rightmost n characters of string
RINCHR()	Intrinsic	Same as INCHR except reverse search direction
RINCHRI()	Intrinsic	Same as RINCHR() except case insensitive
RINSTR()	Intrinsic	Same as INSTR() except reverse search direction
RINSTRI()	Intrinsic	Same as RINSTR() except case insensitive
RJUST$()	Intrinsic	Right justify string in field of spaces
RTRIM$()	Intrinsic	Trim spaces and tabs from right end of string
SPACE$()	Intrinsic	String of n space characters
STRING()	Intrinsic	Convert to type STRING... STRING$() is identical
STUFF$()	Intrinsic	Stuff one string into another
SWAP	Statement	Swap the values of numeric or string variables
TRIM$()	Intrinsic	Remove spaces and tabs from left & right of string
UCASE$()	Intrinsic	Upper case of a string

Composite Data Type

Composite data types are declared and defined in **TYPE** statements in the prolog, before the first function declaration. The names of these *user-defined types* become keywords that can be used just like built-in type names.

Composite variables are fixed-size collections of fixed-size, named *components*. Every component is one of the following:

- Built-in simple numeric type.
- Fixed-size string.
- Previously defined composite type.
- Fixed size one-dimensional array of any of the above.

Composite Type Declaration

Composite types are created by **TYPE** blocks in the prolog, before the first function is declared. The name of the type follows the **TYPE** statement. The types and names of the *components* are defined on subsequent lines, one component per line. An **END TYPE** statement marks the end of the type definition. The following example shows how a composite type called **VICTIM** is created.

```
TYPE VICTIM
  USHORT        .number             ' victim # from 0 to 65535
  STRING * 50   .name               ' victim name up to 50 bytes
  STRING * 50   .address[3]         ' array with (4) 50 byte strings
  SLONG         .zipcode            ' zip code part of address
  GIANT         .phone              ' victim phone number
  UBYTE         .kids               ' number of kids victim has
  UBYTE         .ages[15]           ' age of victim and upto 14 kids
  FUNCADDR      .Poison (STRING)    ' function that poisons the victim
  FUNCADDR      .Shoot (XLONG)      ' function that shoots the victim
  FUNCADDR      .BuryAlive ()       ' function that buries the victim
  FUNCADDR      .Drown ()           ' function that drowns the victim
END TYPE
```

Creating a composite type fixes several things:

- The overall size of variables of the specified type.
- The name of each component.
- The type of each component.
- The size of each component.
- The location of each component within the composite.

Components are naturally aligned. In other words, the addresses of **SSHORT**, **USHORT** components are multiples of **2**, the addresses of **SLONG**, **ULONG**, **SINGLE** components are multiples of **4**, and the addresses of **GIANT**, **DOUBLE** components are multiples of **8**. Sometimes space must be left after a component so the next component begins in natural alignment. This unused space is called *padding*. Careful ordering of elements can eliminate or reduce padding. Composite types larger than 65535 bytes cannot be declared or defined.

Composite Variable

All variables of a given composite type are the same size. This is important, because locating composite variables within disk files is enormously simpler with fixed-size records.

Composite variables are declared in type declaration statements, just like other variables, with the composite type name replacing the built-in type name. For example:

```
SHARED DOUBLE  xx, yy, zz[]          ' Declare conventional variables
SHARED VICTIM  last, current, all[]  ' Declare composite variables
DCOMPLEX  nn, oo, jj[]               ' Declare composite variables
STATIC VICTIM  first, last, sorted[] ' Declare composite variables
```

Component

Components of composite variables are accessed by appending component names, which always begins with a `.` , to the composite variable name, as in **`employee.phone`** where **`employee`** is a composite variable and **`.phone`** is the name of a component. Dots are not valid characters in other symbols, so component names are easy to distinguish from normal symbols.

`SCOMPLEX` *and* **`DCOMPLEX`**

Two *complex number* composite types are predefined for all programs, **`SCOMPLEX`** and **`DCOMPLEX`**. **`SCOMPLEX`** has two **`SINGLE`** components, while **`DCOMPLEX`** has two **`DOUBLE`** components. For both complex types, the component names are **`.R`** and **`.I`**, and contain the real and imaginary parts.

The **`+`**, **`-`**, **`*`**, and **`/`** operators can operate on complex variables, so expressions like:

```
xx = (xx * yy) + (yy / zz)
```

are valid (where **`xx`**, **`yy`**, **`zz`** are all **`SCOMPLEX`** *or* **`DCOMPLEX`**). The components of complex numbers are accessed like any composite variable, as in:

```
xx.R = j#
k# = xx.I
yy.I = zz.I
```

Functions can take composite variables arguments, and/or return composite results. Dozens of **`SCOMPLEX`** and **`DCOMPLEX`** functions are provided in the complex number function library.

Array Variable

Arrays are collections of variables of any single data type, including strings. Elements are numbered from a *lower bound* of `0` to an *upper bound* less than `2147483648`.

`DIM`

Arrays start out empty; they have no elements and take up no space in memory, except for their `XLONG` handle that initially contains `0`. Arrays must be *dimensioned* before their contents can be read or written. When an array is dimensioned by a `DIM` statement, any existing content are freed, a new array is created, and its contents is initialized to zero. When an array is dimensioned, the address of element `0` of its highest dimension is stored in its handle.

`REDIM`

When an array is *redimensioned* in a `REDIM` statement, the values of elements in the old array that are also in the new array are preserved. All new elements are zeroed.

`DIM` *vs* **`REDIM`**

After `DIM` creates or resizes an array, it clears all elements to zero. In contrast, `REDIM` preserves the values of all elements in the original array that are also in the new array, and initializes any new elements to zero. `REDIM` preserves data even in multi-dimensional arrays.

Dimension

`DIM` and `REDIM` can create arrays with `0` to `8` dimensions. Zero dimension arrays are *empty arrays*. The *lowest dimension* is the rightmost. All other dimensions are called *higher dimensions*. Element indices are separated by commas and appear between square brackets, as in `a[b,c,d]`. Arrays are thus easily distinguishable from expressions, intrinsics, and functions.

An entire array is referenced when the square brackets are empty, as in `n[]` or `n$[]`.

Empty Array

Empty arrays are `FALSE`, while arrays with any number of elements are `TRUE`. This makes testing for empty arrays simple and efficient, as illustrated by the following examples:

```
  IF a[] THEN PRINT "a[] has contents"
  IFZ a[] THEN PRINT "a[] is empty"
```

Passing Array Arguments

A whole array can be passed by reference to a function by leaving the square brackets empty, as in:

```
 a = Func(a,@b[],@c[]).
```

When arrays are passed to functions, their type is checked against the argument list in the declaration of the function being called. If the type of the array is not the same as the declared type, a type-mismatch error occurs. However, if the keyword `ANY` appears before an array argument in the function declaration, any type of array may be passed. The called function will receive the array as whatever type array is shown in the argument list on its `FUNCTION` line. The type of the array can be tested with `TYPE()`, as in `t=TYPE(a[])`, and `ATTACH` can attach it to an array of the appropriate type.

Array Element

The elements of an array are variables, and are used in expressions just like normal variables, as in the following examples:

```
FUNCTION  Demo (w, x, y, z$)
  SHARED  a%[]                      ' a%[] has SHARED scope
  SHARED  k[]                       ' k[] has SHARED scope
  STATIC  j$[]                      ' j$[] has STATIC scope

  IFZ j$[] THEN DIM j$[63]          ' DIM j$[] on first entry
  a%[w] = x                         ' wth element of a%[] = x
  k[z] = x * a%[w] * a%[y]          ' use arrays in an expression
  j$[x] = z$                        ' put a string in j$[x]
END FUNCTION
```

Array Implementation

One-dimensional arrays are implemented as a contiguous set of elements, much like other languages. Multi-dimensional arrays, however, are implemented as multi-level trees of one-dimensional arrays. As long as they are created and used in the conventional manner, this implementation difference is invisible. But *tree-structure arrays* have *enormous* advantages.

Tree Structure

Multi-dimensional arrays are arrays of array addresses, except for the lowest dimension which contains data. This permits the construction of *irregular arrays*, for which there are supporting features.

Regular Array

There's no difference between conventional vs tree structure, or regular vs irregular in one dimensional arrays. And multi-dimensional arrays behave just like conventional arrays as long as the features provided to create and manipulate irregular arrays are not employed.

Irregular Array

The distinction between regular and irregular arrays applies only to multi-dimensional arrays. Furthermore, irregular arrays need not be built or manipulated until and unless their special properties are needed. Many programmers may never need them. On the other hand, system programs, and other sophisticated applications are often *vastly* more efficient and easier to implement with irregular arrays.

Regular arrays have the same number of elements in every *instance* of a given dimension. In an array created by `DIM a[3,7]`, the upper bound of the low dimension is 7 for all three upper dimension values `(a[0,7]`, `a[1,7]`, `a[2,7]`, `a[3,7])`. Regular arrays can be thought of as *rectangular*.

Irregular arrays don't need to have the same number of elements in every instance of a given dimension. For example, the upper bounds of the low dimension of a similar irregular array could be `b[0,3]`, `b[1,*]`, `b[2,5]`, `b[3,2]` (where "*" means no elements are allocated in this subdimension). Therefore, irregular arrays cannot be thought of as rectangular, but can be thought of as *ragged arrays*.

Nodes and Data

Arrays contain two kinds of contents, *nodes* and *data*. Elements in the lowest dimension are data. Elements in higher dimensions are nodes, meaning addresses of *sub-arrays*.

Consider `a[]`, created by `DIM a[3,7]`. Five one-dimensional arrays are created. The first array corresponds to the higher dimension, `a[3, ]`, and contains elements 0 to 3. These elements are nodes, addresses of four arrays that contain data elements 0 to 7.

Building Irregular Arrays

To build an irregular array, the individual one-dimensional sub-arrays of a multi-dimensional array are created separately, then assembled into the complete irregular array. Alternatively, a regular array can be converted into an irregular array by detaching subarrays, redimensioning them, then reattaching them **(ATTACH**, **REDIM**, **ATTACH).**

The five one-dimensional arrays created and assembled by

```
  DIM a[3,7]
```

in the preceding example could also be created and assembled by either of the following two functions:

```
FUNCTION  DimDemo1 ()
  SHARED  a[]           ' let's say a[] is a SHARED array
'
  DIM a[3,]             ' create upper dimension of a[]
  DIM a0[7]             ' create a0[7] to be data in a[0, ]
  DIM a1[7]             ' create a1[7] to be data in a[1, ]
  DIM a2[7]             ' create a2[7] to be data in a[2, ]
  DIM a3[7]             ' create a3[7] to be data in a[3, ]
  ATTACH a0[] TO a[0,]  ' attach a0[] to a[0, ]
  ATTACH a1[] TO a[1,]  ' attach a1[] to a[1, ]
  ATTACH a2[] TO a[2,]  ' attach a2[] to a[2, ]
  ATTACH a3[] TO a[3,]  ' attach a3[] to a[3, ]
END FUNCTION

FUNCTION  DimDemo2 ()
  SHARED  a[]           ' let's say a[] is a SHARED array
'
  DIM a[3,]             ' create upper dimension of a[]
  FOR i = 0 TO 3        ' for each element of upper dimension
    DIM n[7]            ' create an array with 7 data elements
    ATTACH n[] TO a[i,] ' attach to upper dimension element
  NEXT i                ' next element of upper dimension
END FUNCTION            '
```

ATTACH *and* **SWAP**

The **ATTACH** statement first verifies the *destination array* or *node* is empty, then swaps it with *source array* or *node*. This has the effect of attaching the source to the destination, leaving the source empty. **SWAP** is identical except the destination is not checked for zero before the arrays or nodes are swapped.

```
  ATTACH   xx[]   TO   yy[]
  ATTACH   xx[]   TO   nn[a, b, ]
  ATTACH   nn[a, b, ]   TO  xx[]
  ATTACH   nn[a, b, ]   TO   oo[c, d, e, ]
  SWAP b[], c[]
  SWAP b[], c[n, ]
  SWAP a$[n, ], a$[m, ]
```

It follows that irregular arrays may have different number of nodes down various branches of the tree structure. Nodes that contain a zero are called *empty nodes* and mark the termination of the tree structure prior to arriving at data. It cannot be known at compile time or runtime what nodes will be empty, the structure of programs must assure that accesses beyond empty nodes are not attempted. It is wise to leave bounds checking enabled during program development to catch errors of this kind because memory corruption and development environment crashes could result.

Excess Comma Notation

The contents of any node can be checked with *excess comma* notation, as in the following examples:

```
  IF a[x, ]  THEN PRINT "non-empty-node"

  IFZ j[x, y, ] THEN PRINT "empty-node"
```

Excess comma notation says a node is being accessed, not data. When nodes are being accessed, excess comma notation is mandatory. When data is begin accessed, excess comma notation is an error. When bounds checking is enabled, node/data mismatches are caught before the offending access occurs.

How would an irregular array `a%[]` be created with upper bounds of `a%[0,3]`, `a%[1,*]`, `a%[2,5]`, `a%[3,2]` where `*` = empty-node?

The following code illustrates:

```
FUNCTION  demo ()
  SHARED  a%[]               ' let's say a%[] is a SHARED array
  DIM a%[3,]                 ' create upper dimension of a[]
  DIM a0%[3]                 ' create a0%[3] to be data in a%[0, ]
' DIM a1%[]                  ' create a1%[]  to be data in a%[1, ]
  DIM a2%[5]                 ' create a2%[5] to be data in a%[2, ]
  DIM a3%[2]                 ' create a3%[2] to be data in a%[3, ]
  ATTACH a0%[] TO a%[0]      ' attach a0%[] data array to a%[0, ]
' ATTACH a1%[] TO a%[1]      ' attach a1%[] data array to a%[1, ]
  ATTACH a2%[] TO a%[2]      ' attach a2%[] data array to a%[2, ]
  ATTACH a3%[] TO a%[3]      ' attach a3%[] data array to a%[3, ]
  IF a%[0,] THEN PRINT "0"   ' will print "0"
  IF a%[1,] THEN PRINT "1"   ' will not print
  IF a%[2,] THEN PRINT "2"   ' will print "2"
  IF a%[3,] THEN PRINT "3"   ' will print "3"
END FUNCTION
```

The two lines commented out in the preceding example are not needed because `DIM a[3,]` initializes all nodes to zero. Since `a[1,]` is already a empty-node, there's no need to create a empty array to attach to `a[1,]`.

Natural Data Type

Arrays have a *natural data type*, determined by type-suffix, declaration, or default. Any access of a data element in an array is a read or write of natural type data.

Considerations of Tree Structure Arrays

- Lowest dimension may be any valid type, including structure types.
- Higher dimensions MUST be dimensioned as higher dimensions `DIM a[h, ]`.
- A given array or subarray must contain all nodes or all data of one type.
- Irregular arrays may contain variable numbers of dimensions.
- Irregular arrays may contain empty nodes (node contents = 0) at any level.
- Programs must assure accesses through empty nodes are not attempted.
- Programs must assure accesses through data elements are not attempted.
- Valid irregular arrays have only data in the lowest dimension.
- Valid irregular arrays have only nodes in higher dimensions.
- All subarrays below a node are freed when the node is detached and freed.
- Array nodes are accessed/tested with "excess comma" syntax (`a%[n,m,]`).
- An array of any number of dimensions can be attached at any empty node.
- Arrays must never be attached to the lowest dimension (data element).
- Arrays must never be attached to non-empty nodes.
- An empty array is one kind of empty node.

Scope

The *scope* of variables can be:

```
AUTO
AUTOX
STATIC
SHARED
SHARED /groupname/
EXTERNAL
EXTERNAL /groupname/
```

Scope controls access to variables in the following way:

`EXTERNAL` variables are accessible to all programs in a single executable, which may be more than one program if linked together into an executable or library. **`EXTERNAL`** variables are not shared between separate executables, whether programs or libraries.

`SHARED` variables are accessible to all functions in a program.

`STATIC` variables are accessible to a single function, but are common to all instances of the function.

`AUTO` and **`AUTOX`** variables are accessible only by a single instance of a function - they are newly created each time the function is called.

Visible Scope

Two *scope prefixes* are defined so **`SHARED`** and **`EXTERNAL`** variables can be visibly marked and not redeclared in every function that accesses them.

Variable names with a **`#`** prefix are **`SHARED`**, whether or not they appear in a **`SHARED`** variable declaration statement or not.

Variable names with a **`##`** prefix are **`EXTERNAL`**, whether or not they appear in an **`EXTERNAL`** variable declaration or not.

Note that variables **`xx`** and **`#xx`** and **`##xx`** are three independent variables, no matter the scope and/or data type of **`xx`**. **`#`** and **`##`** prefixes are considered part of the variable name or symbol.

AUTO

AUTO variables are accessible to one instance of a single function. Each time the function is called a new set of **AUTO** variables are assigned to CPU registers or memory on the local frame, so several sets of a function's **AUTO** variables may exist at any time, each of which is accessible to only one instance of the function. **AUTO** variables are *temporary*, they exist only until the function completes and returns to the caller. Variables are **AUTO** by default, so all undeclared variables are **AUTO**.

In well written programs, most variables are **AUTO**. They are the default because they are completely insulated from all other functions and programs, including other instances of the same function.

AUTO variables are assigned to registers in the order they appear, so putting important variables in **AUTO** statements will cause as many as possible to be assigned to registers. When registers are no longer available, **AUTO** variables are assigned locations on the local stack frame along with **AUTOX** variables. Executing from registers is faster, so register variables can speed program execution.

AUTOX

AUTOX variables are the same as **AUTO** variables except they are always assigned locations on the local stack frame, never in CPU registers. Numeric variables whose addresses are taken must not be declared **AUTO**, since CPU registers do not have addresses.

String, array, and composite variables whose handle addresses are taken must not be declared **AUTO** for the same reason. **AUTOX** is generally needed only when calling **C** functions written to receive arguments that are addresses.

STATIC

STATIC variables are accessible to all instances of a single function. **STATIC** variables are assigned fixed memory locations for the lifetime of the program, so they are local to a single function, but common to all instances of the function, and their values persist between function calls.

SHARED

SHARED variables are accessible to all functions in a single program that declare **SHARED** variables of the same name. **SHARED** variables are allocated space at fixed memory locations for the program lifetime, so their values persist between calls of the functions that access them, and are common to all functions that access them.

A **#** prefix on a variable name declares the variable as **SHARED**, even when the variable is not declared in a **SHARED** statement.

SHARED /groupname/

SHARED */groupname/* variables are accessible to all functions in a single program that declare **SHARED */groupname/*** variables of the same name and ***/groupname/***.

SHARED */groupname/* variables are allocated space at fixed memory locations for the lifetime of the program, so their values persist between calls of the functions that access them, and are common to all functions that access them.

The flexibility of ***/groupname/*** is illustrated in the following example, where the comments describe the variable sharing between two functions:

```
FUNCTION One ()
  SHARED /groupOne/ a,b,c  ' a shared with Two(),  b, c shared with nobody
  SHARED /groupTwo/ i,j,k  ' i shared with Two(),  j, k shared with nobody
  SHARED x, y, z           ' z shared with Two(),  x, y shared with nobody
  x = a * i                ' a, i shared with Two(),  x shared with nobody
  y = b * j                ' b, j, y shared with nobody
  z = c * k                ' z shared with Two(),  c, y shared with nobody
END FUNCTION
'
FUNCTION Two ()
  SHARED /groupOne/ a,n  ' a shared with One(),  n shared with nobody
  SHARED /groupTwo/ i,o  ' i shared with One(),  o shared with nobody
  SHARED c, k, z         ' z shared with One(),  c, k shared with nobody
  x = a * j * n          ' x,j are AUTO, a shared with One(), n not shared
  y = b * i * o          ' y, b are AUTO, i shared with One(), o not shared
  z = c *  p             ' z shared with One(), c, p shared with nobody
END FUNCTION
```

EXTERNAL

EXTERNAL variables are accessible to all functions in all programs in a single *executable* that declare **EXTERNAL** variables of the same name. **EXTERNAL** variables are allocated space at fixed memory locations for the lifetime of the executable, so their values persist between calls of functions that access them, and are shared among all functions (in all linked programs) that access them.

A **##** prefix on a variable name declares the variable as **EXTERNAL**, even when the variable is not declared in an **EXTERNAL** statement.

EXTERNAL variables and arrays may not have the same name, so **a** and **a[]** cannot both be **EXTERNAL**.

EXTERNAL /groupname/

EXTERNAL */groupname/* variables are accessible to all functions in a single executable that declare **EXTERNAL /groupname/** variables of the same name and **/groupname/**.

EXTERNAL */groupname/* variables are allocated space at fixed memory locations for the lifetime of the executable, so their values persist between calls of the functions that access them, and are common to all functions (in all linked programs) that access them.

EXTERNAL variables and arrays may not have the same name, so **aaa** and **aaa[]** cannot both be **EXTERNAL /x/**.

Programs

Program

Programs consist of a *prolog*, followed by one or more *functions*. Programs may be terminated by an **`END PROGRAM`** statement after the last function, but it is not required.

The following program contains a 2 line prolog and a 3 line function:

```
DECLARE FUNCTION  Hello ()    ' PROLOG   - Declare program function
'                             ' PROLOG   - Comment line
FUNCTION  Hello ()            ' FUNCTION - Begin function definition
  PRINT "Hello World"         ' FUNCTION - Function body or contents
END FUNCTION                  ' FUNCTION - End function definition
```

Prolog

A *prolog* consists of the following elements *in the listed order:*

1. *Declarations of all external programs/libraries referenced by the program, if any.*
2. *Declarations/Definitions of all user-defined data-types, if any.*
3. *Declarations of all functions in the program (at least one).*
4. *Declarations of special external functions called by program (optional).*
5. *Declarations of shared & external variables, if any (optional).*
6. *Declarations/Definitions of all shared constants, if any.*

The first declared function is the *entry function* - execution begins here when the program is run.

The functions declared in the prolog immediately follow the prolog. Only blank lines and comment lines may appear between functions. Comment lines and blank lines between functions are associated with the function that follows, so function comments must precede the function they describe.

Prolog Elements Example

```
PROGRAM "trouble"               ' name the program or library
VERSION "0.0000"                ' keep version number updated
IMPORT  "xst"                   ' make standard library available
IMPORT  "mylib"                 ' make custom library available
'
EXPORT                          ' begin exporting types, funcs, etc
  TYPE HANDLE = XLONG           ' declare user-defined type HANDLE
  TYPE LINK                     ' declare user-defined type LINK
    XLONG  .backward            ' define component of type LINK
    XLONG  .forward             ' define component of type LINK
  END TYPE                      ' end declaration of type LINK
'
  DECLARE FUNCTION  Entry ()    ' declare function in this program
END EXPORT                      ' stop exporting (temporarily)
'
  SHARED  lastIndex, thisIndex  ' declare shared variables
  EXTERNAL  flexStatus          ' declare external variable
EXPORT                          ' resume exporting
  $$Off = 0                     ' define and export shared constant
  $$On = -1                     ' define and export shared constant
END EXPORT                      ' stop exporting (the end)
```

Function Library aka Library

A *function library* aka *library* is a collection of functions designed to be called by other programs and/or function libraries. A function library is like a conventional program, except:

- ***its* `PROGRAM` *statement argument must be the same as its filename.***
- ***its external variables are not accessible to other programs or dlls.***
- ***its entry function does little or nothing except initialization.***
- ***its entry function is called automatically when imported.***
- ***it doesn't usually perform a complete activity by itself.***
- ***it is compiled as a function library.***

The math library is a good example: `"xma"` has no external variables, its entry function does nothing, and it performs no coherent activity. It exists only to make its individual math functions and constants available to other programs. Accessory toolsets for *GuiDesigner* are also function libraries.

When a function library is created, the PDE saves a file called ***libname*`.dec`**, where ***libname*** is the name of the program. ***libname*`.dec`** contains all lines in **`EXPORT ... END EXPORT`** blocks in library prologs.

***libname*`.dec`** is needed by programs that **`IMPORT "`*libname*`"`**.

System Functions and Foreign Functions

System functions are library functions in the operating system, for example Win32s for Windows 3.1, Win32 for Windows95 and WindowsNT, OS/2, UNIX,, etc. *Foreign functions* are library functions written in some other programming language that has a compatible function interface.

Both are treated as *foreign functions* in the following discussion.

Programs can call foreign functions just like its own functions and functions in native function libraries. To make foreign functions visible, list the libraries that contains them in the PROLOG, in **`IMPORT`** statements - as in **`IMPORT "kernel32"`**.

Foreign functions and function libraries must have ***libname*`.dec`** files. You'll have to create them yourself or find a company that sells them as an accessory product. It isn't difficult to create or suppliment ***libname*`.dec`** files yourself, since you only have to include declarations for the types, functions, and constants that your program references. Later you can add new type, function, and constant declarations on an as needed basis. The definitions for type and functions are available in operating system documentation as well as ***libname*`.h`** files.

Some ***libname*`.dec`** files are provided with the PDE to make it easy to call common system functions, and **`kernel32.dec`** is one example. These system declaration files don't usually contain every data type, constant, and function in the system function library, but you can add them on an as needed basis.

***libname*`.def`** files are not required to access foreign functions referenced by programs being run in the program development environment, but they are usually required to make standalone executables. ***libname.*`def`** is not needed for system functions since ***libname*`.lib`** files already exist.

IMPORT

`IMPORT "`*`libname`*`"` statements make function libraries available to a program. The functions and shared constants of the function library are *imported* into the program. Programs can call functions in imported libraries as if those functions were in the same program, and programs can reference shared constants defined in imported libraries as if those shared constants were in the same program. *External and shared variables in libraries are not visible however.*

When **`IMPORT "`*`libname`*`"`** is compiled:

- ***libname*`.dec` *is read in and compiled as if it was in the program.***
- ***libname*`.dll` *is loaded and linked with the program (if it exists).***
- ***libname*`.dll` *entry function is called (to initialize the library).***

***libname*`.dec`** contains statements that define types, declare functions, and define shared constants in the function library.

***libname*`.dec`** is created automatically when a program is compiled as a function library by the program development environment (see the following **`EXPORT`** statement). For libraries written in other languages, ***libname*`.dec`** must be written by the programmer or supplier of the function libraries.

EXPORT ... END EXPORT

`EXPORT` and **`END EXPORT`** statements enclose type definitions, function declarations, and shared constant definitions that programs want to *export* - make visible and accessible to other programs.

When a program is compiled into a function library, the lines between **`EXPORT`** and **`END EXPORT`** are *exported* to any program that *imports* the library with **`IMPORT "`*`libname`*`"`**.

Blowback Function - `Blowback()`

Programs running in the PDE are terminated when you "kill" them and when they "crash" due to fatal errors like segment violations. Unfinished business may be pending when programs are terminated. For example, files may have been opened and never closed.

For programs that execute only built-in statements and intrinsics and only call functions in the built-in libraries (`Xst`, `Xma`, `Xcm`, `Xgr`, `Xui`), all such issues are resolved automatically by the PDE and libraries.

But the PDE and built-in libraries have no idea what happens when your program directly calls system functions and unknown libraries. For example, if a program calls a system function instead of the **`OPEN()`** intrinsic to open a file, the file may still be open when the program crashes, is terminated, or finishes without closing the file. Any subsequent attempt to open the file will therefore fail.

To resolve this problem, whenever a program terminates, the PDE calls the **`Blowback()`** function in the program and every library. Any program or library that calls system functions should contain a **`Blowback()`** function to detect and resolve all "unfinished business".

Functions

Functions

Functions consist of the following elements, in the specified order:

- ***`FUNCTION` statement.***
- ***Declarations of variables used in function.***
- ***Declarations / Definitions of all local constants.***
- ***Executable statements including any number of `RETURN` statements.***
- ***`END FUNCTION` statement.***

Executable statements are valid only within functions.

Entry Function

The *entry function* is the first function declared in the prolog in an **`INTERNAL`** or **`DECLARE`** statement. The first time the entry function is called, all **`SHARED`** and **`EXTERNAL`** variables are cleared.

Function Names

Function names are valid symbols followed by a left parenthesis. Unlike other symbols, *function names may not take type-suffixes*, except **`$`** is valid to specify string return type.

By convention, the first character in each word in function names is capitalized, including the first character in the function name. Function names are *always* followed by [whitespace plus] parentheses **`()`**, whether they take arguments or not.

Encapsulation

By default, functions are *fully encapsulated.* All variables are **`AUTO`** unless explicitly declared otherwise *within the function*. Variables declared **`SHARED`** or **`EXTERNAL`** in other functions do not intrude upon same named variables within a function unless they are also declared in a **`SHARED`** or **`EXTERNAL`** statement within the function, or prefixed with an explicit **`SHARED`** or **`EXTERNAL`** scope prefix like **`#shared`** or **`##external`**.

Arguments

Functions take *arguments*, values passed to them each time they are called. Functions can take as few as zero, or as many as 16 arguments. **`GIANT`** and **`DOUBLE`** arguments count as two arguments, as do the preceding arguments.

Input argument data types are specified in **`DECLARE FUNCTION`** statements at the beginning of programs, and in **`FUNCTION`** statements that begin each function. When functions are called, numeric arguments are automatically converted to the declared types before function execution begins. Strings and composite arguments must match the declared type.

Function Declaration

All functions that are defined and/or referenced in a program must be *declared* in the prolog of that program in one of the following declaration statements:

```
DECLARE  FUNCTION [typename] FuncName ([arglist])
INTERNAL FUNCTION [typename] FuncName ([arglist])
EXTERNAL FUNCTION [typename] FuncName ([arglist])
```

`DECLARE FUNCTION` declares functions that will be defined in the current program, and makes them visible to other programs.

`INTERNAL FUNCTION` declares functions that are defined in the current program, and only visible from this program.

`EXTERNAL FUNCTION` declares functions that are defined in another program, and must be visible from this program.

Function declarations determine the number and data type of arguments expected by functions, and the data type of the return value. When functions are called, the compiler examines the declaration information to:

- Verify that the correct number of arguments have been passed to the function.
- Convert each argument to the data type expected by the function before calling the function.
- Convert each argument passed by reference back to original data type after the function returns.
- Know the data type of the value returned by the function so it can be used properly in an expression or assignment.

Argument Checking

In functions that take no arguments, a right parenthesis must follow the left parenthesis, as in **`Func()`** or **`Func ()`**. Otherwise a comma separated list of *parameters* is placed between the parentheses. The number of arguments the function will accept, and the data type of each is determined by the parameter list. The data type of each parameter can be specified in one of the following ways:

```
Typename.                       Func (XLONG, STRING, GIANT, ANY)
Typename followed by symbol.    Func (XLONG x, DOUBLE ddd)
A symbol with a type-suffix.    Func (rip!, tear#, shred$)
Any of these followed by [].    Func (XLONG[], ANY a[], n$[])
...  (final parameter)          Func (a$, ...)  ( EXTERNALs only )
```

ANY

Arguments can be declared as type **`ANY`**, as shown in the preceding examples. This permits any type of variable or array to be passed in this argument position. When arrays of different types are passed this way, the called function must check the type of its corresponding array argument with the **`TYPE()`** intrinsic, and attach it to an appropriate type array, before accessing its elements.

. . .

The `...` parameter exists to make it possible to declare and call **`C`** functions that have a corresponding `...` parameter. `...` must be the final entry in the parameter list. `...` tells the compiler to accept zero or more additional arguments of any data type. This capability exists in **`C`** for implementing functions like **`printf()`**, where the number of arguments and their types will vary. Native functions cannot be declared with a `...` parameter.

Return Type

The *typename* field in function declarations is optional. Functions that return an **`XLONG`** value or no value do not have to specify a *return type*. Type-suffixes other than **`$`** are not valid on function names, so *typename* is needed on functions that do not return **`XLONG`** or no value (or **`STRING`** with **`$`** suffix).

Function Definition

A *function definition* is the code that executes when a function is called, and includes everything from the **`FUNCTION`** statement through the next **`END FUNCTION`** statement.

`FUNCTION` statements have the following syntax:

```
  FUNCTION [typename] FuncName([arglist]) [default-typename]
```

Default Type

The data type of variables not typed by type-suffixes or variable type declaration statements become the default type, which is **`XLONG`**.

Function Arguments

If a function takes no arguments, a right parenthesis must follow the left parenthesis (separable only by whitespace). Otherwise a comma separated list of arguments is placed between the parentheses. The data type of each argument is specified in one of the following ways:

```
  Typename plus argument name       XLONG x, DOUBLE ddd
  Argument name with type-suffix    n!, bend#, name$
  Any of these followed by []       SINGLE j[], name$[]
```

`RETURN` *and* **`EXIT FUNCTION`**

`RETURN` and **`EXIT FUNCTION`** statements cause the same action as **`END FUNCTION`**, except any number of these statements can be used in each function, they need not start a source line, and they may appear within block structures. **`RETURN`** and **`EXIT FUNCTION`** are often used within decision blocks like **`IF`** and **`SELECT CASE`**.

The syntax of **`RETURN`** and **`EXIT FUNCTION`** are:

```
  RETURN [expression]
  EXIT FUNCTION [expression]
```

`END FUNCTION`

`END FUNCTION` statements mark the end of function definitions and have have the following syntax:

```
  END FUNCTION [expression]
```

`END FUNCTION` mark the end of function and causes execution to return to the function that called it. The value of the expression following **`END FUNCTION`** is returned by the function, or, if there is no expression, zero or null-string is returned.

Function Arguments

When functions are called, 0 to 16 values are passed to them. Each of these *arguments* can be a numeric value, a string value, or a whole array. Every **`GIANT`** and **`DOUBLE`** variable counts as two arguments, as does the preceding argument. Arrays and composites count as one argument, regardless of type.

Numeric Arguments

```
a = Func (b, c#+d#, e$$[f,g])
```

String Arguments

```
a = Func (b$, c$+d$, e$[f,g])
```

Array Arguments

```
a = Func (@b[], @c%[], @d#[], @e$[])
```

Argument Kind and Type Checking

The kind and data type of each argument must be compatible with the corresponding argument in the function declaration. The data type of numeric arguments do not have to be identical to the corresponding parameter in the function declaration. When they differ, the compiler converts the value to the declared type before passing it to the function. The compiler will not, however, convert string or composite arguments to numeric types or vice versa. The types of array arguments must be identical, except for arrays given the **`ANY`** type in the function declaration.

Functions may not be called within a function argument list. This prevents argument evaluation order differences from occuring when programs are ported from system to system.

```
ERROR:   a = Func1 ( b, Func2() )    ' argument attempted to call Func2()
OK:      a = Func1 ( b, &Func2() )   ' no argument attempt to call Func2()
```

Pass by Value

Pass by Value is the most natural way to pass arguments to functions. As the name implies, the value of a variable or expression is passed to the called function. In the following example, three numeric and two string values are passed by value to **`Func()`**.

```
  a = Func (b%, c#+d#, f$$, a$, b$+c$)
```

Pass by Reference

Pass by Reference is another way to pass arguments to functions. In *other* languages, the address of a variable is passed to the called function. Since the called function accesses the argument through this address, the calling and called function are sharing the variable. Therefore, if the called function alters its argument, the calling function will find the value of its variable has been changed.

Though functions can return at most one non-argument value, additional values can be returned in pass by reference arguments. In the following statement, **`x#,y#,z#`** are passed to **`Rotate()`** by reference so **`Rotate()`** can return three values in these variables.

From the values of **`object,crossSection,vertex,`** **`Rotate()`** computes three **`DOUBLE`** values and assigns them to its first three arguments (which might be **`a#,b#,c#`**). Since the calling function passed **`x#,y#,z#`** by reference, it receives these final values of **`a#,b#,c#`** from **`Rotate()`** in **`x#,y#,z#`**.

```
  Rotate (@x#, @y#, @z#, object, crossSection, vertex)
```

Implementation

Unlike other languages, arguments passed by reference are actually passed by value to the specified function. But after the called function returns, the calling function grabs the final value of the argument and assigns it to the original variable.

Therefore the result is the same as conventional pass by reference. The original variable is altered in accordance with the operation of the called function. But the native implementation is faster, supports arbitrary mixing of pass by value and pass by reference, and works on variables in registers, which passing by address (ala C) does not.

Consequences of Pass by Reference Implementation
Any combination of arguments can be passed by reference by prefixing them with `@`.

Example Illustrates Pass by Reference
The following contrived program segment demonstrates that each and every time a function is invoked, arguments can be passed to it in any mix of pass by value and pass by reference:

```
FUNCTION IncSum (x, y, z)       ' IncSum takes three arguments...
  INC x : INC y : INC z         ' increments them...
END FUNCTION (x + y + z)        ' End of function
. . .
FUNCTION TestByRef ()
  a = 0 : b = 10 : c = 20       ' Give initial values to a, b, c
                                ' retval   after   after    after
  x = IncSum ( a, b, c)         ' x = 33:  a = 0:  b = 10:  c = 20
  x = IncSum ( a,  b, @c)       ' x = 33:  a = 0:  b = 10:  c = 21
  x = IncSum ( a, @b, c)        ' x = 34:  a = 0:  b = 11:  c = 21
  x = IncSum ( a, @b, @c)       ' x = 35:  a = 0:  b = 12:  c = 22
  x = IncSum (@a, b,c)          ' x = 37:  a = 1:  b = 12:  c = 22
  x = IncSum (@a, b, @c)        ' x = 38:  a = 2:  b = 12:  c = 23
  x = IncSum (@a, @b, c)        ' x = 40:  a = 3:  b = 13:  c = 23
  x = IncSum (@a, @b, @c)       ' x = 42:  a = 4:  b = 14:  c = 24
END FUNCTION
```

Function `IncSum()` is called eight different ways. Conventional pass by value languages would require eight separate functions.

Because various C implementations handle arguments differently, arguments cannot be passed by reference to C functions.

Pass by Address
Pass by Address is provided for calling C functions that expect argument addresses. To pass by address, an `&` (address operator) can be prefixed to numeric, composite, string, array, and function arguments to produce an `XLONG` address. Pass by address is of little value within native programs because pass by value and pass by reference are sufficient, more efficient, and avoid allocation problems caused by pass by address.

Argument Checking
Functions require a specific kind (variable or array), and data type for each argument, as declared in the `DECLARE FUNCTION` parameter list and specified in the `FUNCTION` argument list.

Passing an array where a variable is expected, or vice versa, causes a kind mismatch error. The compiler compares the data type of each argument passed to a function against the declared type. If they are the same, the argument is passed directly. If they are different types, but both numeric, the argument is converted to the declared type before it is passed. Otherwise a type-mismatch error occurs.

Arrays can only be passed by reference because accidentally passing an array by value would degrade overall program performance by a factor of hundreds to millions.

Return Value

Calling functions are free to receive *return values* or ignore them. Functions are also free to return values or not. When a function returns no value, the compiler returns a zero in the data type declared for the function.

The following examples illustrate that there can be any number of **`RETURN`** and **`EXIT FUNCTION`** statements in a function, but only one **`END FUNCTION`**:

```
FUNCTION blivit ()
  RETURN                    ' return zero/empty string
  RETURN (a+b*c+d)          ' return result of expression "(a+b*c+d)"
  EXIT FUNCTION             ' return zero / empty string
  EXIT FUNCTION (retval)    ' return the value of variable "retval"
END FUNCTION final          ' return the value of variable "final"
```

Function Call

Functions sometimes appear in expressions in much the same way as algebreic functions, the value returned by the function being one value in the evaluation of the expression. For example:

```
  a = b * Func (c,d) + e
```

Sometimes the value returned by a function is unimportant, but the actions performed by it are desired, as in the following example:

```
  Func (c,d)
```

Computed Function Call

Sometimes the function to be executed in an expression depends upon the value of a variable or expression. For these situations, *computed function calls* through variables and arrays of the **`FUNCADDR`** data type are provided. The addresses of functions can be loaded into **`FUNCADDR`** variables and arrays during program execution. When a **`FUNCADDR`** variable or array appears in an expression with a **`@`** prefix, the function whose address is read from the variable is called. For example:

```
  a = b * @funcVar(j) + e
  a = b * @funcArray[c,d](x,y) + e
```

In this example, variable **`j`** is an argument to the function whose address was most recently assigned to **`FUNCADDR`** variable **`funcVar`**. Variables **`x`** and **`y`** are arguments to the function whose address was most recently assigned to **`FUNCADDR`** array element **`funcArray[c,d]`**.

Sometimes the return value from a computed function invocation is not needed, but the action performed by it is. For example:

```
  @funcVar (j)
  @funcArray[c,d](x,y)
```

Fall Through

Before calling a function through a **`FUNCADDR`** variable or array, code checks the address in the variable or array element. If the address is zero, no function is called, but a return value of zero is simulated and execution continues at the next statement. Therefore, invalid array elements can be left at zero, and zero can be assigned to **`FUNCADDR`** variables and array elements to cancel existing entries.

Recursion

Functions are *recursive*, meaning they can call themselves directly or indirectly. **`AUTO`** and **`AUTOX`** variables are allocated anew each time the function is invoked. **`STATIC`** variables are shared by every instance of the function.

Execution Control

Execution Order

In most programs, the execution of source lines does not proceed strictly in the order of the source statements. *Execution control* statements provide capabilities to alter program flow in several ways.

Conventional `GOTO`

`GOTO` ***`labelName`*** statements transfer execution to ***`labelName`*** in the same function. Labels are local, so the same ***`labelName`*** can appear in any number of functions.

Computed `GOTO`

Computed `GOTO` statements transfer execution to labels whose addresses are contained in variables and arrays of the `GOADDR` data type. For example:

```
  GOTO @goVar
  GOTO @goArray[n]
```

`GOTO @goVar` jumps to the address in the `GOADDR` variable `goVar`, while `GOTO @goArray[n]` jumps to the address in element `n` of array `goArray[]`. If the address is zero, no `GOTO` is performed and execution continues with the next statement.

The `GOADDRESS()` intrinsic loads label addresses into `GOADDR` variables and arrays as follows:

```
  goVar = GOADDRESS (labelName)
  goArray[n] = GOADDRESS (labelName)
```

Computed `GOTO` statements are most useful when one of a number of actions must be performed based on some variable or condition. For example, the following code segment uses *computed* `GOTO` to jump to one of eight labels whose addresses are in `dispatch[]`, based on a 3-bit field in variable `message`:

```
FUNCTION  Process ( message )
  action = message{3, 29}
  GOTO @dispatch[action]
  '   ...
END FUNCTION
```

Conventional GOSUB

GOSUB SubName calls subroutine **SUB** *SubName*, which executes until an **EXIT SUB** or **END SUB** returns execution to the point following the call. Subroutines are more effective than functions the more the following are true:

- *A large number of arguments would have to be passed to a function.*
- *A routine uses many of the current function's* **AUTO** *variables.*
- *A routine is needed only in the current function.*
- *A routine is not very extensive.*

Some routines are better implemented as subroutines than functions. Consider converting a 20 line subroutine into a function. Many **AUTO** variables needed by the subroutine would become **SHARED** so the new function could access them. **SHARED** variables reduce encapsulation. A few variables could be passed as arguments, but passing more than a few is awkward and time consuming. In contrast, subroutine variables need not be passed or **SHARED**. All function variables are available to the subroutine without overhead. Subroutines can result in more efficient, *better structured* programs.

GOSUB *Example*

Complex decision structures can be difficult to grasp. They stretch over many lines, and executed lines are often scattered throughout the structure. A subroutine call can replace several lines with a single name that describes the routine. The size of the decision structure shrinks considerably, and descriptive routine names replace lines of code whose purpose could not be understood at a glance.

```
SELECT CASE message$
  CASE "CloseWindow"    : GOSUB CloseWindow
  CASE "DisplayWindow"  : GOSUB DisplayWindow
  CASE "ResizeWindow"   : GOSUB ResizeWindow
  CASE "HideWindow"     : GOSUB HideWindow
  CASE ELSE             : GOSUB UnknownMessage
END SELECT
```

Computed GOSUB

Computed **GOSUB** statements call subroutines whose addresses are in **SUBADDR** variables and arrays.

```
GOSUB @subVar
GOSUB @subArray[i]
```

GOSUB @subVar calls the subroutine address in **SUBADDR** variable **subVar**. **GOSUB @subArray[i]** calls the subroutine address in element **i** of **subArray[]**. If **subVar** or **subArray[i]** is zero, no subroutine is called.

The **SUBADDRESS()** intrinsic returns subroutine addresses to be assigned to **SUBADDR** variables and arrays as follows:

```
subVar = SUBADDRESS (SubName)
subArray[i] = SUBADDRESS (SubName)
```

Computed **GOSUB** statements are especially efficient when one of a number of actions must be performed depending on some condition. The following computed **GOSUB** calls one of eight subroutines depending on a three bit field in **token**:

```
GOSUB @subAction[token{3,29}]
```

Conventional Function Call

Function calls invoke functions that perform some action and possibly return a value. Functions are called by naming them, and passing arguments between the parentheses following the function name:

```
time$ = GimmeTime$ ( format )
SetTimeGMT ( currentGMT )
```

Computed Function Call

Computed function calls work like computed **GOSUBs**, except a function is called, not a subroutine. Return type and argument types are specified for **FUNCADDR** variables when they are declared. Function calls through **FUNCADDR** variables work like conventional function calls, except the address of the called function is taken from the **FUNCADDR** variable. The syntax for declaring **FUNCADDR** variables is:

```
[scopeName] FUNCADDR [returnTypeName] funcaddrVariable (typeNameList)
[scopeName] FUNCADDR [returnTypeName] funcaddrArray[] (typeNameList)
```

When a function is called through a **FUNCADDR** variable, the value returned by the called function is the data type specified in the declaration statement. The called function also expects to receive arguments as specified in the declaration statement. The compiler normally makes sure the arguments and return type of functions assigned to **FUNCADDR** variables and arrays are compatible with their declarations. The compiler also makes sure the correct number of arguments is supplied in the invocation, and uses the declaration to type-convert arguments as necessary. Several computed function calls are shown below:

```
@funcVar ()
j = @funcVar (n)
@funcArray[i] (x#, y#)
k = @funcArray[i] (i, j, k)
```

@funcVar() calls the function whose address is in variable **funcVar**. **@funcArray[x](n)** calls the function whose address is in element **x** of array **funcArray[]**. If **funcVar** or **funcArray[x]** is zero, no function is called, a return value of zero is simulated, and execution continues with the next statement. **FUNCADDR** variables can be tested for zero with conventional test statements like **IF**.

```
IF (funcArray[n]) THEN
  j = @funcArray[n] (arg1, arg2)
ELSE
  j = -1
END IF
```

The **FUNCADDRESS()** intrinsic, or **&** address operator, returns function addresses which can be loaded into **FUNCADDR** variables and arrays as follows:

```
funcVar = &FuncName()
funcVar = FUNCADDRESS (FuncName())
subArray[i] = &FuncName()
subArray[i] = FUNCADDRESS (FuncName())
```

Computed function calls are especially efficient when one of several functions must be called based on a variable or condition. For example, the following computed function call to execute one of eight functions whose addresses are in **funcAction[]** depending on a 3-bit field in variable **token**:

```
result = @funcAction [token{3,29}] ()
```

Decisions

There are types of decisions, *two-way, many-way*, plus the already discussed *computed decisions*.

IF *statement*

The **IF** statement is the most basic decision mechanism, providing for a simple choice between two possibilites. In its simplest form, without the **ELSE** part, the **IF** statement executes a block of statements if a variable or expression is **TRUE** (non-zero), or skips it is **FALSE** (zero). In its full form, with the **ELSE** part, the **IF** statement executes the first block of statements if a variable or expression is **TRUE**, or the second block if it is **FALSE**.

The following example shows the two kinds of *2-way* decisions that can be built with **IF** statements.

```
FUNCTION  IfDemo(x, y)
  IF (x < y) THEN x = y: RETURN (-1)
  IF (x > y) THEN RETURN (0) ELSE y = x: RETURN (+1)
END FUNCTION
```

In the first **IF** statement, **x=y** and **RETURN(-1)** are executed if **(x<y)** is **TRUE** (x is less than y), otherwise nothing is executed and program execution continues with the next source line.

In the second **IF** statement, **RETURN(0)** is executed if **(x>y)** is **TRUE** (x is greater than y), otherwise **y=x : RETURN(+1)** executes.

If an executable statement follows **THEN** on the same source line, a one-line **IF** statement is assumed and the end-of-line serves as an implicit **END IF.**

Otherwise a multi-line block structured **IF** is assumed and an explicit **END IF** is required to end it, as in the following example:

```
FUNCTION  IfDemo(x, y)
  IF (x < y) THEN
    x = y
    RETURN (-1)
  END IF
'
  IF (x > y) THEN
    RETURN (0)
  ELSE
    y = x
    RETURN (+1)
  END IF
END FUNCTION
```

Simple **IF** statements, like **IF (x<y) THEN x=y**, are normally written on one line. When more than one statement follows **THEN**, or when an **ELSE** section follows the **THEN** section, multiline form is generally more readable and less error prone.

`IF` *and* `IFZ`

`IF` statements test the following expression for `TRUE` (non-zero), as does the synonym statement `IFT` (if `TRUE`). Quite often, however, it is more natural to test for `FALSE`. `IFZ` (if zero) and `IFF` (if false) are equivalent and provided to test for zero aka false. The following examples illustrate all forms of the `IF` statement.

```
FUNCTION  IfDemo(x, y) lines=none
  IF    x    THEN PRINT "x    is TRUE,  (non-zero)"
  IFT   x    THEN PRINT "x    is TRUE,  (non-zero)"
  IFF   x    THEN PRINT "x    is FALSE, (zero)"
  IFZ   x    THEN PRINT "x    is ZERO,  (zero)"
  IF    x$   THEN PRINT "x$   is TRUE,  (non-empty-string)"
  IFT   x$   THEN PRINT "x$   is TRUE,  (non-empty-string)"
  IFF   x$   THEN PRINT "x$   is FALSE, (empty-string)"
  IFZ   x$   THEN PRINT "x$   is ZERO,  (empty-string)"
  IF    x[]  THEN PRINT "x[]  is TRUE,  (non-empty-array)"
  IFT   x[]  THEN PRINT "x[]  is TRUE,  (non-empty-array)"
  IFF   x[]  THEN PRINT "x[]  is FALSE, (empty-array)"
  IFZ   x[]  THEN PRINT "x[]  is ZERO,  (empty-array)"
  IF    f()  THEN PRINT "f()  returned TRUE,  (non-zero)"
  IFT   f()  THEN PRINT "f()  returned TRUE,  (non-zero)"
  IFF   f()  THEN PRINT "f()  returned FALSE, (zero)"
  IFZ   f()  THEN PRINT "f()  returned ZERO,  (zero)"
END FUNCTION
```

SELECT CASE

SELECT CASE statements select zero or more of many alternatives. These multi-line block structures have sufficient flexibility and options that they are appropriate for all but the simplest decisions.

Syntax

```
SELECT CASE [ALL] <test-expression>
  CASE <case-expressions>                ' any number of these
        ...  zero or more statements
  CASE <case-expressions>                ' another one
        ...  zero or more statements
  CASE ELSE                              ' only one CASE ELSE
        ...  zero or more statements
END SELECT
```

Test Expression

In the **SELECT CASE** statement, the *test-expression* can be **TRUE**, **FALSE**, a numeric expression, or a string expression. As they are encountered, the values in the **CASE** statements are compared with this test expression. For example, in a block that begins with **SELECT CASE x** , the values of the subsequent **CASE** expressions are compared with the value **x** had when **SELECT CASE x** was executed. In this example, **x** is the test-expression.

SELECT CASE *- zero or one of many*

When a match between the test-expression and a case-expression is found, the code following the **CASE** statement is executed, until a **CASE**, **EXIT SELECT**, **CASE ELSE**, or **END SELECT** is executed. This is a *one of many* test, because *at most one* CASE block is executed.

If the test expression doesn't equal any of the **CASE** expressions and there is no **CASE ELSE**, none are executed *(none of many)*.

The last **CASE** statement can be **CASE ELSE**. If the **CASE ELSE** is reached, the code following it will be executed. This catch-all statement can be used to eliminate the *none of many* possibility, and is useful for catching unexpected conditions.

SELECT CASE ALL *- n of many*

The **ALL** option can appear in a **SELECT CASE** statement to create an *n of many* test. The code following *all* **CASE** statements having a case-expression matching the test-expression will be executed. **CASE ELSE** is not compatible with the **ALL** option, as no way exists to know whether a match was found.

A **CASE ALL** can be used in place of **CASE ELSE** however, and as the name implies, the code following **CASE ALL** is executed if the **CASE ALL** is reached.

SELECT CASE TRUE

When the **TRUE** option is used in the **SELECT CASE** statement, **CASE** expressions match if they are **TRUE**. The **TRUE** option can be used with any mix of numeric and string cases. **ALL** can be combined with **TRUE** if desired.

```
  SELECT CASE TRUE
  SELECT CASE ALL TRUE
```

SELECT CASE FALSE

The **FALSE** option works like the **TRUE** option except the **CASE** expressions are tested for **FALSE** instead of **TRUE**. **SELECT CASE FALSE** is similar to **SELECT CASE 0**, but it is faster and can test any mix of numeric and string types.

```
  SELECT CASE FALSE
  SELECT CASE ALL FALSE
```

Example 1

With the **TRUE** option, any combination of test conditions can be accomodated, as demonstrated below:

```
SELECT CASE TRUE
  CASE  a:                  PRINT "Variable 'a' is TRUE"
  CASE  a*b+c*d:            PRINT "Expression 'a*b+c*d' is TRUE"
  CASE  (x < y), (y < z):   PRINT "(x < y) or (y < z) is TRUE"
  CASE  p$, q$, r$, s$:     PRINT "At least 1 string has contents"
  CASE  a[], b[], c[]:      PRINT "At least 1 array has contents"
  CASE  j, r$, p[a-b]:      PRINT "At least 1 is TRUE"
  CASE  j, k, l:            PRINT j, k, l
  CASE  humidity > 100:     PRINT "The sky is falling."
  CASE  a$, b$+c$:          PRINT "a$ or b$+c$ or both empty."
  CASE  !raining:           PRINT "You can go out now."
  CASE  ERROR(-1):          PRINT "There's an ERROR."
  CASE  hope[]:             PRINT "There is hope[]."
  CASE  x$ < CHR$(x):       PRINT "x string < x number."
  CASE  ELSE:               PRINT "Nothing tested is true."
END SELECT
```

Example 2

In the following example, **ALL** *cases that are true* will execute.

```
SELECT CASE ALL TRUE
  CASE  j, k, l:          PRINT j, k, l
  CASE  humidity > 100:   PRINT "The sky is falling."
  CASE  a$, b$+c$:        PRINT "a$ or b$+c$ not empty."
  CASE  !raining:         PRINT "You can go out now."
  CASE  ERROR(-1):        PRINT "There's an ERROR."
  CASE  hope[]:           PRINT "There is hope[]."
  CASE  x$ < CHR$(x):     PRINT "x string < x number."
  CASE  ALL:              PRINT "All printed are true."
END SELECT
```

EXIT SELECT

EXIT SELECT transfers execution past the matching **END SELECT** statement, bypassing any **CASE ELSE** or **CASE ALL** statements.

NEXT CASE

NEXT CASE transfers execution directly to the next **CASE** statement where execution continues as if no **CASE** statement had matched. **NEXT CASE** transfers execution *to* the next **CASE** statement, not into the block, so the next **CASE** block and subsequent **CASE** blocks are executed only if one of its case-expressions matches the test expressions.

ELSEIF

There is no **ELSEIF** statement because **SELECT CASE TRUE** is equivalent, more flexible, better structured, easier to read, more compact, and faster.

Loops and Iteration

Two *iteration* or *loop* structures are supported:

```
  DO...LOOP       DO loops
  FOR...NEXT      FOR loops
```

DO ... LOOP

DO loops are the most flexible form of interation. **WHILE** and **UNTIL** can be appended to **DO**, **LOOP**, both, or neither.

DO *Options*

The order of execution in **DO** loops is altered when any of the following statements execute:

```
  DO DO          continue at the top (the DO)
  DO LOOP        continue at the bottom (the LOOP)
  EXIT DO        exit (past the LOOP)
```

The following example illustrates these options:

```
  DO WHILE (x < y)            ' jump past LOOP if (x < y)
    Shuffle (@k, @n, @z)      ' new values for k, n, z
    PRINT "Start DO"          '
    IF z THEN DO LOOP         ' jump down to LOOP
    PRINT "z = 0"             '
    IF k THEN DO DO           ' jump up to DO
    PRINT "k = 0"             '
    IF n THEN EXIT DO         ' jump past LOOP
    PRINT "n = 0"             '
  LOOP UNTIL (j > p)          ' jump to DO if (j > p)
```

EXIT DO *Example*

The following example utilizes two **DO** loops. The outer **DO** loop has no test conditions on the **DO** or **LOOP** statements. This would cause an endless loop condition if not for the **EXIT DO** that executes when **testString$** is empty.

```
FUNCTION PlayTheHashGame ()
  $BYTE0 = BITFIELD (8, 0)
  DO
    testString$ = INLINE$ ("Compute hash for string  ===>> ")
    IFZ testString$ THEN EXIT DO
    stringLength = LEN (testString$)
    byteOffset = 0
    hash = 0
    DO WHILE (byteOffset < stringLength)
      testByte = testString${byteOffset}
      hash = hash + testByte
      INC byteOffset  ' DANGER: don't forget this line !!!
    LOOP
    hash = hash{$BYTE0}
    PRINT "The hash for "; testString$; " is: "; HEXX$(hash, 2)
  LOOP
  PRINT "*****  DONE  *****"
END FUNCTION
```

FOR . . . NEXT

FOR loops eliminate a common programming error that occurs with **DO** loops, namely failure to update a *loop variable* or *test variable.*

If **INC byteOffset** is left out of the preceeding example, the 1st byte of **testString$** is added to **hash** indefinitely, instead of each byte being added once as intended. The following example performs the same function, except the inner **DO** loop is replaced by a **FOR** loop. **byteOffset** is incremented automatically by the **NEXT** statement, avoiding the possibility of an inadvertent endless loop.

FOR *Example*

```
FUNCTION PlayTheHashGame ()
  $BYTE0 = BITFIELD (8, 0)
  DO
    testString$ = INLINE$("Compute hash for string  ===>> ")
    hash = 0
    IF testString$ THEN EXIT DO
    FOR byteOffset = 0 TO LEN (testString$) - 1
      testByte = testString${byteOffset}
      hash = hash + testByte
    NEXT byteOffset
    hash = hash{$BYTE0}
    PRINT "The hash for "; testString$; " is: "; HEXX$(hash, 2)
  LOOP
  PRINT "*****  DONE  *****"
END FUNCTION
```

FOR *Options*

The order of execution in **FOR** loops is altered when any of the following statements execute:

```
  DO FOR        continue at the top (the FOR)
  DO NEXT       continue at the bottom (the NEXT)
  EXIT FOR      exit (past the NEXT)
```

The following example illustrates these options:

```
  FOR i = j TO k STEP x     '
    Shuffle (@x, @y, @z)    ' new values for x, y, z
    PRINT "Start FOR"       '
    IF x THEN DO NEXT       ' jump down to NEXT statement
    PRINT "x = 0"           '
    IF y THEN DO FOR        ' jump up to FOR statement
    PRINT "y = 0"           '
    IF z THEN EXIT FOR      ' jump past NEXT statement
    PRINT "z = 0"           '
  NEXT i                    '
```

STEP

If no **STEP** is given, the step size defaults to **1**.

File Processing

Overview

Many programs need to save and/or access information from permanent devices, usually hard disks. File I/O statements and intrinsics built in to the language create, access, manipulate, and close files. More elaborate I/O functions are easily built from these capabilities - the standard library contains many.

File Number

Programs refer to a specific file by means of the *file number* returned by **`OPEN()`**. Until a file has been opened, programs cannot access its contents.

File Pointer

A *file pointer* variable is maintained for every open file. It points to a byte location within the file. When a file is opened, its file pointer is initialized to zero. When a file is read or written, the file pointer is advanced to the byte after the data read or written. Since the file pointer is a byte offset from the beginning of the file, the first byte of a file is at file position zero. The file position intrinsics are:

Name	Meaning	Returns
EOF()	End of file	TRUE if file pointer past last byte
POF()	Position of file	File pointer
LOF()	Length of file	Number of bytes (position of last byte)
SEEK()	Move file pointer	Moves file pointer to specified offset

OPEN

Files must be *opened* by **`OPEN()`** before their contents can be accessed. **`OPEN()`** takes two arguments, the name of the file, and an open mode. The mode determines whether the file is open for reading, writing, or both. It also specifies non-standard behavior such as opening a fresh version of the file for writing (delete any existing copies and start with an empty file), versus work with the existing contents.

$$RD	Open file for reading only
$$WR	Open file for writing only
$$RW	Open file for reading and writing
$$WRNEW	Open file for writing only (delete existing)
$$RWNEW	Open file for reading and writing (delete existing)
$$RDSHARE	Open file for reading only - other programs can also open the file
$$WRSHARE	Open file for writing only - other programs can also open the file
$$RWSHARE	Open file for reading and writing - other programs can open the file

`OPEN()` returns an **`XLONG`** *filenumber*. All file references, whether use filenumbers to identify the file to operate on. File numbers, not file names, are the key to accessing opened files.

CLOSE

`CLOSE()` complements **`OPEN()`**. **`CLOSE()`** *closes* a file and releases its file number.

INFILE$()

`INFILE$()` reads a line of text from a file. For example, **`stringVar$=INFILE$(file)`** reads bytes from an open **`file`** into **`stringVar$`** up to the next newline character, or the end of the file, whichever comes first. The newline character is not put in the string, but the file pointer is moved beyond it.

`INFILE$()` reads newline-terminated strings of the kind written by **`PRINT [filenumber],`** statements.

READ

`READ [`*filenumber*`],` *variable-list*

The `READ` statement reads binary data from an open file into one or more variables. `READ` reads the number of bytes into each variable that is appropriate to the type of the variable. For example, one byte is read into `UBYTE` variables, whether they are simple variables or elements of arrays or composites. Since simple `UBYTE` variables are held as 32-bit or 64-bit values, `READ` reads a byte and converts it into the 32-bit or 64-bit value before storing it in the variable. The extra bits of unsigned variables are filled with zeros, while the extra bits of signed variables are filled with the most-significant bit of the value read from the file.

`READ` also reads data into strings, arrays, composites, and portions of composites. Bytes are read from the file to fill the variable or component, unless the end of file is reached first. For example, if `abc$` contains six bytes, `READ [n], abc$` will read six bytes from the file into `abc$`.

WRITE

`WRITE [`*filenumber*`],` *variable-list*

The `WRITE` statement writes binary data from one or more variables to file filenumber. `WRITE` writes the number of bytes appropriate to the type of the variable. For example, one byte is written for `UBYTE` variables, whether they are simple variables or elements of arrays or composites. Though simple `UBYTE` variables are held as 32-bit or 64-bit values, `WRITE` writes only the least significant byte. This has the effect of clipping off any higher bits that may exist. Where adverse effects are possible when reading back out-of-range values, programs should range check the values with type-conversion intrinsics before writing the variables to disk.

`WRITE` also writes strings, arrays, composites, and components of composites. All bytes in the variable, array, composite, or component are written to the file.

READ *and* **WRITE** *Simple Variables, Strings, Arrays*

READ and **WRITE** are binary I/O statements. They read and write data without alteration, interpretation, or conversion.

Strings are written without a length, and neither null terminators nor newline characters are appended. Only the string data itself is written. If a string is empty, nothing is written. Arrays are handled much like strings. No length is written, only the array data.

Except for arrays and strings, all variables including those of user-defined composite types, have a known, fixed size. Reading fixed-size variables from a file is simple. Reading into variables of the same types that were written guarantees correct operation.

Arrays and strings of known size can be written much like fixed-size variables. Before they are read back correctly, however, the array or string must be sized properly. Arrays and strings can be sized by **DIM** and **NULL$()** in preparation for a **READ**.

When arrays or strings of unknown size or type must be stored in a binary file, additional information must be written before the data.

For example, when a variable size string is written, an **SLONG** variable containing the number of elements could be written before the string. Before the string is read, the number of elements is read into an **SLONG** variable, which is used in **NULL$()** to size string appropriately for the read.

For example:

```
  READ [ifile], n
  a$ = NULL$ (n)
  READ [ifile], a$
'
  READ [ifile], n
  IF (n <= 0) THEN
    DIM a[]
  ELSE
    DIM a[n-1]
    READ [ifile], a[]
  END IF
```

READ *and* **WRITE** *Composite Variables*

Components of composite variables can be read or written with **READ** and **WRITE**. Components can be simple variables, fixed-size strings, fixed-size arrays, or array elements. The follow code segment illustrates these combinations:

```
'
' *****  PROLOG  *****
'
TYPE POINT                           ' User defined type POINT
  XLONG  .x                          ' component .x
  XLONG  .y                          ' component .y
  XLONG  .z                          ' component .z
END TYPE
'
TYPE LINE
  POINT  .a                          ' POINT in the line
  POINT  .b                          ' POINT in the line
  USHORT .color[2]                   ' three color values  (r, g, b)
END TYPE
'
TYPE BOX
  LINE      .top                      ' top line in box
  LINE      .left                     ' left line in box
  LINE      .right                    ' right line in box
  LINE      .bottom                   ' bottom line in box
  STRING * 20 .name                   ' name of this box
END TYPE
'
DECLARE FUNCTION ReadWrite (o, i)     '
  ...                                 '
FUNCTION ReadWriteDemo (o, i)         '
  SHARED BOX  box0, box1              '
  SHARED LINE  a, b, c, d             '
  SHARED POINT  a0, a1, b0, b1        '
'                                     '
  WRITE [o], box0                     ' write entire BOX variable
  WRITE [o], box1.top                 ' write .top LINE component
  WRITE [o], box1.name                ' write .name STRING * 20
  WRITE [o], box0.top.color[]         ' write .color array
  WRITE [o], box1.left.color[n]       ' write .color array element
'
  READ [i], box1                      ' read entire BOX variable
  READ [i], box0.left                 ' read .left LINE component
  READ [i], box0.name                 ' read .name STRING * 20
  READ [i], box0.right.color[]        ' read .color array
  READ [i], box0.bottom.color[1]      ' read .color array element
END FUNCTION
```

Errors

Compile Time Errors

When programs under development are compiled, the compiler sometimes finds text that is not valid code. Typing mistakes and upper/lower case errors are most common.

When the compiler detects an error in the source program, it prints the line containing the error, points at the position of the error on the line, and displays an error message like *syntax error* or *type mismatch*.

The error pointer is not always accurate. Sometimes the compiler does not realize there is an error until later on the line, and does not know where the error begins.

The general purpose compile-time error is *syntax error*, which means that something invalid or unexpected was encountered on the source line. In many cases, the compiler gives more helpful messages like *type mismatch* or *overflow*.

The compile-time error messages are listed and described in the following section.

After CASE ELSE	**`CASE`** statement after **`CASE ELSE`** or **`CASE ALL`**.
Bad CASE ALL	**`CASE ALL`** without **`ALL`** in **`SELECT CASE`** statement.
Bad GOSUB destination	**`GOSUB`** something other than a subroutine.
Bad GOTO destination	**`GOTO`** something other than **`GOTO`** label.
Bad Symbol	A symbol that is valid in some contexts appears in an invalid context.
Bad Bitspec	*Width* not 0 to 32 or *Offset* not 0 to 31.
Bad Pass By Reference	Pass by reference not supported. Only individual variables can be passed by reference, so it is invalid to attempt to pass expressions by reference. It is invalid to pass by reference to C functions, because different implementations of C allocate arguments differently.
Bad Pass By Value	Pass by value not supported. Arrays cannot be passed by value.
Compiler Error	Error possibly due to error in compiler. If an error cannot be found in the source code, these errors should be reported so fixes can be included in future releases. Be sure to isolate as narrowly as possible the source code necessary to create this error.
Component Error	Poorly formed component of composite variable, or a component name that was never defined for the composite type.
Crossed Functions (X/S/C)	Declared a native function but defined with **`CFUNCTION`** line, or function declare as C function but defined with **`FUNCTION`** line.
DECLARE after SHARED	In the PROLOG, **`SHARED`** statements must appear after the last function is declared, so **`DECLARE`** statements after **`SHARED`** statements are errors.
DECLARE too late	Attempt to declare function after **`FUNCTION`** or **`SHARED`** statement.
Duplicate Declaration	Attempt to declare a function or variable twice.
Duplicate Definition	Attempt to define the same function twice.
Duplicate Label	Attempt to have two labels of the same name in a function.
Duplicate Type	Attempt to declare the same composite type twice.
CASE ELSE in CASE ALL	**`CASE ELSE`** following **`CASE ALL`** in same **`SELECT CASE`** block.
Entry Function Error	Attempt to declare entry function in an **`EXTERNAL`** statement.
Expect Assignment	An assignement was expected but not found.
Expression Error	Expression doesn't resolve properly. Expression error or bug.
Internal / External	Attempt to define a function that was declared **`EXTERNAL`**.
Kind Mismatch	The kind of a language element is incorrect. For example, it is a kind-mismatch error to attempt to pass a variable as a function argument when an array argument was declared.
Literal Error	Error involving a literal number or string. For example, an attempt to take the handle address of a literal number or string is a literal error.

Need Excess Comma	Excess comma notation required in **`ATTACH`**, **`SWAP`**, or elsewhere.
Need Subscript	One dimensional array subscripts are required in composite array component declarations.
Nesting Error	The kind of block structure being ended is not properly nested. Note that errors in **`IF`** statement and other block structures may cause subsequent erroneous reporting of nesting errors. The termination of statement processing by an error blinds the compiler to the termination of a block structure on the same line. This is especially common with **`IF`** statements. When the compiler reports many nesting errors, all errors other than nesting errors should be fixed, then another compilation attempted before attention is paid to nesting errors. The reporting of bad nesting error messages generally terminates when **`END FUNCTION`** is reached, as nesting variables are all reset to zero.
Node / Data Mismatch	Attempt to mix an array node with data in **`ATTACH`** or **`SWAP`**.
Outside Functions Error	Attempt to put an executable statement outside the bounds of a function.
Overflow Error	A value is outside the acceptable range.
Register Address	Attempt to take the address of a variable in a register, or the handle of a variable whose handle is in a register.
Duplicate Type Spec	Attempt to define the same composite type more than one time.
Scope Mismatch	Attempt to declare a **`GOADDR`** or **`SUBADDR`** variable in an **`EXTERNAL`** or **`SHARED`** statement. Since **`GOTO`** labels and subroutines are limited to local execution (from within) the function, variables and arrays that hold these addresses should never be visible outside the function.
Sharename Error	Invalid /sharename/ field in type declaration statement.
Syntax Error	General purpose error to denote invalid syntax.
Too Few Arguments	Too few arguments to an intrinsic or function.
Too Late	Attempt to declare variables or constants in a function after an executable statement. Following the **`FUNCTION`** statement are variable declarations followed by local constant declarations followed by executable statement.
Too Many Arguments	Too many arguments to an intrinsic or function.
Type Mismatch	The type of a variable is improper or incompatible with the type of another variable in the same statement.
Undeclared	Attempt to define or call an undeclared function.
Undefined	Attempt to call an undefined function.
Unimplemented	Attempt to executed a statement, intrinsic, or language feature that is planned but not yet implemented, or no longer implemented.
Within Function	**`FUNCTION`** statement within another function statement.

Runtime Errors and Exceptions

Errors can occur while programs run. Serious errors like memory faults are unexpected *exceptions*, while other *errors*, like overflow caused by improper user input are forseeable and even preventable. Errors like reaching the end of a file during a read operation may even be counted on by programs.

Runtime Errors

When an *error* occur, the program continue running. Therefore programs can, and usually should test to see if an error occured after any operation or function call that could produce an error. Programs can then take appropriate actions depending on the source and nature of each error. Depending on the situation, programs can:

- *ignore the error.*
- *retry the attempt.*
- *try to perform the operation in another way.*
- *request user assistance to work around the problem.*
- *terminate the program.*

`ERROR()` *and* **`ERROR$()`**

When an error occurs, an error number is assigned to an internal **`##ERROR`** system variable, which can be quickly and efficiently examined and/or modified with the **`error = ERROR(newError)`** intrinsic. **`error`** is the existing value of **`##ERROR`** returned by **`ERROR()`**, and **`newError`** is a new error number assigned to **`##ERROR`**. Set **`newError`** to **`0`** to clear the error to **`0`** to ready **`##ERROR`** for the next error, to **`-1`** to leave **`##ERROR`** unaltered, or to an appropriate error number to report an error.

`error$ = ERROR$(error)` converts **`error`** into an error string.

Runtime Error Handling

Many programs follow operations that can produce an error with:

```
IF ERROR(-1) THEN RETURN  ' -1 means don't change value in ##ERROR

... or

IF ERROR(-1) THEN         ' detect error (no change to ##ERROR)
  error = ERROR(0)        ' clear ##ERROR
  ... error
  ... handler
  ... code
END IF

... or

error = ERROR ($$ErrorNatureOverflow)  ' get old error & set new one
```

Error Numbers

The *error numbers* assigned to `##ERROR` have two parts. `$$ErrorObject` is an unsigned byte value in the next to lowest byte that identifies the source of the error, while `$$ErrorNature` is an unsigned byte value in the low byte to identify the kind of error. Error values assigned to `##ERROR` are therefore:

```
(($$ErrorObject << 8) + $$ErrorNature)
```

The `$$ErrorObject` and `$$ErrorNature` constants are defined in the standard library file "`xst.dec`".

Programmer Defined Errors

Programs can combine their own error handling with the built-in error handling by assigning *programmer-defined error numbers* to `##ERROR` whenever they detect an error. In this way, one quick

```
IF ERROR(-1) THEN ...
       or
error = ERROR(0)
IF error THEN ...
```

will test for any system or program generated detected error.

Consider a function that opens a file and reads some formatted data. If there are errors opening the file or reading it, the function could simply `RETURN` and expect the calling function to find the error in `##ERROR`. On the other hand, the function might not encounter errors opening, reading, and closing the file, but might find improperly formatted data in the file.

The function could return an error message in `##ERROR` to indicate the error condition, for example:

```
error = ERROR($$ErrorObjectFile << 8 OR $$ErrorNatureInvalidFormat)
```

If none of the pre-defined messages is appropriate, a program can define its own `$$ErrorObject` and/or `$$ErrorNature` constants to create its own error numbers. These numbers should start at 255 and work lower to assure they won't conflict with values defined in later versions of the standard library.

Runtime Exceptions

When runtime *exceptions* occur, programs do not continue running. An *exception handler function* is executed immediately, before even the next machine instruction executes, much less the next statement.

When a program is running in the PDE, the exception handler invokes the PDE, which reports the error in a runtime error window. The program has not been terminated yet, however, so debug features like the function call window and variables window are available to help determine the cause of the exception. It's even possible to fix certain problems and let the program continue in some situations. For example, if **`a=b/c`** causes a divide-by-zero exception because **`c=0`**, you can set **`c`** to a non-zero value in the variable window and continue execution of the program. You could also move the current execution pointer to the next line with **`DebugJump`** to skip the divide statement completely.

By default, standalone programs and libraries terminate upon exceptions, though they can install their own exception handler to work around many exceptions or at least terminate more gracefully - see **`XstGetExceptionFunction()`** and **`XstSetExceptionFunction()`** in the standard library.

```
$$ExceptionNone
$$ExceptionSegmentViolation
$$ExceptionOutOfBounds
$$ExceptionBreakpoint
$$ExceptionBreakKey
$$ExceptionAlignment
$$ExceptionDenormal
$$ExceptionDivideByZero
$$ExceptionInvalidOperation
$$ExceptionOverflow
$$ExceptionStackCheck
$$ExceptionUnderflow
$$ExceptionInvalidInstruction
$$ExceptionPrivilege
$$ExceptionStackOverflow
$$ExceptionReserved
$$ExceptionUnknown
$$ExceptionUpper
```

Appendix A : Standard Character Set

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	00	<null>	32	20	<space>	64	40	@	96	60	`
1	01		33	21	!	65	41	A	97	61	a
2	02		34	22	"	66	42	B	98	62	b
3	03		35	23	#	67	43	C	99	63	c
4	04		36	24	$	68	44	D	100	64	d
5	05		37	25	%	69	45	E	101	65	e
6	06		38	26	&	70	46	F	102	66	f
7	07	<alarm>	39	27	'	71	47	G	103	67	g
8	08	<back>	40	28	(	72	48	H	104	68	h
9	09	<tab>	41	29	)	73	49	I	105	69	i
10	0A	<newline>	42	2A	*	74	4A	J	106	6A	j
11	0B		43	2B	+	75	4B	K	107	6B	k
12	0C		44	2C	,	76	4C	L	108	6C	l
13	0D	<return>	45	2D	-	77	4D	M	109	6D	m
14	0E		46	2E	.	78	4E	N	110	6E	n
15	0F		47	2F	/	79	4F	O	111	6F	o
16	10		48	30	0	80	50	P	112	70	p
17	11		49	31	1	81	51	Q	113	71	q
18	12		50	32	2	82	52	R	114	72	r
19	13		51	33	3	83	53	S	115	73	s
20	14		52	34	4	84	54	T	116	74	t
21	15		53	35	5	85	55	U	117	75	u
22	16		54	36	6	86	56	V	118	76	v
23	17		55	37	7	87	57	W	119	77	w
24	18		56	38	8	88	58	X	120	78	x
25	19		57	39	9	89	59	Y	121	79	y
26	1A		58	3A	:	90	5A	Z	122	7A	z
27	1B		59	3B	;	91	5B	[	123	7B	{
28	1C		60	3C	<	92	5C	\	124	7C	|
29	1D		61	3D	=	93	5D	]	125	7D	}
30	1E		62	3E	>	94	5E	^	126	7E	~
31	1F		63	3F	?	95	5F	_	127	7F	

Appendix B : Translating Programs

BASIC	Native	Native Comments
ABS()	ABS()	More types
ACCESS		
ALIAS		
AND	AND	More types
ANY		
APPEND		
AS		
ASC()	ASC()	More capable
ATN()	ATAN()	Math function library
BEEP		
BINARY		
BLOAD	XstBinLoad()	also READ
BSAVE	XstBinSave()	also WRITE
BYVAL		Functions are BYVAL by default
CALL		
CALLS		
CASE	CASE	More capable
CCUR()	GIANT()	More intuitive, capable
CDBL()	DOUBLE()	More intuitive, capable
CDECL	CFUNCTION	More intuitive, capable
CHAIN		
CHDIR		
CHDRIVE		
CHR$()	CHR$()	More capable
CINT()	SSHORT()	More intuitive, capable
CIRCLE		See GraphicsDesigner
CLEAR		
CLNG()	SLONG()	More intuitive, capable
CLOSE	CLOSE()	
CLS		See GraphicsDesigner
COLOR		See GraphicsDesigner
COM		
COMMAND$	XstGetCommandLineArguments()	
COMMON		
CONST	$$NAME =	More intuitive, visible
COS()	COS()	
CSNG()	SINGLE()	More intuitive, capable
CSRLIN()		
CURDIR$()		
CURRENCY	GIANT	More intuitive, capable
CVC()	GIANT()	More intuitive, capable
CVD()	DOUBLE()	More intuitive, capable
CVI()	SSHORT()	More intuitive, capable
CVL()	SLONG()	More intuitive, capable
CVS()	SINGLE()	More intuitive, capable

DATA		
DATE$()	XstGetDateAndTime()	
DECLARE	DECLARE	More capable
DEF		
DEFCUR	GIANT	More intuitive, capable
DEFDBL	DOUBLE	More intuitive, capable
DEFINT	SSHORT	More intuitive, capable
DEFLNG	SLONG	More intuitive, capable
DEFSNG	SINGLE	More intuitive, capable
DEFSTR	STRING	More intuitive, capable
DELETE		
DIM	DIM	More intuitive, capable
DIR$()		
DO	DO	More capable
DOUBLE	DOUBLE	More intuitive, capable
DRAW	===>>	See GraphicsDesigner
ELSE	ELSE	
ELSEIF	SELECT CASE	More capable
END	END	More capable
ENDIF	ENDIF	
ENVIRON		
ENVIRON$()	XstGetEnvironmentVariable() : XstGetEnvironmentVariables()	
EOF()	EOF()	
EQV		
ERASE		
ERDEV()		
ERDEV$()		
ERL()		
ERR()	ERROR()	Get and Set error #
ERROR		
EVENT		
EXIT	EXIT	More capable
EXP()	EXP()	Math function library
FIELD		
FILEATTR()		
FILES	XstGetFiles()	???
FIX()	FIX()	More types
FOR	FOR	
FRE()		
FREEFILE()		
FUNCTION	FUNCTION	More intuitive, capable
GET	READ	More convenient, capable
GOSUB	GOSUB	More capable
GOTO	GOTO	More capable
HEX$()	HEX$()	More capable
IF	IF	More capable
IMP		
INKEY$()		
INP()		
INPUT	INLINE$()	Also INFILE$()
INPUT$()	INLINE$()	Also INFILE$()
INSTR()	INSTR()	
INT()	INT()	More types
INTEGER	SSHORT	More intuitive
IOCTL()		
IS		
KILL		

LBOUND()		Lower bound is always 0
LCASE$()	LCASE$()	
LEFT$()	LEFT$()	
LEN()	LEN()	
LET		
LINE		See GraphicsDesigner
LIST		
LOC()		
LOCAL	AUTO, AUTOX, STATIC	
LOCATE		See GuiDesigner
LOCK	XstLockFileSection()	
LOF()	LOF()	
LOG()	LOG()	Math function library
LONG	SLONG	More intuitive
LOOP	LOOP	
LPOS()		
LPRINT		
LSET		
LTRIM$()	LTRIM$()	
MID$()	MID$(), STUFF$()	
MKC$()	STRING()	
MKD$()	STRING()	
MKI$()	STRING()	
MKL$()	STRING()	
MKS$()	STRING()	
MKDIR	XstMakeDirectory()	
MOD	MOD	
NAME		
NEXT	NEXT	
NOT	NOT	
OCT$()	OCT$(), OCTO$()	
OFF		
ON	GOTO @, GOSUB @	
OPEN	OPEN	
OPTION		
OR	OR	
OUT		
OUTPUT		
PAINT	See GraphicsDesigner	
PALETTE	See GraphicsDesigner	
PCOPY		
PEEK()	UBYTEAT()	
PMAP()		
POINT()	See GraphicsDesigner	
POKE	UBYTEAT()	
POS()		
PRESET	See GraphicsDesigner	
PRINT	PRINT	
PSET		
RANDOM		
RANDOMIZE		
READ		
REDIM	REDIM	
REM	'	
RESET		
RESTORE		
RESUME		
RETURN	END SUB	
RIGHT$()	RIGHT$()	
RMDIR		
RND()		
RSET		
RTRIM$()	RTRIM$()	

```
RUN
```

SADD()	&stringVar$	
SCREEN()		
SEEK()	SEEK()	
SEG		
SELECT	SELECT	
SETMEM()		
SGN()	SGN()	
SHARED	SHARED	
SHELL	SHELL()	
SIGNAL()	XstCauseException() ???	
SIN()	SIN()	Math function library
SINGLE	SINGLE	
SLEEP	XstSleep()	
SOUND		
SPACE$()	SPACE$()	
SPC()		
SQR()	SQRT()	Math function library
SSEG()		
SSEGADD()		
STACK()		
STATIC	STATIC	SIMILAR (More intuitive, capable)
STEP	STEP	SAME
STICK()		
STOP	STOP	
STR$()	STR$(), STRING$(), SIGNED$(), STRING()	
STRING	STRING	
STRING$()	CHR$()	
SUB	FUNCTION	
SWAP	SWAP	more capable
SYSTEM	SHELL()	???
TAB()	TAB()	
TAN()	TAN()	Math function library
THEN	THEN	
TIME$()	XstGetDateAndTime()	
TIMER()	XstStartTimer() : XstKillTimer()	
TO	TO	
TYPE	TYPE	
UBOUND()	UBOUND()	
UCASE$()	UCASE$()	
UEVENT		
UNLOCK		
UNTIL	UNTIL	
UPDATE		
USING	FORMAT$()	
VAL()	DOUBLE()	
VARPTR()	&	address operator
VARPTR$()		
VIEW	===>>	See GraphicsDesigner
WAIT		
WEND		
WHILE	DO WHILE	
WIDTH		
WINDOW	===>>	See GraphicsDesigner
WRITE	WRITE	SIMILAR
XOR	XOR	More types

Appendix C : Keywords

Keyword	Catagory	Description
ABS()	Intrinsic	Absolute Value
ALL	Auxiliary	In SELECT CASE, execute all matching cases
AND	Operator	Bitwise AND (integer only)
ASC()	Intrinsic	Numeric value of ASCII character in string
ATTACH	Statement	Attach one array/subarray to another
AUTO	Statement	Declare variable to be AUTO scope
AUTOS	Auxiliary	CLEAR AUTOS initializes AUTO and AUTOX variables
AUTOX	Statement	Declare variable to be AUTOX scope
BIN$()	Intrinsic	Binary format string of integer (010010111...)
BINB$()	Intrinsic	Binary format string of integer (0b010010111...)
BITFIELD()	Intrinsic	Define bitfield constant or variable
CASE	Statement	In SELECT CASE; test cases, execute block on match
CFUNCTION	Statement	Declare/Define C function or C callable function
CHR$()	Intrinsic	Convert ASCII numeric value to 1 byte string
CJUST$()	Intrinsic	Center justify string in field of spaces
CLEAR	Statement	CLEAR AUTOS initializes AUTO and AUTOX variables
CLOSE()	Intrinsic	Close disk, console, or communications file
CLR()	Intrinsic	Clear bit field in integer
CSIZE()	Intrinsic	Count number of bytes before 1st zero byte
CSIZE$()	Intrinsic	Clip string off at 1st null character
CSTRING$()	Intrinsic	Convert C string into native string
DCOMPLEX	Statement	Declare variables to be type DCOMPLEX
DEC	Statement	Decrement numeric variable
DECLARE	Statement	Declare function prototype
DHIGH()	Intrinsic	Extract high 32-bits from DOUBLE
DIM	Statement	Dimension an array and zero contents
DLOW()	Intrinsic	Extract low 32-bits from DOUBLE
DMAKE()	Intrinsic	Make DOUBLE from two 32-bit integers
DO	Statement	Initiate a DO...LOOP loop block
DOUBLE	Statement	Declare variables to be type DOUBLE
DOUBLE()	Intrinsic	Convert numeric or string to type DOUBLE
DOUBLEAT()	Intrinsic	Write DOUBLE value into specified memory address
DOUBLEAT()	Intrinsic	Read DOUBLE value from specified memory address
ELSE	Statement	Optional in IF...THEN...ELSE...ENDIF blocks
END	Statement	END program, function, or block structure
ENDIF	Statement	End of IF...THEN...ELSE...ENDIF block
EOF()	Intrinsic	Is file pointer beyond end of file
ERROR()	Intrinsic	Return and/or set error number
ERROR$()	Intrinsic	Convert error number to error string
EXIT	Statement	EXIT function or block structure
EXPORT	Statement	Export type, constant, function declarations
EXTERNAL	Statement	Declare variables to be EXTERNAL scope
EXTS()	Intrinsic	Extract signed bit field
EXTU()	Intrinsic	Extract unsigned bit field
FALSE	Auxiliary	Optional in SELECT CASE statements
FIX()	Intrinsic	Integerize with round towards zero
FOR	Statement	Initiate a FOR...NEXT loop block
FORMAT$()	Intrinsic	Create a formatted string from format spec and value
FUNCADDR	Statement	Declare variable to be FUNCADDR type
FUNCADDR()	Intrinsic	Convert to type FUNCADDR
FUNCADDRESS()	Intrinsic	Get address of function

GHIGH()	Intrinsic	Extract high 32-bits from GIANT
GIANT	Statement	Declare variable to be type GIANT
GIANT()	Intrinsic	Convert to type GIANT
GIANTAT()	Intrinsic	Write GIANT value into specified memory address
GIANTAT()	Intrinsic	Read GIANT value from specified memory address
GLOW()	Intrinsic	Extract low 32-bits from GIANT
GMAKE()	Intrinsic	Create GIANT from two 32-bit integers
GOADDR	Statement	Declare variable to be type GOADDR
GOADDR()	Intrinsic	Convert to type GOADDR
GOADDRESS()	Intrinsic	Get address of GOTO label
GOSUB	Statement	Call a local subroutine
GOTO	Statement	Jump to a local label
HEX$()	Intrinsic	Create hexadecimal string form of integer
HEXX$()	Intrinsic	HEX$ with "0x" prefix
HIGH0()	Intrinsic	Find bit number of most significant 0 bit
HIGH1()	Intrinsic	Find bit number of most significant 1 bit
IF	Statement	IF TRUE test in IF...ENDIF block
IFF	Statement	IF FALSE test in IF...ENDIF block
IFT	Statement	IF TRUE test in IF...ENDIF block
IFZ	Statement	IF ZERO test in IF...ENDIF block
IMPORT	Statement	Import function library
INC	Statement	Increment a variable
INCHR()	Intrinsic	Find first search-set character in string
INCHRI()	Intrinsic	Case insensitive INCHR()
INFILE$()	Intrinsic	Input string from file
INLINE$()	Intrinsic	Input string from main console
INSTR()	Intrinsic	Find first substring in string
INSTRI()	Intrinsic	Case insensitive INSTR()
INT()	Intrinsic	Integer part of number
INTERNAL	Statement	Declare function to have INTERNAL scope
LCASE$()	Intrinsic	Convert constents of string to lower case
LCLIP$()	Intrinsic	Clip n bytes from left end of string
LEFT$()	Intrinsic	Leftmost n characters of string
LEN()	Intrinsic	# of elements in string
LIBRARY	Statement	Reserved
LIBRARY()	Intrinsic	Returns TRUE if compiled as library
LJUST$()	Intrinsic	Left justify string in field of spaces
LOF()	Intrinsic	Length of file in bytes
LOOP	Statement	End DO...LOOP block
LTRIM$()	Intrinsic	Trim spaces and tabs from left of string
MAKE()	Intrinsic	Make a bit-field
MAX()	Intrinsic	Return maximum of two arguments (larger)
MID$()	Intrinsic	Extract arbitrary part of string
MIN()	Intrinsic	Return minimum of two arguments (smaller)
MOD	Operator	Arithmetic MODULUS operator
NEXT	Statement	End FOR...NEXT loop
NOT	Operator	Bitwise NOT operator (invert all bits)
NULL$()	Intrinsic	Create string of n null characters
OCT$()	Intrinsic	Octal string from integer (1234567012...)
OCTO$()	Intrinsic	Octal string from integer (0o1234567012...)
OPEN()	Intrinsic	Open a disk, console, or communications file
OR	Operator	Bitwise OR operator

POF()	Intrinsic	Position of file pointer
PRINT	Statement	Print to console, disk, or communications file
PROGRAM	Statement	Name program or END PROGRAM
PROGRAM$()	Intrinsic	Return program name defined by PROGRAM statement
QUIT()	Intrinsic	Quit executing the program (terminate program)
RCLIP$()	Intrinsic	Clip n bytes from right end of string
READ	Statement	Read from file into variables
REDIM	Statement	Redimension an array, preserve contents
RETURN	Statement	Return from function (not from GOSUB)
RIGHT$()	Intrinsic	Extract rightmost n characters of string
RINCHR()	Intrinsic	Same as INCHR except reverse search direction
RINCHRI()	Intrinsic	Same as RINCHR() except case insensitive
RINSTR()	Intrinsic	Same as INSTR() except reverse search direction
RINSTRI()	Intrinsic	Same as RINSTR() except case insensitive
RJUST$()	Intrinsic	Right justify string in field of spaces
ROTATER()	Intrinsic	Rotate word right n bits
RTRIM$()	Intrinsic	Trim spaces and tabs from right end of string
SBYTE	Statement	Declare variables to be type SBYTE
SBYTE()	Intrinsic	Convert to type SBYTE
SBYTEAT()	Intrinsic	Write SBYTE value into specified memory address
SBYTEAT()	Intrinsic	Read SBYTE value from specified memory address
SCOMPLEX	Statement	Declare variables to be type SCOMPLEX
SEEK()	Intrinsic	Set file pointer position
SELECT	Statement	Select from one of n alternatives
SET()	Intrinsic	Set specified range of bits = 1s
SFUNCTION()	Intrinsic	Declare or define a "system" function
SGN()	Intrinsic	Sign of number (-1, 0, +1)
SHARED	Statement	Declare variables to have SHARED scope
SHELL()	Intrinsic	Execute a command line string
SIGN()	Intrinsic	Sign of number (-1, +1)
SIGNED$()	Intrinsic	Convert to type STRING (leading "-" or "+")
SINGLE	Statement	Declare variables to be type SINGLE
SINGLE()	Intrinsic	Convert to type SINGLE
SINGLEAT()	Intrinsic	Write SINGLE value into specified memory address
SINGLEAT()	Intrinsic	Read SINGLE value from specified memory address
SIZE()	Intrinsic	Size of string or array in bytes
SLONG	Statement	Declare variables to be type SLONG
SLONG()	Intrinsic	Convert to type SLONG
SLONGAT()	Intrinsic	Write SLONG value into specified memory address
SLONGAT()	Intrinsic	Read SLONG value from specified memory address
SMAKE()	Intrinsic	Make a type SINGLE from a 32-bit integer value
SPACE$()	Intrinsic	Create a string of n space characters
SSHORT	Statement	Declare variables to be type SSHORT
SSHORT()	Intrinsic	Convert to type SSHORT
SSHORTAT()	Intrinsic	Write SSHORT value into specified memory address
SSHORTAT()	Intrinsic	Read SSHORT value from specified memory address
STATIC	Statement	Declare variables to be STATIC scope
STEP	Auxiliary	Increment size in FOR.TO.STEP...NEXT blocks
STOP	Statement	Stop program execution here
STR$()	Intrinsic	Convert to type STRING (leading "-" or " ")
STRING	Statement	Declare variables to be type STRING
STRING()	Intrinsic	Convert to type STRING (leading "-" or "")
STRING$()	Intrinsic	Convert to type STRING (leading "-" or "")
STUFF$()	Intrinsic	Stuff one string into another
SUB	Statement	Begin a subroutine
SUBADDR	Statement	Declare variables to be type SUBADDR
SUBADDR()	Intrinsic	Convert to type SUBADDR
SUBADDRESS()	Intrinsic	Get address of subroutine
SWAP	Statement	Swap the values of the same type

TAB()	Intrinsic	Append spaces to PRINT string to get to column n
THEN	Auxiliary	Used in IF...THEN...ELSE...ENDIF blocks
TO	Auxiliary	Used in FOR...NEXT blocks and ATTACH statements
TRIM$()	Intrinsic	Remove spaces and tabs from left & right of string
TRUE	Statement	Optional in SELECT CASE statements
TYPE	Statement	Declare/Define user-defined composite data type
TYPE()	Intrinsic	Return type number of variable, array, component
UBOUND()	Intrinsic	Upper bound of any dimension of an array
UBYTE	Statement	Declare variables to be type UBYTE
UBYTE()	Intrinsic	Convert variable to type UBYTE
UBYTEAT()	Intrinsic	Write UBYTE value into specified memory address
UBYTEAT()	Intrinsic	Read UBYTE value from specified memory address
UCASE$()	Intrinsic	Convert contents of string to upper case
ULONG	Statement	Declare variables to be type ULONG
ULONG()	Intrinsic	Convert variable to type ULONG
ULONGAT()	Intrinsic	Write ULONG value into specified memory address
ULONGAT()	Intrinsic	Read ULONG value from specified memory address
UNION	Statement	Declare/Define user-defined composite data type
UNTIL	Auxiliary	Used in DO and LOOP statements
USHORT	Statement	Declare variables to be type USHORT
USHORT()	Intrinsic	Convert variable to type USHORT
USHORTAT()	Intrinsic	Write USHORT value into specified memory address
USHORTAT()	Intrinsic	Read USHORT value from specified memory address
VERSION	Statement	Define version number of program
VERSION$()	Intrinsic	Return version number defined by VERSION statement
VOID	Statement	Declare that a function returns no value
WHILE	Auxiliary	Used in DO and LOOP statements
WRITE	Statement	Write variables to a file
XLONG	Statement	Declare variables to be type XLONG
XLONG()	Intrinsic	Convert variable to type XLONG
XLONGAT()	Intrinsic	Write XLONG value into memory at specified address
XLONGAT()	Intrinsic	Read XLONG value from memory at specified address
XMAKE()	Intrinsic	Retype to XLONG from any type (MSW if GIANT/DOUBLE)
XOR	Operator	Bitwise exclusive-OR operator

Appendix D : Operators

Operator	Description
~ NOT	BITWISE NOT
& AND	BITWISE AND
^ XOR	BITWISE XOR
| OR	BITWISE OR
!!	LOGICAL TEST
!	LOGICAL NOT
&&	LOGICAL AND
^^	LOGICAL XOR
||	LOGICAL OR
=	ASSIGNMENT
=	LOGICAL COMPARE FOR EQUAL
<>	LOGICAL COMPARE FOR NOT EQUAL
<	LOGICAL COMPARE FOR LESS THAN
<=	LOGICAL COMPARE FOR LESS THAN OR EQUAL
>	LOGICAL COMPARE FOR GREATER THAN
>=	LOGICAL COMPARE FOR GREATER THAN OR EQUAL
==	LOGICAL COMPARE FOR EQUAL
!=	LOGICAL COMPARE FOR NOT EQUAL
!<	LOGICAL COMPARE FOR NOT LESS THAN
!<=	LOGICAL COMPARE FOR NOT LESS THAN OR EQUAL
!>	LOGICAL COMPARE FOR NOT GREATER THAN
!>=	LOGICAL COMPARE FOR NOT GREATER THAN OR EQUAL
<<	BITWISE LEFT SHIFT (carry in zeros)
>>	BITWISE RIGHT SHIFT (carry in zeros)
<<<	ARITHMETIC UP SHIFT (carry in zeros)
>>>	ARITHMETIC DOWN SHIFT (carry in most significant bit)
+	CONCATENATE
+	ARITHMETIC ADD
-	ARITHMETIC SUBTRACT
*	ARITHMETIC MULTIPLY
\	ARITHMETIC INTEGER DIVIDE
/	ARITHMETIC FLOATING POINT DIVIDE
+	ARITHMETIC UNARY PLUS
-	ARITHMETIC UNARY MINUS
&	ADDRESS OF DATA
&&	ADDRESS OF HANDLE

OP	ALT	KIND	CLASS	OPERANDS	RETURNS	PREC	COMMENTS
&		unary	10	AnyType	Address	12	Address of Object Data
&&		unary	10	AnyType	Address	12	Address of Object Handle
!		unary	9	Numeric	T/F	12	Logical Not (TRUE if 0, else FALSE)
!!		unary	9	Numeric	T/F	12	Logical Test (FALSE if 0, else TRUE)
NOT	~	unary	9	Integer	SameType	12	Bitwise NOT
+		unary	8	Numeric	SameType	12	Plus
-		unary	8	Numeric	SameType	12	Minus
>>>		binary	7	Integer	LeftType	11	Arithmetic Up Shift
<<<		binary	7	Integer	LeftType	11	Arithmetic Down Shift
<<		binary	7	Integer	LeftType	11	Bitwise Left Shift
>>		binary	7	Integer	LeftType	11	Bitwise Right Shift
**		binary	4	Numeric	HighType	10	Power
/		binary	4	Numeric	HighType	9	Divide
*		binary	4	Numeric	HighType	9	Multiply
\\		binary	6	Numeric	Integer	9	Integer Divide
MOD		binary	6	Numeric	Integer	9	Modulus (Integer Remainder)
+		binary	5	Numeric	HighType	8	Add
+		binary	5	String	String	8	Concatenate
-		binary	4	Numeric	HighType	8	Subtract
AND	&	binary	3	Integer	HighType	7	Bitwise AND
XOR	^	binary	3	Integer	HighType	6	Bitwise XOR
OR	|	binary	3	Integer	HighType	6	Bitwise OR
>	!<=	binary	2	NumStr	T/F	5	Greater-Than
>=	!<	binary	2	NumStr	T/F	5	Greater-Or-Equal
<=	!>	binary	2	NumStr	T/F	5	Less-Or-Equal
<	!>=	binary	2	NumStr	T/F	5	Less-Than
<>	!=	binary	2	NumStr	T/F	4	Not-Equal
=	==	binary	2	NumStr	T/F	4	Equal (also "!<>")
&&		binary	1	Integer	T/F	3	Logical AND
^^		binary	1	Integer	T/F	2	Logical XOR
||		binary	1	Integer	T/F	2	Logical OR
=		binary		NumStr	RightType	1	Assignment
					T/F		T/F always returned as XLONG

Appendix E : Statements

ATTACH	Attach array node
AUTO	Declare AUTO scope
AUTOX	Declare AUTOX scope
DECLARE	Declare a function and it's parameters
DIM	Dimension an array
DO	Begin a DO...LOOP block
DOUBLE	Declare DOUBLE type (IEEE double-float)
END	End the program, a function, some block structure
EXIT	Exit a function or some block structure
EXPORT	Begin exporting type and constant declarations and function definitions
FOR	Begin a FOR...NEXT block
FUNCTION	Define a function and it's arguments
GOADDR	Declare a GOADDR type
GOSUB	Go to a subroutine
GOTO	Go to a line-label
IF	Begin IF...THEN...ELSE...ENDIF decision block
IMPORT	Import function library
INTERNAL	Declare function, scope = INTERNAL (local)
LIBRARY	Reserved
PRINT	Print to screen or file
PROGRAM	Begin and name program
READ	Read variables from diskfile
REDIM	Re-dimension an array; preserve overlapping data
RETURN	Return [value] from a function
SBYTE	Declare SBYTE type
SELECT	Begin SELECT_CASE block
SHARED	Declare SHARED scope
SINGLE	Declare SINGLE type
SLONG	Declare SLONG type
SSHORT	Declare SSHORT type
STATIC	Declare STATIC scope
STOP	Stop the program
STRING	Declare STRING type
SUB	Begin subroutine (SUB SubName...END SUB block)
SUBADDR	Declare SUBADDR type
TYPE	Declare and Define user-defined composite data type
UBYTE	Declare UBYTE type
ULONG	Declare ULONG type
UNION	Declare and Define user-defined composite data type
USHORT	Declare USHORT type
WRITE	Write variables to file
XLONG	Declare XLONG type

Appendix F : Intrinsics

ABS()	Absolute value
ASC()	Value of ASCII character
CHR$()	String with n characters of specified value
CJUST$()	Center justify string in field of spaces
CLR()	Clear a bit field
CSIZE()	Size of C string (to 1st null byte)
CSIZE$()	String up to null terminator
CSTRING$()	Clip string at 1st null byte
DHIGH()	High word of DOUBLE type
DLOW()	Low word of DOUBLE type
DMAKE()	Make DOUBLE type from two XLONGs
DOUBLE()	Convert to DOUBLE type
DOUBLEAT()	Direct memory access
ERROR()	Return and/or set error number
ERROR$()	Convert error number into error string
EXTS()	Extract signed bit field
EXTU()	Extract unsigned bit field
FIX()	Fix rouding
FUNCADDR()	Convert to FUNCADDR type
FUNCADDRAT()	Direct memory access
FUNCADDRESS()	Get address of function (type FUNCADDR)
GHIGH()	Get high word of GIANT value
GIANT()	Convert to GIANT type
GIANTAT()	Direct memory access
GLOW()	Get low word of GIANT value
GMAKE()	Make GIANT from two XLONGs
GOADDR()	Convert to GOADDR type
GOADDRAT()	Direct memory access
GOADDRESS()	Get address of GOTO label (type GOADDR)
HEX$()	n digit hexadecimal string
HEXX$()	n digit hexadecimal string with "0x" prefix
HIGH0()	Bit # of highest 0 bit
HIGH1()	Bit # of highest 1 bit
INCHR()	1st character in string to match one in list
INCHRI()	Case insensitive INCHR()
INSTR()	Find string within string
INSTRI()	Case insensitive INSTR()
INT()	Integerize
LCASE$()	Lower case a string
LCLIP$()	Clip left n bytes from string
LEFT$()	Left n characters of a string
LEN()	Length of a string in elements
LIBRARY()	Return TRUE if compiled as library
LJUST$()	Left justify string in field of spaces
LTRIM$()	Trim spaces and tabs from left end of string
MAKE()	Make an arbitrary bit field
MID$()	Extract arbitrary substring from a string
NULL$()	Return string of null characters
PROGRAM$()	Return program name defined by PROGRAM statement
RCLIP$()	Clip right n bytes from string
RIGHT$()	Right n characters of a string
RINCHR()	last character in string to match one in list
RINCHRI()	Case insensitive RINCHR()
RINSTR()	Reverse direction INSTR()
RINSTRI()	Case insensitive RINSTR()
RJUST$()	Right justify string in field of spaces
RND()	Rounding
ROTATER()	n bit rotate right

ROUND()	**Rounding**
RTRIM$()	**Trim spaces and tabs from right end of string**

SBYTE()	Convert to SBYTE type
SBYTEAT()	Direct memory access
SET()	Set a bit field
SIGN()	Sign (-1 or +1)
SIGNED$()	Convert to STRING type (leading "-" or "+")
SINGLE()	Convert to SINGLE type
SINGLEAT()	Direct memory access
SIZE()	Size of variable
SGN()	Sign (-1, 0, +1)
SLONG()	Convert to SLONG type
SLONGAT()	Direct memory access
SMAKE()	Make a SINGLE type from an XLONG (not a convert)
SPACE$()	String of n space characters
SQR()	Square root
SSHORT()	Convert to SSHORT type
SSHORTAT()	Direct memory access
STR$()	Convert to STRING type (leading "-" or " ")
STRING()	Convert to STRING type (leading "-" or "")
STRING$()	Convert to STRING type (leading "-" or "")
STUFF$()	Stuff one string into another
SUBADDR()	Convert to SUBADDR type
SUBADDRAT()	Direct memory access
SUBADDRESS()	Get address of subroutine
TAB()	Tab to column n in PRINT statements
TRIM$()	Trim tabs and spaces from both ends of string
TRUNC()	Truncate
TYPE()	Type of variable, array, component
UBOUND()	Upper bound of array (at any node)
UBYTE()	Convert to UBYTE type
UBYTEAT()	Direct memory access
UCASE$()	Upper case a string
ULONG()	Convert to ULONG type
ULONGAT()	Direct memory access
USHORT()	Convert to USHORT type
USHORTAT()	Direct memory access
VERSION$()	Return version number defined by VERSION statement
XLONG()	Convert to XLONG type
XLONGAT()	Direct memory access
XMAKE()	Retype to XLONG from any type (not a convert)

Appendix G : Language Reference

KEYWORD ***intrinsic*** ***statement*** ***operator***	***returnValues*** and ***arguments*** are **XLONG** unless otherwise stated. **SLONG** and **ULONG** accepted for **XLONG** with no overhead. ***integer*** **-** any integer type except **GIANT** ***integers*** **-** any integer type including **GIANT** ***numeric*** **-** ***integers*** or ***floating point*** ***float*** **-** **SINGLE** or **DOUBLE** ***single*** **-** **SINGLE** ***double*** **-** **DOUBLE** ***string*** **-** character string (string of unsigned bytes) ***numString*** **-** numeric or string ***expression*** **-** numeric or string ***bitspec*** **-** bitfield specification returned by **BITFIELD()**

ABS()

intrinsic

***numeric* = ABS(*numeric*)**

Return the absolute value of any simple numeric type. The return type is the same as the argument.

```
a = +23
b = -23
c = ABS(a)    ' c = 23
d = ABS(b)    ' d = 23
```

ALL

See **SELECT CASE**.

AND

binary bitwise operator

integers* = *integers* AND *integers

Bitwise **AND** two integer operands to produce an integer result.

```
a = 0x05F0
b = 0xFFA0
c = a AND b   ' c = 0x05A0
```

ASC()

intrinsic

***integer* = ASC(*string*)**
***integer* = ASC(*string, position*)**

Return one byte from a string. The first argument is the string to extract the byte from. The second argument is the position of the byte to extract, or 1 if no second argument is given.

```
a$ = "abcde"
b = ASC(a$)      ' b = 'a'
c = ASC(a$,4)    ' c = 'd'
d = ASC(a$,6)    ' d = -1    ( specified byte not within string )
```

ATTACH

statement

ATTACH *arrayNode* TO *arrayNode*

Move an array from source node to destination node. If the destination node is not empty a runtime error occurs. After the array is moved to the destination node, the source node is zeroed.

```
ATTACH a[] TO b[]              ' attach array to array
ATTACH a[] TO b[i,]            ' attach array to node
ATTACH a[i,] TO b[]            ' attach node to array
ATTACH a[i,] TO b[j,k,]        ' attach node to node
ATTACH a$ TO b$[i]             ' attach string to node
ATTACH b$[i] TO a$             ' attach node to string
```

AUTO
AUTOX

statement

AUTO [*typename*] *variables*
AUTOX [*typename*] *variables*

Declare variables with **AUTO** and **AUTOX** scope.

```
AUTO  a, b$, c[]
AUTO SINGLE  x, y, z
AUTO MYTYPE  first[], middle[], last[]
AUTOX  i, j$, k[]
AUTOX DOUBLE  p[], q[], r[], s[], t[]
AUTOX TYPERS  fast[], accurate[], both[]
```

BIN$()
BINB$()

intrinsic

***string* = BIN$(*integers*)**
***string* = BINB$(*integers*)**
***string* = BIN$(*integers, digits*)**
***string* = BINB$(*integers, digits*)**

Return the binary string representation of an integer. The return string has as many characters as necessary to represent the integer, or the number specified by the second argument, whichever is larger.

```
a = 0xF0F5
a$ = BIN$(a)        ' a$ = "1111000011110101"
a$ = BINB$(a)       ' a$ = "0b1111000011110101"
a$ = BIN$(a,20)     ' a$ = "00001111000011110101"
a$ = BINB$(a,20)    ' a$ = "0b00001111000011110101"
a$ = BIN$(a,5)      ' a$ = "1111000011110101"
a$ = BINB(a,5)      ' a$ = "0b1111000011110101"
```

BITFIELD()

intrinsic

bitspec = `BITFIELD(`***width, offset***`)`

Return a bitfield specification appropriate for bitfield extract and bitfield intrinsics. The first argument is the width of the bitfield. The second argument is the offset of the bitfield from the least significant bit. Valid widths are 1 to 31, and valid offsets are 0 to 31.

Works in prolog given two integer constants, and elsewhere in programs with integer values.

```
$$BYTE0 = BITFIELD(8,  0)   ' bits 00-07  : low byte
$$BYTE1 = BITFIELD(8,  8)   ' bits 08-15  : next higher
$$BYTE2 = BITFIELD(8, 16)   ' bits 16-23  : next higher
$$BYTE3 = BITFIELD(8, 24)   ' bits 24-31  : high byte
$$KINDS = BITFIELD(5, 21)   ' bits 21-25  : 5 wide bitfield at bit 21
spacing = BITFIELD(3, 29)   ' bits 29-31  : 3 wide bitfield at bit 29
```

CASE

See **`SELECT CASE`**.

CFUNCTION

See **`FUNCTION`**.

CHR$()

intrinsic

string = `CHR$(`***integer***`)`
string = `CHR$(`***integer, count***`)`

Return a string of 1 or more copies of a character. The first argument is the value of the character. The second argument is the number of copies of the character, which defaults to 1 when not given.

```
a$ = CHR$(32)          ' a$ = " "           ( 1 space character )
a$ = CHR$(32, 8)       ' a$ = "        "    ( 8 space characters )
a$ = CHR$('x', 8)      ' a$ = "xxxxxxxx"    ( 8  'x'  characters )
```

CJUST$()
LJUST$()
RJUST$()

intrinsic

string = `CJUST$(`***string, length***`)`
string = `LJUST$(`***string, length***`)`
string = `RJUST$(`***string, length***`)`

Return a string center-justified, left-justified, or right-justified in a field of space characters. The first argument is the string to justify. The second argument is the field width. The return string is always the specified width. If a string cannot be exactly centered, the extra space follows the string. If the string is longer than the field, the string is left justified and clipped at the end of the field.

```
a$ = "cat"
b$ = "catamaran"
a$ = CJUST$(a$,7)      ' a$ = "  cat  "    ( exact centering )
a$ = CJUST$(a$,8)      ' a$ = "  cat   "   ( excess space on right )
a$ = CJUST$("xxx",9)   ' a$ = "   xxx   "  ( exact centering )
a$ = CJUST$(b$,3)      ' a$ = "cat"        ( no fit, left justify )
a$ = LJUST$(a$,6)      ' a$ = "cat   "     ( left justify cat in 6 )
a$ = LJUST$(b$,7)      ' a$ = "catamar"    ( string longer than 7 )
a$ = LJUST$("xxx",8)   ' a$ = "xxx     "   ( left justify xxx in 8 )
```

```
a$ = LJUST$(b$,3)       ' a$ = "cat"        ( string longer than 3 )
a$ = RJUST$(a$,6)       ' a$ = "   cat"     ( right justify in 6 )
a$ = RJUST$(a$,7)       ' a$ = "tamaran"    ( clip leading to fit )
a$ = RJUST$("xxx",8)    ' a$ = "     xxx"   ( right justify in 8 )
a$ = RJUST$(b$,3)       ' a$ = "ran"        ( clip leading to fit )
```

CLOSE()

intrinsic

integer = `CLOSE(`*fileNumber*`)`

Close an open file and return 0 to indicate success. The argument is the filenumber assigned to the file when it was opened. If the argument is not the filenumber of an open file, `-1` is returned.

```
err = CLOSE (ifile)      ' test for close error
CLOSE (ofile)            ' ... or don't bother
```

CLR()
SET()

intrinsic

integer = `CLR(`*integer, bitspec*`)`
integer = `CLR(`*integer, width, offset*`)`
integer = `CLR(`*integer, bitspec*`)`
integer = `CLR(`*integer, width, offset*`)`
integer = `SET(`*integer, bitspec*`)`
integer = `SET(`*integer, width, offset*`)`
integer = `SET(`*integer, bitspec*`)`
integer = `SET(`*integer, width, offset*`)`

Clear or set a field of bits. The first argument is the integer value in which the bits are cleared or set. In the two argument version, the second argument is a bitspec that contains the width and offset of the bitfield to clear or set. In the three argument version the second and third arguments are the width and offset of the bitfield to clear or set.

```
$$TYPE = BITFIELD(5,16)      ' in PROLOG
$KIND = BITFIELD(5,24)       ' in a function
kind = BITFIELD(5,24)        ' variable as bitspec
i = 0xFFFFFFFF               ' i = 0xFFFFFFFF
j = 0x00000000               ' j = 0x00000000
a = CLR(i, $$TYPE)           ' a = 0xFFE0FFFF
a = CLR(i, $KIND)            ' a = 0xE0FFFFFF
a = CLR(i, kind)             ' a = 0xE0FFFFFF
a = CLR(i, 8, 4)             ' a = 0xFFFFF00F
a = SET(j, $$TYPE)           ' a = 0x001F0000
a = SET(j, $KIND)            ' a = 0x1F000000
a = SET(j, kind)             ' a = 0x1F000000
a = SET(j, 8, 4)             ' a = 0x00000FF0
```

CSIZE()

intrinsic

integer = `CSIZE(`*string*`)`

Return the number of bytes in a string before the first zero byte.

```
a$ = ""
b$ = "abcdefg"
c$ = "abc\0defg"
a = CSIZE(a$)       ' a = 0
a = CSIZE(b$)       ' a = 7
a = CSIZE(c$)       ' a = 3
```

CSIZE$()

intrinsic

string = `CSIZE$(`*string*`)`

Return a copy of a zero terminated string. The end of the string is the byte before the first zero byte.

```
a$ = ""
b$ = "abcdefg"
c$ = "abc\0defg"
s$ = CSIZE$(a$)     ' s$ = ""
s$ = CSIZE$(b$)     ' s$ = "abcdefg"
s$ = CSIZE$(c$)     ' s$ = "abc"
```

CSTRING$()

intrinsic

string `= CSTRING$(`*address*`)`

Return a copy of a string at a memory address. The end of the string is the byte before the first zero byte. This intrinsic converts C style strings into native strings. The string at the specified address is not altered or freed.

```
cstr = cfunc()
a$ = CSTRING$(cstr)
```

DCOMPLEX

statement

[*scope*] `DCOMPLEX` *variables*

Declare double precision complex variables.
Also see ***Declarations*** after the alphabetical listing.

```
DCOMPLEX  ii, jj, kk
DCOMPLEX  x[], y[], z[]
SHARED DCOMPLEX  a, b, c, a[], b[], c[]
```

DECLARE FUNCTION
EXTERNAL FUNCTION
INTERNAL FUNCTION

statement

`DECLARE FUNCTION` [*type*] *Func*`(`[*parameters*]`)`
`EXTERNAL FUNCTION` [*type*] *Func*`(`[*parameters*]`)`
`INTERNAL FUNCTION` [*type*] *Func*`(`[*parameters*]`)`

Declare a function:

`DECLARE` - ***function in current program, visible to all programs.***
`INTERNAL` - ***function in current program, invisible to other programs.***
`EXTERNAL` - ***function not in current program, visible to all programs.***

The optional return ***type*** is any built-in or user-defined data type.

parameters is an optional comma-separated list of up to 16 built-in or user-defined type names or symbols from which the kind and type of each function argument can be determined.

```
DECLARE  FUNCTION  Twist ()
INTERNAL FUNCTION  DOUBLE  DarkSine ( a#, b#, c#[] )
EXTERNAL FUNCTION  DCOMPLEX  DoodleDuxis ( DCOMPLEX dd, DCOMPLEX ee )
```

DHIGH()
DLOW()

intrinsic

integer `= DHIGH(`*double*`)`
integer `= DLOW(`*double*`)`

Return the high or low 32-bits of a double precision floating point number. The high 32-bits of a double precision floating point number contains the sign bit, exponent, and high part of the mantissa, while the low 32-bits contains the low part of the mantissa.

```
a# = 0d40010802DEADCODE      ' a# = double variable
b# = 0d4000800080000000      ' b# = double variable
i = DHIGH(a#)                ' i = 0x40010802
j = DHIGH(b#)                ' j = 0x40008000
i = DLOW(a#)                 ' i = 0xDEADCODE
j = DLOW(b#)                 ' j = 0x80000000
```

DIM
REDIM

statement

`DIM` *array*`[`*subscripts*`]`
`REDIM` *array*`[`*subscripts*`]`

Dimension or redimension an array. 0 to 8 comma-separated integer subscripts give the upper bounds for 0 to 8 dimensions. The lower bound of all arrays is 0.

When an array is dimensioned, the existing contents are first freed, then memory space for the array is allocated and filled with zeros. When an array is redimensioned, the existing contents not in the new size are lost, the existing contents in both the old and new size are unchanged, and contents in the new size only are zeroed.

```
DIM a[]                 ' a[] becomes an empty array
DIM a#[upper]           ' dimension a#[], upper bound = u
DIM points[100000]      ' dimension points[], upper bound = 100000
DIM a$[tops]            ' dimension string array, upper bound = tops
DIM a[i,j,k]            ' dimension a three dimensional array
DIM a[i,j,]             ' dimension a two dimensional array of nodes
```

DLOW()

See `DHIGH()`.

DMAKE()

intrinsic

double `= DMAKE(`***high32, low32***`)`

Return a 64-bit double precision floating point number assembled from two 32-bit integer parts.

```
hi = 0x40008000              ' high 32-bits of desired DOUBLE
lo = 0x80000000              ' low 32-bits of desired DOUBLE
i# = DMAKE(hi, lo)           ' i# = 0d4000800080000000
i# = DMAKE(si OR uman, lman) ' i# = assemble sign, hi/lo mantissa
```

DO
DO WHILE
DO UNTIL
DO DO
DO LOOP
EXIT DO
LOOP
LOOP WHILE
LOOP UNTIL

statement

`DO`
`DO WHILE` ***numString***
`DO UNTIL` ***numString***
`DO DO` **[*level*]**
`DO LOOP` **[*level*]**
`EXIT DO` **[*level*]**
`LOOP`
`LOOP WHILE` ***numString***
`LOOP UNTIL` ***numString***

`DO...LOOP` blocks.

`DO` begins a loop, and execution continues on the next line.

`DO WHILE` continues execution on the next line if the numeric or string value is `TRUE`, otherwise execution continues after the matching `LOOP` statement.

`DO UNTIL` continues execution on the next line if the numeric or string value is `FALSE`, otherwise execution continues after the matching `LOOP` statement.

`DO DO` jumps directly to the `DO` statement at the beginning of a loop block from anywhere inside the block.

`DO LOOP` jumps directly to the `LOOP` statement at the end of a loop block from anywhere inside the block.

`EXIT DO` jumps directly past the `LOOP` statement at the end of the loop block, from anywhere inside the block.

`LOOP` jumps to the matching `DO` statement.

`LOOP WHILE` jumps to the matching `DO` statement if the numeric or string expression is `TRUE`.

`LOOP UNTIL` jumps to the matching `DO` if the numeric or string expression is `FALSE`.

```
hash = 0
IF a$ THEN
  o = 0                  ' offset = 0
  u = UBOUND(a$)         ' u = upper offset in a$
  DO WHILE (o <= u)      ' do hash loop while offset <= upper offset
    hash = hash + a${o}  ' add next byte to hash
    INC o                ' offset to next byte in a$
  LOOP
END IF
```

DOUBLE

See ***Declarations*** after the alphabetical listing.

DOUBLE()

See ***Type Conversions*** after the alphabetical listing.

DOUBLEAT()

See ***Direct Memory Access*** after the alphabetical listing.

ELSE

See `IF`.

END FUNCTION

statement

`END FUNCTION` **[*expression*]**

End a function and return a value. If no return expression is given, a zero or empty string is returned.

END IF

See `IF`.

END SELECT

See `SELECT CASE`.

END SUB

See `SUB`.

END TYPE

See `TYPE`.

EOF()

intrinsic

integer `= EOF(`***fileNumber***`)`

Return `TRUE` if filepointer points beyond the end of a file.

```
a = EOF(ifile)
DO UNTIL EOF(ifile)
```

ERROR()

intrinsic

error `= ERROR(`***newError***`)`

Return `##ERROR` error number and assign a new error number unless `newError = -1`.

```
IF ERROR(-1) THEN   ' test ##ERROR without clearing it
error = ERROR(0)    ' get current ##ERROR, then clear it
error = ERROR(-1)   ' get current ##ERROR without changing it
error = ERROR(new)  ' get current ##ERROR and assign a new value
```

ERROR$()

intrinsic

error$ `= ERROR$(`***error***`)`

Convert an error number into an error string.

```
error = ERROR(0)          ' get error number
error$ = ERROR$(error)    ' error$ = error string
```

EXIT DO

statement

`EXIT DO` **[*level*]**

Exit a `DO` loop.

Jump past the ***n*** n^{th} `LOOP` statement, where ***n=1*** or ***level***.

```
DO
  a$ = INLINE$ ("Enter another choice ===>> ")
  IFZ a$ THEN EXIT DO
  RegisterEntry (a$)
LOOP
```

EXIT FOR

statement

EXIT FOR [*level*]

Exit a **FOR** loop.

Jump past the ***n***n^{th} **NEXT** statement, where ***n=1*** or ***level***.

```
FOR i = 0 TO 1000
  a$ = INLINE$ ("Enter another choice ===>> ")
  IFZ a$ THEN EXIT FOR
  RegisterEntry (i, a$)
NEXT i
```

EXIT FUNCTION

See **RETURN**.

EXIT IF

statement

EXIT IF [*level*]

Exit an **IF** block.

Jump past the n^{th} **END IF** statement, where ***n=1*** or ***level***.

```
IF enabled THEN
  IF (value < max) THEN
    IF verify[entry] THEN
      a = verify[entry]
      IF (a < 0) THEN EXIT IF 4
      verify[entry] = a+1
    END IF
  END IF
END IF
```

EXIT SELECT

statement

EXIT SELECT [*level*]

Exit a **SELECT CASE** block.

Jump past the n^{th} **END SELECT** statement, where ***n=1*** or ***level***.

```
SELECT CASE a
  CASE 1: GOSUB FireAway
  CASE 2: GOSUB BreakOut
  CASE 3: k = Prefix(n)
          IFZ k THEN EXIT SELECT
          GOSUB PassOff
END SELECT
```

EXIT SUB

See **SUB**.

EXPORT

EXPORT

Begin exporting type and shared constant definitions and function declarations from a program being compiled as a function library. All source lines between **EXPORT** and **END EXPORT** statements are put in "***filename*.dec**".

```
EXPORT
  TYPE LINK
    XLONG  .backward
    XLONG  .forward
  END TYPE
'
  DECLARE FUNCTION DOUBLE  Square (x#)
  DECLARE FUNCTION DOUBLE  Cube (x#)
END EXPORT
'
```

```
EXPORT
  $$Off = 0
  $$On = -1
END EXPORT
```

EXTERNAL

See ***Declarations*** after the alphabetical listing.

EXTERNAL FUNCTION

See **`DECLARE FUNCTION`**.

EXTS()
EXTU()

intrinsic

integer **`= EXTS(`*integer, bitspec*`)`**
integer **`= EXTS(`*integer, width, offset*`)`**
integer **`= EXTU(`*integer, bitspec*`)`**
integer **`= EXTU(`*integer, width, offset*`)`**

Extract a signed or unsigned field of bits. The first argument is the integer value from which the bits are extracted. In the two argument version, the second argument is a bitspec that contains the width and offset of the bitfield to extract. In the three argument version the second and third arguments are the width and offset of the bitfield to extract.

The extracted bitfield is returned in the low bits of the result, either sign-extended or zero-extended to fill the 32-bit integer result.

```
$$TYPE = BITFIELD(4,16)       ' in PROLOG
$KIND = BITFIELD(4,20)        ' in a function
spaz = BITFIELD(8,24)         ' variable as bitspec
i = 0x89ABCDEF                ' i = 0x89ABCDEF
a = SET(i, $$TYPE)            ' a = 0x0000000B
a = SET(i, $KIND)             ' a = 0x0000000A
a = SET(i, spaz)              ' a = 0x00000089
a = SET(i, 8, 4)              ' a = 0x000000DE
```

FALSE

See **`SELECT CASE`**.

FIX()

See **`INT()`**.

FOR
DO FOR
DO NEXT
EXIT FOR
NEXT

statement

`FOR` ***var* = *numeric*** **`TO`** ***numeric*** **[`STEP`** ***numeric*]**
`DO FOR` **[*level*]**
`DO NEXT` **[*level*]**
`EXIT FOR` **[*level*]**
`NEXT` **[*var*]**

Begin and end a **`FOR`**...**`NEXT`** loop.

`DO FOR` jumps directly to the **`FOR`** statement.

`DO NEXT` jumps directly to the **`NEXT`** statement.

`EXIT FOR` jumps directly past the **`NEXT`** statement.

```
FOR i = 0 TO last
  i$ = item$[i]
  IFZ i$ THEN DO NEXT
  IF i$ = "outta here" THEN EXIT FOR
  IF (i$ = "trash") THEN i = i + 3 : DO FOR
  Register (i$)
NEXT i
```

FORMAT$()

intrinsic

string **`= FORMAT$(`*format$, argument*`)`**

Return a string formatted per a format spec string and argument.

`FORMAT$()` creates and returns a string representation of up a numeric or string ***argument***, formatted according to a ***format$*** string.

The following are valid format fields for string arguments:

```
  &  Print string exactly, no upper or lower length limit
  <  Left justify string in <<<<< field  (1 to 255 "<" characters)
  >  Right justify string in >>>>> field  (1 to 255 ">" characters)
  |  Center justify string in ||||| field  (1 to 255 "|" characters)
```

The following are valid character sequences for numeric arguments:

```
###     digit positions               ######         54321
.       decimal point                 ###.##         23.00
,       commas every 3 places         ##,###.##     27,654,321.80
^^^^    short exponent                #.###^^^^     3.711e+22
^^^^^   long exponent                 #.##^^^^^     3.71d+022
(###)   negative #s in parens         (#####)       (54321)
$       leading $ symbol              $###.##         $3.76
*       * in leading zeros            *####.##      ****3.76
+       print leading +/- sign        +####.##      +234.56
+       print trailing +/- sign       ####.##+       234.56+
-       print trailing - if negative  ####.##-       234.56-
_       print next character exactly  a_$_#_._-_+   a$#.-+

x = 23
y$$ = 23456
z# = 11111.222
j$ = "scam"
fa$ = " ###"
fb$ = "###,###"
fc$ = "###.####^^^^^"
fd$ = "||||||"
PRINT FORMAT$ (fa$,x)                      ' "  23"
PRINT FORMAT$ (fb$,y$$)                    ' " 23,456"
PRINT FORMAT$ (fc$,z#)                     ' " 111.1122d+002
PRINT FORMAT$ (fd$,j$)                     ' " scam "
a$ = FORMAT$ (fa$,x) + FORMAT$ (fb$,y$$)  ' a$ = "  23 23,456"
```

FUNCADDR

statement

[*scope*] `FUNCADDR` **[*type*]** ***variable*`(`[*parameters*]`)`**
[*scope*] `FUNCADDR` **[*type*]** ***array*`[]``(`[*parameters*]`)`**

Declare a `FUNCADDR` variable or array.

```
STATIC FUNCADDR a ( STRING, XLONG, XLONG, XLONG )
SHARED FUNCADDR DOUBLE  Manglex[] ( DOUBLE, DOUBLE )
```

FUNCADDR()

See ***Type Conversion*** after the alphabetical listing.

FUNCADDRESS()

intrinsic

funcaddr `=` `FUNCADDRESS(`***FuncName***`())`

Return the address of a function. Same as `&Func()`.

```
process[GetColor] = FUNCADDRESS (XuiGetColor())
process[SetColor] = FUNCADDRESS (XuiSetColor())
```

FUNCTION
SFUNCTION
CFUNCTION

statement

`FUNCTION` **[*type*]** ***FuncName*`(`*arguments*`)`** **[*defaultType*]**
`SFUNCTION` **[*type*]** ***FuncName*`(`*arguments*`)`** **[*defaultType*]**
`CFUNCTION` **[*type*]** ***FuncName*`(`*arguments*`)`** **[*defaultType*]**

Begin the body of a function. The return type defaults to `XLONG` unless specified otherwise. The ***arguments*** are a comma-separated list of variables receiving input values from calling functions. The type of variables in the function not declared otherwise defaults to `XLONG` unless a ***defaultType*** is specified.

`FUNCTION` denotes a normal function.

`SFUNCTION` denotes operating system functions with standard operating system function protocol, or native functions to be called by the operating system expecting standard operating system protocol **(STDCALL)**.

CFUNCTION denotes C functions or native functions to be called by functions expecting standard C function protocol **(CDECL)**.

```
FUNCTION  Blivit ( a, b, c$, d[] )
FUNCTION  Bondar ( i#, j#, k#, l#, m# )
SFUNCTION DOUBLE  DoLittle ( DOUBLE lo, DOUBLE hi )  DOUBLE
CFUNCTION DOUBLE  ArcCosh ( DOUBLE arg )
```

GHIGH()
GLOW()

intrinsic

integer = **GHIGH(*giant*)**

Return the high or low 32-bits of a 64-bit signed integer.

```
a$$ = 0x40010802DEADCODE      ' a$$ = giant variable
b$$ = 0x4000800080000000      ' b$$ = giant variable
i = GHIGH(a$$)                ' i = 0x40010802
j = GHIGH(b$$)                ' j = 0x40008000
i = GLOW(a$$)                 ' i = 0xDEADCODE
j = GLOW(b$$)                 ' j = 0x80000000
```

GIANT — See ***Declarations*** after the alphabetical listing.

GIANT() — See ***Type Conversions*** after the alphabetical listing.

GIANTAT() — See ***Direct Memory Access*** after the alphabetical listing.

GLOW() — See **GHIGH()**.

GMAKE()

intrinsic

giant = **GMAKE(*high32, low32*)**

Return a 64-bit signed integer assembled from two 32-bit integers.

```
hi = 0x40008000                ' hi 32-bits of desired GIANT
lo = 0x80000000                ' lo 32-bits of desired GIANT
i$$ = GMAKE(hi, lo)            ' i$$ = 0x4000800080000000
i$$ = GMAKE(upper, lower)      ' i$$ = make GIANT from two XLONGs
```

GOADDR — See ***Declarations*** after the alphabetical listing.

GOADDR() — See ***Type Conversions*** after the alphabetical listing.

GOADDRAT() — See ***Direct Memory Access*** after the alphabetical listing.

GOADDRESS()

intrinsic

goaddr = **GOADDRESS(*label*)**

Return the address of a **GOTO** label.

```
g = GOADDRESS (label)
h[i] = GOADDRESS (another)
```

GOSUB

statement

GOSUB ***SubName***
GOSUB @*subVar*
GOSUB @*subArray*[*indices*]

Call a subroutine directly, through a variable, or through an array.

```
GOSUB subroutine
GOSUB @subVar
GOSUB @subArray[i]
```

GOTO

statement

GOTO ***label***

Jump to a label directly, through a variable, or through an array.

```
GOTO label
GOTO @goVar
GOTO @goArray[i]
```

HEX$()
HEXX$()

intrinsic

string = `HEX$(`*integers*`)`
string = `HEXX$(`*integers*`)`
string = `HEX$(`*integers, digits*`)`
string = `HEXX$(`*integers, digits*`)`

Return the hexadecimal string representation of an integer. The return string has as many characters as necessary to represent the integer in hexadecimal, or the number specified by the second argument, whichever is larger.

```
a = 0xDEADC0DE
a$ = HEX$(a)         ' a$ = "DEADC0DE"
a$ = HEXX$(a)        ' a$ = "0xDEADC0DE"
a$ = HEX$(a,2)       ' a$ = "DEADC0DE"
a$ = HEXX$(a,2)      ' a$ = "0xDEADC0DE"
a$ = HEX$(a,12)      ' a$ = "0000DEADC0DE"
a$ = HEXX$(a,12)      ' a$ = "0x0000DEADC0DE"
```

HIGH0()
HIGH1()

intrinsic

integer = `HIGH0(`*integer*`)`
integer = `HIGH1(`*integer*`)`

Return the bit number of the most significant 0 or 1 bit.

```
a = 0x00C03333
b = 0xFFFE0000
c = HIGH0(a)       ' c = 31
d = HIGH0(b)       ' d = 16
e = HIGH1(a)       ' e = 23
f = HIGH1(b)       ' f = 31
```

IF
IFZ
ELSE
END IF

statement

`IF` *expression* `THEN` *statements*
`IF` *expression* `THEN` *statements* `ELSE` *statements*
`IF` *expression* `THEN`
  statements
`END IF`
`IF` *expression* `THEN`
  statements
`ELSE`
  statements
`END IF`

Execute statements conditionally.

```
IF a THEN PRINT "a is non-zero"
IFZ a THEN PRINT "a is zero"
IF a$ THEN PRINT "a$ has contents"
IFZ a$ THEN PRINT "a$ is empty"
IF a[] THEN PRINT "a[] has contents"
IFZ a[] THEN PRINT "a[] is empty"
IF (humidity > 80) THEN PRINT "Close The Roof"
IF a THEN
  IFZ b THEN
    PRINT "a is true, b is not"
  ELSE
    PRINT "a is true, b is true"
  END IF
END IF
```

IMPORT

`IMPORT "`*libname*`"`

Import function library ***libname*** to make its exported types, constants and functions visible to this program. `"`***libname***`.dec"` is read and compiled, and if it exists, `"`***libname***`.dll"` is loaded, linked to the program, and its entry function is called to initialize the library.

```
IMPORT "xma"
IMPORT "xst"
```

INC
DEC

statement

`INC` ***variable***
`DEC` ***variable***

Add one to a variable or subtract one from a variable.

```
INC a
INC a[i]
INC x#
INC parent.kids
INC parent[grid].kids
DEC a
DEC a[i]
DEC x#
DEC parent.kids
DEC parent[grid].kids
```

INCHR()
INCHRI()
RINCHR()
RINCHRI()

intrinsic

***integer* = `INCHR(`*searchMe$, searchFor$*`)`**
***integer* = `INCHR(`*searchMe$, searchFor$, start*`)`**
***integer* = `INCHRI(`*searchMe$, searchFor$*`)`**
***integer* = `INCHRI(`*searchMe$, searchFor$, start*`)`**
***integer* = `RINCHR(`*searchMe$, searchFor$*`)`**
***integer* = `RINCHR(`*searchMe$, searchFor$, start*`)`**
***integer* = `RINCHRI(`*searchMe$, searchFor$*`)`**
***integer* = `RINCHRI(`*searchMe$, searchFor$, start*`)`**

Search a string for any of the characters in another string. Return the position of the first match.

`INCHR()`	= ***forward search, case sensitive***
`INCHRI()`	= ***forward search, case insensitive***
`RINCHR()`	= ***reverse search, case sensitive***
`RINCHRI()`	= ***reverse search, case insensitive***

```
a$ = "Help me please!"
b$ = "ABCDEFG"
c$ = "mromjtp"
a = INCHR(a$,b$)        ' a = 0    "e" != "E"
a = INCHR(a$,c$)        ' a = 4    "p"
a = INCHR(a$,c$,10)     ' a = 0    past matches
a = INCHRI(a$,b$)       ' a = 2    "e" = "E"
a = INCHRI(a$,c$)       ' a = 4    "p"
a = INCHRI(a$,c$,5)     ' a = 6    "m"
a = RINCHR(a$,b$)       ' a = 0    "e" != "e"
a = RINCHR(a$,c$)       ' a = 9    "p"
a = RINCHR(a$,c$,7)     ' a = 6    "m"
a = RINCHRI(a$,b$)      ' a = 14   "e" = "E"
a = RINCHRI(a$,c$)      ' a = 9    "p"
a = RINCHRI(a$,c$,3)    ' a = 0    before matches
```

INFILE$()

intrinsic

***string* = `INFILE$(`*fileNumber*`)`**

Return the next line from an open file.

```
FUNCTION  FindStringInFile (test$, fileName$)
  line = 0
  ERROR (0)
  found = $$FALSE
  ifile = OPEN (fileName$, $$RD)
  IF ERROR (-1) THEN RETURN (-1)
  DO UNTIL EOF(ifile)
    line$ = INFILE$ (ifile)
    check = INSTR (line$, test$)
    IF check THEN
```

```
      found = $$TRUE
      PRINT "Found ===>> "; line$
      EXIT DO
    END IF
    INC line
  LOOP
  CLOSE (ifile)
END FUNCTION
```

`INLINE$()`

intrinsic

string = `INLINE$`(*prompt$*)

Return a line from the standard input device (the keyboard).

```
a$ = INLINE$ ("Enter your name here ===>> ")
IF (a$ = "Bill Gates") THEN PRINT "Yeah, tell me another one."
```

`INSTR()`
`INSTRI()`
`RINSTR()`
`RINSTRI()`

intrinsic

integer = `INSTR`(*searchMe$, searchFor$*)
integer = `INSTR`(*searchMe$, searchFor$, start*)
integer = `INSTRI`(*searchMe$, searchFor$*)
integer = `INSTRI`(*searchMe$, searchFor$, start*)
integer = `RINSTR`(*searchMe$, searchFor$*)
integer = `RINSTR`(*searchMe$, searchFor$, start*)
integer = `RINSTRI`(*searchMe$, searchFor$*)
integer = `RINSTRI`(*searchMe$, searchFor$, start*)

Search a string for another string. Return the position of the match.

`INSTR()`	= *forward search, case sensitive*
`INSTRI()`	= *forward search, case insensitive*
`RINSTR()`	= *reverse search, case sensitive*
`RINSTRI()`	= *reverse search, case insensitive*

```
a$ = "HEALTHY, wealthy, and wise!"
b$ = "He"
c$ = "alt"
a = INSTR(a$,b$)        ' a =  0    no match
a = INSTR(a$,c$)        ' a = 12    "alt"
a = INSTR(a$,c$,13)     ' a =  0    past match
a = INSTRI(a$,b$)       ' a =  1    "HE" = "He"
a = INSTRI(a$,c$)       ' a =  4    "ALT" = "alt"
a = INSTRI(a$,c$,5)     ' a = 12    "alt" = "alt"
a = RINSTR(a$,b$)       ' a =  0    no match
a = RINSTR(a$,c$)       ' a = 12    "alt" = "alt"
a = RINSTR(a$,c$,7)     ' a =  0    before match
a = RINSTRI(a$,b$)      ' a =  1    "HE" = "He"
a = RINSTRI(a$,c$)      ' a = 12    "alt" = "alt"
a = RINSTRI(a$,c$,4)    ' a =  3    "ALT" = "alt"
```

`INT()`
`FIX()`

intrinsic

float = `INT`(*float*)
float = `FIX`(*float*)

Return argument rounded toward nearest integer or toward zero.

```
b# =  2.552
c# = -2.552
w# = INT(b#)    ' w# =  3.000#
x# = INT(c#)    ' x# = -3.000#
y# = FIX(b#)    ' y# =  2.000#
z# = FIX(c#)    ' z# = -2.000#
```

`INTERNAL FUNCTION`

See `DECLARE FUNCTION`.

`LCASE$()`
`UCASE$()`

intrinsic

string = `LCASE$`(*string*)
string = `UCASE$`(*string*)

Convert all characters in a string to lower case or upper case.

```
a$ = LCASE$ ("THE big LIE")     ' a$ = "the big lie"
a$ = LCASE$ ("BIGGER LIES")     ' a$ = "bigger lies"
a$ = UCASE$ ("THE big LIE")     ' a$ = "THE BIG LIE"
a$ = UCASE$ ("bigger lies")     ' a$ = "BIGGER LIES"
```

LCLIP$()
RCLIP$()

intrinsic

string = `LCLIP$(`***string***`)`
string = `LCLIP$(`***string, count***`)`
string = `RCLIP$(`***string***`)`
string = `RCLIP$(`***string, count***`)`

Clip characters off the left or right end of a string.

```
a$ = LCLIP$ ("This old man", 5)   ' a$ = "old man"
a$ = LCLIP$ ("This old man", 15)  ' a$ = ""
a$ = RCLIP$ ("This old man", 5)   ' a$ = "This ol"
a$ = RCLIP$ ("This old man", 15)  ' a$ = ""
```

LEFT$()
RIGHT$()

intrinsic

string = `LEFT$(`***string***`)`
string = `LEFT$(`***string, length***`)`
string = `RIGHT$(`***string***`)`
string = `RIGHT$(`***string, length***`)`

Copy the first or last characters from a string. The first argument is the string to copy characters from. The second argument is the desired length of the result string, or 1 if no second argument is given. If the string has fewer characters than the desired length, the result string is a copy of the argument string.

```
x$ = "This old man"
a$ = LEFT$(x$)              ' a$ = "T"
a$ = LEFT$(x$, 5)           ' a$ = "This "
a$ = LEFT$(x$, 15)          ' a$ = "This old man"
a$ = RIGHT$(x$)             ' a$ = "n"
a$ = RIGHT$(x$, 5)          ' a$ = "d man"
a$ = RIGHT$(x$, 15)         ' a$ = "This old man"
```

LEN()

intrinsic

integer = `LEN(`***string***`)`

Return the number of characters in a string.

```
x$ = ""
y$ = "four"
a = LEN(x$)      ' a = 0
a = LEN(y$)      ' a = 4
```

LIBRARY()

integer = `LIBRARY(`***integer***`)`

Return `$$TRUE` if program is compiled as library.

```
FUNCTION  Entry ()
  Xui ()
  InitGui ()
  InitProgram ()
  IF LIBRARY(0) THEN RETURN   ' main program processes messages
'
  DO
    XgrProcessMessages (1)
  LOOP UNTIL #terminateProgram
END FUNCTION
```

LJUST$()

See `CJUST$()`.

LOF()

intrinsic

integer = `LOF(`***fileNumber***`)`

Return the length of a disk file. The argument is the filenumber.

```
FUNCTION GimmeFile (filename$)
```

```
  ifile = OPEN (filename$, $$RD)  ' open filename$ for read only
  length = LOF (ifile)            ' length = # of bytes in filename$
  a$ = NULL$ (length)             ' make a string that long
  READ [ifile], a$                ' and read the file into it
  CLOSE (ifile)                   ' then close the file
  RETURN (a$)                     ' and return the string
END FUNCTION
```

LOOP

See **DO**.

LTRIM$()

See **TRIM$()**.

MAKE()

intrinsic

integer = **MAKE(***integer, bitspec***)**
integer = **MAKE(***integer, width, offset***)**

Shift a field of bits up from the least significant bits. The first argument contains the source bitfield in its least significant bits. In the two argument version, the second argument is a bitspec that contains the width and offset of the bitfield to create. In the three argument version the second and third arguments are the width and offset of the bitfield to create. The bitfield of the specified width is shifted left from the least significant bits by offset bits. All bits outside the bitfield are cleared.

```
i = 0xAAAAAAAA        ' i = 0b10101010101010101010101010101010
j = BITFIELD(5,7)     ' j = bitspec for 5-bit wide field at bit 7-11
a = MAKE(i,j)         ' a = 0b00000000000000000000010100000000
                      '                           *****
```

MAX()
MIN()

intrinsic

numeric = **MAX(***numeric, numeric***)**
numeric = **MIN(***numeric, numeric***)**

Return the maximum or minimum of two values. Maximum means closest to positive infinity and negative means closest to negative infinity. Both values must be the same data type, and the result is the same type as the arguments.

```
a = 23
b = 123
c = -47
a# = 23
b# = 123.45
c# = .0001
d# = -4d+22
m = MAX(a,b)        ' m = 123
m = MAX(a,c)        ' m = 23
m# = MAX(a#,b#)     ' m# = 123#
m# = MAX(a#,c#)     ' m# = 23#
m# = MAX(a#,d#)     ' m# = 23#
m = MIN(a,b)        ' m = 23
m = MIN(a,c)        ' m = -47
m# = MIN(a#,b#)     ' m# = 23#
m# = MIN(a#,c#)     ' m# = .0001
m# = MIN(a#,d#)     ' m# = -4d+22
```

MID$()

intrinsic

string = **MID$(***string, start***)**
string = **MID$(***string, start, length***)**

Copy part of a string. The first argument is the string to copy from. The second argument is the position of the first character to copy. The third argument is the number of characters to copy. If no third argument is given, the rest of the characters in the string are copied.

```
x$ = "This old man"
a$ = MID$(a$, 5)          ' a$ = " old man"
a$ = MID$(a$, 5, 6)       ' a$ = " old m"
a$ = MID$(a$, 7, 4)       ' a$ = "ld m"
```

```
a$ = MID$(a$, 7, 9)       ' a$ = "ld man"
```

`MIN()`

See **`MAX()`**.

`MOD`

binary arithmetic operator

integers = `MOD` ***numeric*** → ***integers = numeric*** `MOD` ***numeric***

Return the integer modulus, which is the remainder from a division. Floating point operands are converted to **`XLONG`** before the modulus operation is performed.

```
a = 23 : b = 5 : c = 9 : d = 2
x = a MOD b                         ' x = 3  (23/5 = 4 + remainder 3)
x = b MOD a                         ' x = 5  (5/23 = 0 + remainder 5)
```

`NEXT`

See **`FOR`**.

`NOT`

unary bitwise operator

integer = `NOT` ***integer***

Return an integer with all bits inverted.

```
a = 0xAAAAFF00
a = NOT a         ' a = 0x555500FF
```

`NULL$()`

intrinsic

string = `NULL$(`***length***`)`

Create a string of null characters. The argument is the number of null characters in the result string. The string returned has the specified number of null characters, plus the additional null character that serves as a terminator. The null terminator is not part of the string and must not be changed.

```
FUNCTION GimmeFile (filename$)
  ifile = OPEN (filename$, $$RD)   ' open filename$ for read only
  length = LOF (ifile)             ' length = # of bytes in filename$
  a$ = NULL$ (length)              ' make a string that long
  READ [ifile], a$                 ' and read the file into it
  CLOSE (ifile)                    ' then close the file
  RETURN (a$)                      ' and return the string
END FUNCTION
```

`OCT$()`
`OCTO$()`

intrinsic

string = `OCT$(`***integers***`)`
string = `OCTO$(`***integers***`)`
string = `OCT(`***integers, digits***`)`
string = `OCTO$(`***integers, digits***`)`

Return the octal string representation of an integer. The return string has as many characters as necessary to represent the integer in octal, or the number specified by the second argument, whichever is larger.

```
a = 0o00007643210
a$ = OCT$(a)        ' a$ = "76543210"
a$ = OCTO$(a)       ' a$ = "0o76543210"
a$ = OCT$(a,2)      ' a$ = "76543210"
a$ = OCTO$(a,2)     ' a$ = "0x76543210"
a$ = OCT$(a,12)     ' a$ = "000076543210"
a$ = OCTO$(a,12)    ' a$ = "0x000076543210"
```

`OPEN()`

intrinsic

integer = `OPEN(`***fileName$, mode***`)`

Open a disk file. The first argument is the filename. The second argument contains an integer value that determines how the file will be open as follows (values from **`xst.dec`**):

```
0x00 = $$RD       = Open existing file for reading only.
0x01 = $$WR       = Open existing file for writing only.
```

```
0x02 = $$RW       = Open existing file for reading and writing.
0x03 = $$WRNEW    = Open new file for writing only.
0x04 = $$RWNEW    = Open new file for reading and writing.
0x10 = $$RDSHARE  = Open existing file for reading only.
0x20 = $$WRSHARE  = Open existing file for writing only.
0x30 = $$RWSHARE  = Open existing file for reading and writing.
```

Opening an existing file prepares the existing file for reading and/or writing, or if no file yet exists, creates a new file. Opening a new file deletes any existing file with the same name before it creates the new, empty file. The filenumber of the opened file is returned.

```
FUNCTION  UpperFile (filename$, newname$)
  ifile = OPEN (filename$, $$RD)
  ofile = OPEN (newname$, $$WRNEW)
  length = LOF (ifile)
  a$ = NULL$ (length)
  READ [ifile], a$
  WRITE [ofile], a$
  CLOSE (ifile)
  CLOSE (ofile)
END FUNCTION
```

OR

binary bitwise operator

integers = *integers* `OR` *integers*

Bitwise `OR` two integer operands to produce an integer result.

```
a = 0x05F5
b = 0xFFA0
c = a OR b   ' c = 0xFFF5
```

POF()

intrinsic

integer = `POF(`*fileNumber*`)`

Return the position of the file pointer for an open diskfile. The argument is the filenumber of the open file.

```
ifile = OPEN (filename$, $$RD) ' open a file to read
final = LOF(ifile)             ' final = length of filename$
line$ = INFILE$(ifile)         ' line$ = first text line in file
pos   = POF(ifile)             ' pos = file pointer after first line
```

PRINT

statement

`PRINT` [[*fileNumber*`,`]] [*arguments*] [`,` | `;`]

Print numeric and/or string arguments to display screen or disk file. No space is printed between arguments separated by a semi-colon, though each subsequent semi-colon prints one space. Arguments preceded by a comma are spaced to the next tab position before they are printed. A semi-colon at the end of the argument list suppresses the newline character (`"\n"`) that is otherwise printed.

```
age = 25
name$ = "Sophia"
a = 4
b$ = "cat"
c# = -3.3
PRINT                         ' Print a blank line
PRINT [ofile]                 ' Print a blank line to a file
PRINT "Her age is"; age       ' Print "Her age is 25"
PRINT "Her age is";;; age     ' Print "Her age is   25"
PRINT "Her name is "; name$   ' Print "Her name is Sophia"
PRINT "Her name is", name$    ' Print "Her name is    Sophia"
PRINT "Her name is",, name$   ' Print "Her name is        Sophia"
PRINT a; b$; c                ' Print " 4cat-3.3"
PRINT a, b$, c                ' Print " 4  cat -3.3"
PRINT c; b$, a                ' Print "-3.3cat 4"
PRINT [ofile], a, b, c        ' Print " 4  cat -3.3" to ofile
```

PROGRAM

`PROGRAM "`*name*`"`

statement

Begin and name a program. Argument must be a literal string. **PROGRAM** statements are required when compiling libraries.

```
PROGRAM  "xma"
```

PROGRAM$()

intrinsic

string **= PROGRAM$(***integer***)**

Return program name defined in **PROGRAM** statement.

```
program$ = PROGRAM$(0)
```

QUIT()

intrinsic

[*integer* **=] QUIT(***integer***)**

Quit a program and return to program that ran this one, which usually is the operating system. The return value is lost since the program is terminated before it can return from the call. The argument is passed to the program that ran the current program.

```
IF #terminate THEN
  DisasterHandler ()   ' prepare for quit
  QUIT()               ' quit the program
END IF
```

RCLIP$()

See **LCLIP$()**.

READ

statement

READ [*fileNumber***],** ***variables***

Read data from a disk file into variables. The first argument is the filenumber of the file to read from. The variables can be any combination of numeric variables, string variables, composite variables, composite variable components, or one dimensional arrays of any type except strings.

The number of bytes read into each variable equals the data size of the variable. For example, one byte is read into **SBYTE** and **UBYTE** variables, two bytes into **SSHORT** and **USHORT** variables, etc.

This is true even though simple variables shorter than 32-bit are held in memory as 32-bit or 64-bit values. In cases where the data size and storage size are not equal, the data size is read, zero or sign-extended to 32-bits or 64-bits, then saved in the storage variable.

When data is read into a string variable, the number of bytes needed to fill the string are read directly into the string, overwriting the previous contents.

When data is read into an array variable, the number of bytes needed to fill the array are read directly into the array, overwriting the previous contents.

READ and **WRITE** are complementary statements.

```
DIM a[31]
DIM a#[63]
a$ = NULL$(256)
READ [ifile], a@@, b@@, c@@ ' read 1 byte each into UBYTE variables
READ [ifile], a, b, c       ' read 4 bytes each into XLONG variables
READ [ifile], a!, b!, c!    ' read 4 bytes each into SINGLE varaibles
READ [ifile], a#, b#, c#    ' read 8 bytes each into DOUBLE variables
READ [ifile], a$            ' read 256 bytes to fill STRING a$
READ [ifile], a[]           ' read 128 bytes to fill XLONG a[31]
READ [ifile], a#[]          ' read 512 bytes to fill DOUBLE a#[63]
READ [ofile], pixel         ' read all bytes in composite variable
READ [ofile], pixel.color   ' read all bytes in component
READ [ofile], name.kid[]    ' read all bytes in component array
```

REDIM

See **DIM**.

RETURN
EXIT FUNCTION

statement

RETURN [*expression*]
EXIT FUNCTION [*expression*]

Return from the currently executing function to the one that called it. The value of the expression is returned to the calling function. If no expression is given, a zero or empty string is returned.

```
RETURN                     ' return zero or empty string
RETURN  a                  ' return value of a
RETURN  a$                 ' return string a$
RETURN ((a+b)*(c+d))       ' return value of numeric expression
RETURN (a$ + b$ + c$)      ' return value of string expression
```

RIGHT$()

See **LEFT$()**.

RINCHR()

See **INCHR()**.

RINCHRI()

See **INCHR()**.

RINSTR()

See **INSTR()**.

RINSTRI()

See **INSTR()**.

RJUST$()

See **CJUST$()**.

ROTATER()

instrinsic

integer = **ROTATER(***integer, count***)**

Closed rotate bits right any number of bits. Bits shifted out the least significant bit are recirculated back into the most significant bit.

```
x = 0xDEADC0DE
y = 0x01020304
a = ROTATER(x, 16)      ' a = 0xC0DEDEAD
a = ROTATER(y,  4)      ' a = 0x40102030
```

RTRIM$()

See **TRIM$()**.

SBYTE

See ***Declarations*** after the alphabetical listing.

SBYTE()

See ***Type Conversions*** after the alphabetical listing.

SBYTEAT()

See ***Direct Memory Access*** after the alphabetical listing.

SCOMPLEX

statement

[*scope*] **SCOMPLEX** *variables*

Declare single precision complex variables.

Also see ***Declarations*** after the alphabetical listing.

```
SCOMPLEX  ii, jj, kk
SCOMPLEX  x[], y[], z[]
SHARED SCOMPLEX  a, b, c, a[], b[], c[]
```

SEEK()

intrinsic

[*filePointer* =] **SEEK(***fileNumber, filePointer***)**

Move the file pointer of a file to a new position. The new position can be beyond the end of the file.

```
SEEK (dataFile, recordNumber * recordSize)
fp = SEEK(ifile, d3)
```

SELECT CASE
END SELECT

SELECT CASE TRUE
SELECT CASE FALSE
SELECT CASE *expression*

statement

SELECT CASE ALL TRUE
SELECT CASE ALL FALSE
SELECT CASE ALL *expression*
END SELECT

Begin a multi-way decision block and establish the test expression that subsequent **CASE** statements test against. The test expression can be **TRUE, FALSE,** or a numeric or string ***expression***.

Each expression in the subsequent **CASE** statements is tested agains the test expression. If any expression matches the test expression, the statements following the **CASE** statement are executed, up to the next **CASE** or **END SELECT** statement.

To match **TRUE** requires a non-zero number or a non-empty string. To match **FALSE** requires a zero number or an empty string. To match expression requires an equal numeric value or string.

Normal **SELECT CASE** blocks are called ***one-of-many*** decision blocks because at most one **CASE** statement block is executed. This occurs because an invisible jump past the matching **END SELECT** is inserted before every **CASE** statement except the first.

SELECT CASE ALL blocks are called ***n-of-many*** decision blocks ine because ***all*** **CASE** statements are tested for matches. Whether or not the code in the preceeding block was executed, tests are performed for each successive **CASE** statement.

END SELECT marks the end of every **SELECT CASE** block.

```
SELECT CASE a           ' test against the contents of a
SELECT CASE a$          ' test against the contents of a$
SELECT CASE TRUE        ' test for non-zero or non-empty string
SELECT CASE FALSE       ' test for zero or empty string
SELECT CASE ALL a       ' test all cases against the contents of a
SELECT CASE ALL a$      ' test all cases against the contents of a$
SELECT CASE ALL TRUE    ' test all cases for non-zero or non-empty$
SELECT CASE ALL FALSE   ' test all cases for zero or empty string
'
SELECT CASE x
  CASE a       : PRINT "a = x"
  CASE b,c,d   : PRINT "a or b or c = x"
  CASE e+f     : PRINT "a = e+f"
  CASE ELSE    : PRINT "None of the above"
END SELECT
'
SELECT CASE ALL TRUE
  CASE i, j, k          : PRINT "i or j or k is non-zero"
  CASE humidity > 100   : PRINT "The sky is falling"
  CASE a$, b$+c$        : PRINT "a$ or b$+c$ is has contents"
  CASE !raining         : PRINT "You can go out now"
  CASE ERROR(-1)        : PRINT "Something bad happened"
  CASE hope[]           : PRINT "There is hope[]"
  CASE ALL              : PRINT "All printed are true"
END SELECT
```

SET()

See **CLR()**.

SFUNCTION

See **FUNCTION**.

SGN()
SIGN()

integer = SGN(***numeric***)
integer = SIGN(***numeric***)

Return the sign of a number. If the argument is negative, return **-1**.

intrinsic

If the argument is positive, return `+1`. If the argument is zero, `SGN()` returns `0` while `SIGN()` returns `+1`.

```
i = -23
j =   0
k = +74
x# = -123.4
y# = +.0002
PRINT SGN(i), SIGN(i)      ' Print "-1  -1"
PRINT SGN(j), SIGN(j)      ' Print " 0  +1"
PRINT SGN(k), SIGN(k)      ' Print "+1  +1"
PRINT SGN(x#), SIGN(x#)    ' Print "-1  -1"
PRINT SGN(y#), SIGN(y#)    ' Print "+1  +1"
```

`SHARED`

See ***Declarations*** after the alphabetical listing.

`SHELL()`

intrinsic

[*integer* `=`] `SHELL(`***command$***`)`

Execute a program. The command string executes as it would at an operating system prompt except the full program name including extent must be specified. Control then returns control to the program. Some operating systems support concurrent execution of both processes, meaning the program does not wait until the newly started program completes.

```
SHELL ("xb.exe acircle.x -bc")   ' wait for xb.exe to complete
SHELL (":xb.exe acircle.x -bc")  ' execute concurrently if supported
```

`SIGN()`

See `SGN()`.

`SIGNED$()`

See `STRING()`.

`SINGLE`

See ***Declarations*** after the alphabetical listing.

`SINGLE()`

See ***Type Conversions*** after the alphabetical listing.

`SINGLEAT()`

See ***Direct Memory Access*** after the alphabetical listing.

`SIZE()`

intrinsic

integer `= SIZE(`***typename***`)`
integer `= SIZE(`***variable***`)`
integer `= SIZE(`***array***`[])`
integer `= SIZE(`***array***`[`***n***`])`
integer `= SIZE(`***array***`[`***n***`,])`
integer `= SIZE(`***composite***`)`
integer `= SIZE(`***composite.component***`)`
integer `= SIZE(`***composite.component***`[])`
integer `= SIZE(`***composite.component***`[`***n***`])`
integer `= SIZE(`***composite.componentString***`)`

Return the size of data in bytes. The argument can be a built-in or user-defined ***typename***, a numeric or string ***variable***, the highest dimension of an ***array***`[]`, an ***array***`[`***n***`]` element, an intermediate dimension of an ***array***`[`***n***`,]`, a ***composite*** variable, a ***component*** of a composite variable, a ***component***`[]` array, a ***component***`[`***n***`]` array element, or a ***componentString***.

```
name$ = "John Galt"
a = SIZE(SLONG)       ' size of built-in type SLONG (4 bytes)
a = SIZE(COLOR)       ' size of user-defined composite type COLOR
a = SIZE(employee)    ' size of variable 'employee'
a = SIZE(name$)       ' size of string variable (9 bytes)
a = SIZE(prog[])      ' size of highest dimension of prog[]
a = SIZE(prog[n,]     ' size of sub-array prog[n, ]
a = SIZE(prog[m,]     ' size of sub-array prog[m, ]
a = SIZE(box)         ' size of composite variable box
a = SIZE(box.top)     ' size of composite.component box.top
```

	`a = SIZE(box.ele[])    ' size of composite.array box.ele[]`
	`a = SIZE(box.ele[1])   ' size of composite.array element box.ele[1]`
	`a = SIZE(box.name)     ' size of composite string box.name`

SLONG	See ***Declarations*** after the alphabetical listing.
SLONG()	See ***Type Conversions*** after the alphabetical listing.
SLONGAT()	See ***Direct Memory Access*** after the alphabetical listing.

SMAKE()
 	
intrinsic
	single = **SMAKE(*integer*)**
	
	Change the type of a 32-bit integer to **SINGLE**. The bit pattern of the return value is the same as the integer argument, but the return type is **SINGLE**.
	
	`x = 0x48000000         ' x  = 0x48000000  ( a large integer number)`
	`a! = SMAKE(x)          ' a! = 0s48000000  ( a small SINGLE number)`

SPACE$()
 	
intrinsic
	string = **SPACE$(*length*)**
	
	Create a string of space characters. The length of the return string is given by the argument.
	
	`a$ = SPACE$(4)                ' a$ = "    "`
	`a$ = "x" + SPACE$(2) + "x"    ' a$ = "x  x"`

SSHORT	See ***Declarations*** after the alphabetical listing.
SSHORT()	See ***Type Conversions*** after the alphabetical listing.
SSHORTAT()	See ***Direct Memory Access*** after the alphabetical listing.
STATIC	See ***Declarations*** after the alphabetical listing.
STEP	See **FOR**.

STRING()
STRING$()
SIGNED$()
STR$()
 	
intrinsic
	string = **STRING(*numeric*)**
	string = **STRING$(*numeric*)**
	string = **SIGNED$(*numeric*)**
	string = **STR$(*numeric*)**
	
	Convert a numeric argument into a string. The only difference between these intrinsics is the form the sign takes in the returned string. **STRING()** and **STRING$()** puts a leading "-" on negative numbers and nothing on positive numbers. **SIGNED$()** puts a leading "+" or "-" on all numbers, including "+0". **STR$()** puts a leading "-" on negative numbers and a leading space character " " on positive numbers.
	
	`a# = -23.456`
	`b# = +11.111`
	`a$ = STRING(a#) + STRING(b#)     ' a$ = "-23.45611.111"`
	`a$ = STRING$(a#) + STRING$(b#)   ' a$ = "-23.45611.111"`
	`a$ = SIGNED$(a#) + SIGNED$(b#)   ' a$ = "-23.456+11.111"`
	`a$ = STR$(a#) + STR$(b#)         ' a$ = "-23.456 11.111"`

STUFF$()
 	
intrinsic
	string = **STUFF(*stringInto$, stringFrom$, start*)**
	string = **STUFF(*stringInto$, stringFrom$, start, length*)**
	
	Stuff one string into another. Stuff ***stringFrom$*** into ***stringInto$*** starting at ***start***. Stuff no more than ***length*** characters.
	
	`x$ = "This lazy man"`
	`y$ = "flabberghast"`
	`z$ = "is wolf"`
	`a$ = STUFF$(x$, y$, 6, 4)     ' a$ = "This flab man"`

	`a$ = STUFF$(x$, y$, 6)        ' a$ = "This flabberg"`
	`a$ = STUFF$(x$, z$, 6)        ' a$ = "This is wolfn"`
	`a$ = STUFF$(x$, z$, 6, 5)     ' a$ = "This is woman"`

SUB
EXIT SUB
END SUB
 	
statement
	SUB ***SubroutineName***
	
	SUB - Begin subroutine.
	EXIT SUB - Exit subroutine before its end.
	END SUB - End subroutine.
	
	A subroutine is a portion of a function that can be called only from within the same function by **GOSUB** ***SubroutineName***.
	
	END SUB returns to the statement following the **GOSUB**.
	EXIT SUB jumps to the **END SUB** which then returns.
	
	`SUB Sandwitch`
	`  entry = Convert (a, b)`
	`  IFZ entry THEN EXIT SUB      ' exit the subroutine`
	`  entry = Chew (a, b)`
	`END SUB`

SUBADDR	See ***Declarations*** after the alphabetical listing.
SUBADDR()	See ***Type Conversions*** after the alphabetical listing.
SUBADDRAT()	See ***Direct Memory Access*** after the alphabetical listing.

SUBADDRESS()
 	
instrinsic
	subaddr **= SUBADDRESS(*label*)**
	
	Return the address of a subroutine.
	
	`g = SUBADDRESS (SubName)      ' g = address of subroutine SubName`
	`h[i] = SUBADDRESS (SumArray)  ' h[i] = address of subroutine SumArray`

SWAP
 	
statement
	SWAP ***string, string***
	SWAP ***variable, variable***
	SWAP ***arrayNode, arrayNode***
	
	Swap the contents of:
	• ***string variables***
	• ***numeric variables of the same type***
	• ***array elements of the same type***
	• ***strings, arrays, array nodes***
	
	`SWAP a, b           ' swap XLONG variables`
	`SWAP a#, b#         ' swap DOUBLE variables`
	`SWAP a$, b$         ' swap STRING variables`
	`SWAP a, b[n]        ' swap XLONG variable with XLONG array element`
	`SWAP a[i], b[j]     ' swap XLONG array elements`
	`SWAP a$[i], b$[j]   ' swap STRING array elements`
	`SWAP a$[i], b$      ' swap STRING array element with string`
	`SWAP a[], b[]       ' swap two entire arrays`
	`SWAP a[], b[i,]     ' swap array with a sub-array`
	`SWAP a[i,], b[i,]   ' swap two sub-arrays`

TAB()
 	
intrinsic
	TAB(*integer*)
	
	Append spaces to **PRINT** string to reach a horizontal character position. **TAB()** is valid only in **PRINT** statements.
	
	`PRINT a; TAB(10); b; TAB(24); c; TAB(40); d$; TAB(60); e`
	`PRINT TAB(40); x, y, z`

THEN	See **IF**.

TO

See **FOR**.

TRIM$()
LTRIM$()
RTRIM$()

intrinsic

string = TRIM$(*string*)
string = LTRIM$(*string*)
string = RTRIM$(*string*)

Trim whitespace and non-printable characters from both ends of a string, the left end of a string, or the right end of a string. All characters from `0x00` to `0x20` and `0x80` to `0xFF` are removed.

```
x$ = "\n\nXXX\t\0\1\2  "
y$ = "\t  \nZZZ  \t\t\n"
a$ = LTRIM$(x$)               ' a$ = "XXX\t\0\1\2  "
a$ = LTRIM$(y$)               ' a$ = "ZZZ  \t\t\n"
a$ = RTRIM$(x$)               ' a$ = "\n\nXXX"
a$ = RTRIM$(y$)               ' a$ = "\t  \nZZZ"
a$ = TRIM$(x$)                ' a$ = "XXX"
a$ = TRIM$(y$)                ' a$ = "YYY"
```

TRUE

See **SELECT CASE**.

TYPE
UNION
END TYPE

statement

TYPE *TypeName*
UNION
END TYPE
END UNION

Declare user-defined composite types and begin/end union overlays. **UNION** statements are valid only within **TYPE** declaration blocks.

```
TYPE COLOR
  USHORT  .red
  USHORT  .green
  USHORT  .blue
  USHORT  .intensity
END TYPE
'
TYPE POINT
  XLONG   .x
  XLONG   .y
  COLOR   .color
END TYPE
'
TYPE LINE
  POINT   .a
  POINT   .b
  XLONG   .thickness
END TYPE
'
TYPE DPOINT
  DOUBLE  .x
  DOUBLE  .y
  COLOR   .color
END TYPE
'
TYPE ARBITRARY            ' hold an arbitrary data type value
  UNION                   ' the following overlay each other
  SBYTE     .sbyte
  UBYTE     .ubyte
  SSHORT    .sshort
  USHORT    .ushort
  SLONG     .slong
  ULONG     .ulong
  XLONG     .xlong
  GIANT     .giant
  SINGLE    .single
  DOUBLE    .double
  SCOMPLEX  .scomplex
  DCOMPLEX  .dcomplex
  END UNION               ' end overlay
  XLONG     .type         ' type currently stored in the variable
END TYPE
```

TYPE()

integer = TYPE(*typeName*)

intrinsic

integer = `TYPE(`*typeName.component*`)`
integer = `TYPE(`*variable*`)`
integer = `TYPE(`*variable.component*`)`
integer = `TYPE(`*array*`[])`
integer = `TYPE(`*array*`[`*subscripts*`,])`

Return the type number of a built-in type, composite type, composite type component, variable, component of a composite variable, array, or array node.

Shared constants are defined for the built-in types. User-defined types are assigned during compilation. A given user-defined type is not necessarily assigned the same type number on subsequent compilations, or from library to library. Don't make any assumptions about the assignment of type numbers in function libraries or other external modules.

```
a = TYPE(SLONG)            ' a = type number of SLONG     ($$SLONG)
a = TYPE(DCOMPLEX)         ' a = type number of DCOMPLEX  ($$DCOMPLEX)
a = TYPE(DCOMPLEX.I)       ' a = type number of component ($$DOUBLE)
a = TYPE(employee)         ' a = type number of variable employee
a = TYPE(maxValue[])       ' a = type number of array maxValue[]
a = TYPE(prog[func,])      ' a = type number of node      ($$XLONG)
a = TYPE(employee.salary)  ' a = type number of variable component
```

`UBOUND()`

intrinsic

integer = `UBOUND(`*string*`)`
integer = `UBOUND(`*array*`[])`
integer = `UBOUND(`*array*`[`*subscripts*`,])`

Return the upper bound of a string or array dimension.

```
x$ = ""
y$ = "0"
z$ = "012345"
DIM a[]
DIM b[0]
DIM c[7]
DIM d[3,]
DIM x[11]
DIM y[15]
DIM z[31]
ATTACH x[] TO d[0]
ATTACH y[] TO d[1]
ATTACH z[] TO d[2]
a = UBOUND(x$)      ' a = -1    ( empty string )
a = UBOUND(y$)      ' a =  0    ( upper bound = 0 : 1 element )
a = UBOUND(z$)      ' a =  5    ( upper bound = 5 : 6 elements )
a = UBOUND(a[])     ' a = -1    ( empty array )
a = UBOUND(b[])     ' a =  0    ( upper bound = 0 : 1 element )
a = UBOUND(c[])     ' a =  7    ( upper bound = 7 : 8 elements )
a = UBOUND(d[])     ' a =  3    ( upper bound of highest dimension )
a = UBOUND(d[0,])   ' a = 11    ( upper bound of d[0,] )
a = UBOUND(d[1,])   ' a = 15    ( upper bound of d[1,] )
a = UBOUND(d[2,])   ' a = 31    ( upper bound of d[2,] )
a = UBOUND(d[3,])   ' a = -1    ( d[3,] is an empty array )
```

`UBYTE`	See ***Declarations*** after the alphabetical listing.
`UBYTE()`	See ***Type Conversions*** after the alphabetical listing.
`UBYTEAT()`	See ***Direct Memory Access*** after the alphabetical listing.
`UCASE$()`	See **`LCASE$()`**.
`ULONG`	See ***Declarations*** after the alphabetical listing.
`ULONG()`	See ***Type Conversions*** after the alphabetical listing.
`ULONGAT()`	See ***Direct Memory Access*** after the alphabetical listing.

UNION	See **TYPE**.
UNTIL	See **DO**.
USHORT	See ***Declarations*** after the alphabetical listing.
USHORT()	See ***Type Conversions*** after the alphabetical listing.
USHORTAT()	See ***Direct Memory Access*** after the alphabetical listing.

VERSION

statement

VERSION "*number*"

Give the program a version number. Argument must be literal string.

```
VERSION  "0.1234"
```

VERSION$()

intrinsic

string **= VERSION$(*integer*)**

Return version number string defined in **VERSION** statement.

```
version$ = VERSION$(0)
```

VOID	See **DECLARE** and **FUNCTION**.
WHILE	See **DO**.

WRITE

statement

WRITE [*fileNumber*], *variables*

Write variables to a disk file. The first argument is the filenumber of the file to write to. The remaining arguments are any combination of numeric variables, string variables, composite variables, composite variable components, or one dimensional arrays of any type except strings.

The number of byte written from each variable equals the data size of the variable. For example, one byte is written from **SBYTE** and **UBYTE** variables, two bytes from **SSHORT** and **USHORT** variables, four bytes from **SLONG** and **SINGLE** variables, etc.

This is true even though simple variables shorter than 32-bits are held in memory as 32-bit or 64-bit values. In cases where the data size and storage size are not equal, the least significant part of the variable is written.

When a string variable is written, the number of bytes written is the number of bytes in the string..

When an array variable is written, the number of bytes written is the number of bytes in the array. Only one dimension can be written.

```
DIM a[31]                  ' a[] contains 32 XLONG values
DIM a#[63]                 ' a#[] contains 64 DOUBLE values
a$ = CHR$('*', 256)        ' a$ contains 256 '*' characters
WRITE [ifile], a@, b@, c@  ' WRITE 1 byte each from SBYTE variables
WRITE [ifile], a, b, c     ' WRITE 4 bytes each from XLONG variables
WRITE [ifile], a!, b!, c!  ' WRITE 4 bytes each from SINGLE variables
WRITE [ifile], a#, b#, c#  ' WRITE 8 bytes each from DOUBLE variables
WRITE [ifile], a$          ' WRITE all 256 bytes from STRING a$
WRITE [ifile], a[]         ' WRITE all 128 bytes from XLONG a[31]
WRITE [ifile], a#[]        ' WRITE all 512 bytes from DOUBLE a#[63]
WRITE [ofile], pixel       ' WRITE all bytes in composite variable
WRITE [ofile], pixel.color ' WRITE all bytes in component
WRITE [ofile], name.kid[]  ' WRITE all bytes in component array
```

XLONG	See ***Declarations*** after the alphabetical listing.
XLONG()	See ***Type Conversions*** after the alphabetical listing.
XLONGAT()	See ***Direct Memory Access*** after the alphabetical listing.

XMAKE()
 	
intrinsic
	***xlong* = XMAKE(*integer*)**
	***xlong* = XMAKE(*single*)**
	
	Make an **XLONG** value from an integer or single argument.
	
	`x! = 2.000`
	`a = XMAKE(x!)        ' a = XLONG with same bit pattern as x!`

Declaration

statements

AUTO
AUTOX
STATIC
SHARED
EXTERNAL

SBYTE
UBYTE
SSHORT
USHORT
SLONG
ULONG
XLONG
GOADDR
SUBADDR
FUNCADDR
GIANT
SINGLE
DOUBLE
STRING
SCOMPLEX
DCOMPLEX

[*scope*] [*type*] *variables*

Declare the scope and data type of variables. Variables that do not appear in a declaration statement default to **AUTO** scope and the default data type, which is **XLONG** unless an explicit default type is specified on the **FUNCTION** line.

Composite variables with user-defined types are declared in the same way as built-in type variables. Substitute the name of the composite type for **DOUBLE** in the syntax example above.

The valid scopes are:

AUTO	local, temporary, possibly registered
AUTOX	local, temporary, never registered
STATIC	local, permanent
SHARED	sharable within program
SHARED /name/	sharable within program (/name/ must match)
EXTERNAL	sharable between programs
EXTERNAL /nane/	sharable between programs (/name/ must match)

The built-in data types are:

SBYTE	signed byte	8-bit integer
UBYTE	unsigned byte	8-bit integer
SSHORT	signed short	16-bit integer
USHORT	unsigned short	16-bit integer
SLONG	signed long	32-bit integer
ULONG	unsigned long	32-bit integer
XLONG	register type	32-bit integer (or 64-bit integer)
GOADDR	address type	32-bit integer (or 64-bit integer)
SUBADDR	address type	32-bit integer (or 64-bit integer)
FUNCADDR	address type	32-bit integer (or 64-bit integer)
GIANT	signed giant	64-bit integer
SINGLE	single float	32-bit floating point
DOUBLE	double float	64-bit floating point
STRING	ubyte string	8-bit unsigned byte characters
SCOMPLEX	single complex	32-bit : 32-bit floating point
DCOMPLEX	double complex	64-bit : 64-bit floating point

```
DOUBLE  ii, jj, kk
SINGLE  x[], y[], z[]
AUTOX USHORT  width[], height[]
STATIC DCOMPLEX  a, b, c, d, e, f
SHARED UBYTE  a[], b[], c[]
SHARED STRING  victum, name[], address[], phone[], excuse[]
```

Type Conversion	***type* = *type*(*numString*)**

intrinsics

`SBYTE()`
`UBYTE()`
`SSHORT()`
`USHORT()`
`SLONG()`
`ULONG()`
`XLONG()`
`GOADDR()`
`SUBADDR()`
`FUNCADDR()`
`GIANT()`
`SINGLE()`
`DOUBLE()`
`STRING()`
`STRING$()`

Convert a numeric or string value to the specified ***type***.

The argument can be any built-in numeric or string data type, except the complex number types. The argument is converted to the data type specified by the name of the intrinsic.

These intrinsics will convert any built-in numeric or string type to any other. The return type is always the name of the intrinsic.

Range checking is performed by these conversion intrinsics, even when conversion is not required. An overflow error occurs at runtime when a value cannot be represented in the specified return type.

```
a# = DOUBLE(temp)
a# = DOUBLE(value$)
a# = DOUBLE(a*b+c*d)
a@@[i] = UBYTE(n*p+q)
a[n] = USHORT(i+j+k)
a$ = STRING$(i&)
a$ = STRING$(j!)
a$ = STRING$(k#)
```

Direct Memory Access

intrinsics

SBYTEAT()
UBYTEAT()
SSHORTAT()
USHORTAT()
SLONGAT()
ULONGAT()
XLONGAT()
GIANTAT()
SINGLEAT()
DOUBLEAT()

double **= DOUBLEAT(*****address*****)**
double **= DOUBLEAT(*****address, offset*****)**
double **= DOUBLEAT(*****address,*** **[*****element*****])**
DOUBLEAT(***address*****) =** ***double***
DOUBLEAT(***address, offset*****) =** ***double***
DOUBLEAT(***address,*** **[*****element*****]) =** ***double***

The ***Direct Memory Access*** intrinsics read directly from a memory address, or write directly to a memory address. They are provided for the rare cases it is necessary to read or write a memory address directly, as when the operating system or another language provides an address of data, rather than data itself.

Address validation is not performed, so misaligned accesses, segment faults, and other memory access errors are not prevented. Therefore these intrinsics should be avoided when possible, and used with care. The name of any direct memory access intrinsic can be substituted for **DOUBLEAT** in any of the six syntax examples shown above.

The first three forms ***read*** from a memory address, while the second three forms ***write*** to a memory address.

The address is specified in one of three ways:

address	***address*** **=** ***address***
address, offset	***address*** **=** ***address*** **+** ***offset***
address, **[*****element*****]**	***address*** **=** ***address*** **+ (*****element*** ***** ***size*****)**

In the first form, the address is given directly. In the second form, the address is computed by adding a byte offset to the address. In the third form, the address is computed by adding the element number times its size in bytes to the address.

```
a# = DOUBLEAT(address)
a# = DOUBLEAT(address, offset)
a# = DOUBLEAT(address, [element])
DOUBLEAT(address) = a#
DOUBLEAT(address, offset) = a#
DOUBLEAT(address, [element]) = a#
```