Xbasic for Linux- A Quick Reference Guide for the great freeware 32/64 bit programming language **- Xbasic for Windows**

A **"symbol"** in computer terms is a mixture of letters, digits, and possibly underlines, representing a number, variable, function, keyword, or label. Other characters may be operators, separators, prefixes or suffixes, or in some cases have no particular meaning. A symbol that is followed by a **$ is a variable that will contain a string,** (that is, some text). A symbol that is followed by some other characters, like **@, %, &, ~, $$, !** or **# is a numeric variable**.

Many of those characters can have a different meaning in some other context. (**A symbol with no suffix is also a numeric variable**.) A symbol followed **by a bracket [** is the name of an **array,** there will be 0 or more expressions following the bracket until the closing **bracket]** is found which will indicate which element or part of the array is referred to. **Arrays** may use **suffixes** before the bracket to indicate what type of information an element can contain. A symbol followed by a **parenthesis (** is a **function** name, it will be followed by 0 or more function arguments until a closing **parenthesis)** is found - and **functions** also can have a **suffix** before the parenthesis indicating the type of result they produce. A comma, period (except when used as a decimal point), colon, or semicolon is **a separator**. Besides their use with functions, **parentheses ()** can be used for grouping as is common in math. **Brackets [ ]**can be used to indicate a file. **Braces { }** indicate a **bitmask** or character in a string. **Double quotes** indicate a string, **Single quotes** can be either a single character or a comment. Other symbols are used mostly for math.

True/False is always returned as XLONG. 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. **String variables & arrays** start out empty and are **FALSE**.

A string with **a null character**, and **an array with one element are not empty**, and are therefore **TRUE**.

All of the following are made **TRUE** by the following statements: **DIM** a[63] : **DIM** a[0] : **DIM** a$[63] : **DIM** a$[0]. All of the array variables are made **FALSE** by the following statements: **DIM** a[] : **REDIM** a[] : **SWAP** a[], b[c,] ' **If b[c,] is empty** then a[] becomes empty: **ATTACH** a[] **TO** a[]b[c,] ' a[] becomes empty :

The bitwise **AND** (also written **&**), is used for bit masks and similar things. The other is a **"logical" and**, which is written as **&&.I**t makes more sense for the bit mask usage to have a higher priority than the comparison. If you really wanted to use **AND** for a bit mask in an IF, you'd have to use parentheses to force it to work properly.

AND can't operate on strings, generates a compiler error. *Example*: **IF** (a$ = "Y") **AND** (b$ = "Y") **THEN** ………*or use* ……….**IF** a$ = "Y" **&&** b$ = "Y" **THEN** ……….. *A quick comparison of* **AND** *and* **&&** ...5 **AND** 5 = 5, 5 **&&** 5 = **TRUE (-1)** : 3 **AND** 5 = 1, **3 && 5** = **TRUE**: 2 **AND** 5 = 0, 2 **&&** 5 = TRUE : 0 **AND** 5 = 0, 0 **&&** 5 = **FALSE** (0) : 0 **AND** 0 = 0, 0 **&&** 0 = **FALSE**

Similarly, there is both a bitwise **OR** (also written **|**) and a **logical or** (written **||**), a bitwise **NOT** (also written **~**) and a **logical not** (written **!**), a bitwise **XOR** (also **^**) and a logical exclusive or (written **^^**). Use **||** instead of **OR** for **logical or** operators: example: **IF** (result_type = **$$NAME**) **||** (result_type = **$$ myType THEN** ………Don't confuse **TRUE** vis the pre-defined constant **$$TRUE** –**1.**

IF a **THEN PRINT** "a" prints a if a **<>** 0: **IF** (a = **$$TRUE**) **THEN PRINT** "a" prints a if a = **–1:** **FALSE** vs the pre-defined constant **$$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.

XBasic SCOPES for variables

Prefix	Scope	lifetime	accessible	Comment
	AUTO	function	function	temporary, local, maybe in CPU register
	AUTOX	function	function	temporary, local, never in CPU register
	STATIC	program	program	permanent, within local function
#	**SHARED**	program	program	permanent, shared with program (all Functions)
##	**EXTERNAL**	program	linked	permanent, shared with linked module
$	**LOCAL Constan**t	function	function	constant visible within one function
$$	**SHARED Constant**	program	program	constant visible throughout a program

Xbasic was written by Max Reason & is now in the public domain.

XBasic BACKSLASH characters non-printable characters for imbedding in literal strings

character	Hex /Octal	code	ASCI I	character	Hex/Octal	code	ASCII	Back Slash = Escape Character
\0	0x00	null	00	**\e**	0x1B	escape	27	To use a back slash in a string you must use two back slashes, as Xbasic sees the first as an escape character. The **Xbasic Constant $$PathSlash$** is a path separator instead of "\\". It works for both **LINIX** and **Windows**.
\a	0x07	alarm (bell)	07	**\f**	0x0C	form-feed	12	
\b	0x08	backspace	08	**\n**	0x0A	newline	10	
\d	0x7F	delete	127	**\r**	0x0D	return	13	
\t	0x09	tab	09	**\"**	0x22	double-quote	34	
\v	0x0B	vertical-tab	11	**\000**	0o000	octal value	-	
\	0x5C	**backslash**	92	**\xHH**	0xHH	hex value 0xHH	-	
\'	0x27	single-quote	39					

**Xbasic supports: IF True, IF FALSE, IF Zero, plus
IF.., ELSE.., EXIT IF.., END IF, blocks each on a separate line**
	IF…this…
	THEN
		http://gnetools.sourceforge.net/xbsupport/
		http://xbnotes.freehosting.net/
			**Default numeric type is XLONG
			if not declared in PROLOG**
				DECLARE FUNCTION **[type]** FuncName ([arglist])
				DECLARE FUNCTION *STRING* DisplayFiles (path$, type$, data$[]

IF f() **THEN PRINT** "f() returned **TRUE**, (non-zero)" .Normal IF	…do this	http://www.maxreason.com/software/xbasic/xbasic.html		
IFT f() **THEN PRINT** "f() returned **TRUE**, (non-zero)" **Xbasic's IF true**	**ELSE** …	▲ Many pages of useful information ▲	**In the Function it self**	**FUNCTION STRING** DisplayFiles (path$, type$, data$[])
IFF h() **THEN PRINT** "h() returned **FALSE**, (zero)" **Xbasic's IF false**	..do this	▼ New for 2021 ▼	To call the Function	x$ = DisplayFiles (**@**path$, **@**type$, **@**data$[])
IFZ f() **THEN PRINT** "f() returned **ZERO**, (zero)" **Xbasic's IF zero**	**END IF**	User group & GITHUB sharing site for xbasic	**SUB**	Subroutines exist within functions, and share all variables in the
EXIT IF [level] :Jump past the nth IF statement, where n=1 or level. EG: **EXIT IF 4**		https://groups.io/g/MaxReasonsxBasic/messages		In the function. **EXIT SUB** can be used to conditionally exit a **SUB**.
True/False returns an XLONG number , The constant $$TRUE returns -1:		**https://github.com/xbwlteam**	**END SUB**	**SHARED** and **STATIC** variables retain values after exiting the function.
$$FALSE returns a numeric value of 0				**RETURN** or **EXIT FUNCTION** statements can be used to conditionally
Xbasic supports several **SELECT CASE** arguments in addition to the usual **SELECT CASE** :			**RETURN(#a$)**	exit the function. **#a$** is a **SHARED type *STRING*.**
SELECT CASE <test expression > **CASE 1, CASE B , CASE ELSE, END SELECT** block of statements (*each on a separate line*).			**EXIT FUNCTION**	This allows **RETURN, EXIT FUNCTION** or
SELECT CASE [ALL] n of many true cases will execute: **SELECT CASE TRUE** first true case will execute:			**END FUNCTION(a$)**	**END FUNCTION** to return a **STRING** (**#a$**) to the calling **FUNCTION.**
SELECT CASE FALSE first false case will execute: **NEXT CASE** transfers execution directly to the next **CASE** statement:			**STATIC**, variables retain values within their Function. **Other functions can't see or modify static variables .**	
SELECT CASE ALL TRUE ALL cases that are true will execute: **SELECT CASE ALL FALSE** ALL cases that are false will execute.			**SHARED** variables **(prefix #) are available to, and may be modified by other functions.**	
EXIT SELECT (n) n is assumed to be 1 if blank , EXIT SELECT 2 would exit out of 2 levels of nested SELECT CASE statements or see GOTO below				

Xbasic's GOTO label; labels must start in column 1 & end with colon ; First letter in the label must be lower case. First letter of each imbedded word should be upper case EG; lable2: , clearGrid2: see also GOTO @goVar & GOTO @goArray[i]

Xbasic's DO ... LOOP : also supports: **DO WHILE, DO UNTIL,** with conditional branching via: **DO DO** continue at the top **(the DO)**, **DO LOOP** continue at the bottom (**the LOOP), EXIT DO** exit (past the **LOOP)**

Xbasic's FOR. x = 1 **TO** 10 **STEP** 2: (do something) **NEXT** x: also supports: conditional branching via :**DO FOR** continue at the top (the **FOR**), **DO NEXT** continue at the bottom (the **NEXT**), **EXIT FOR** exit (past the **NEXT**)

Xbasic for Linux- A Quick Reference Guide for the great freeware 32/64 bit programming language - Xbasic for Windows

Xbasic keywords are in upper case. Xbasic is case sensitive: `FOR` is a keyword while for, foR, For, FOr, FoR, fOr, fOR are seven valid independent symbols. It is very bad programming to use them

Xbasic operators - 1

op.	alt.	kind	operands	returns	class	precedence	comment
		unary	any-type	**XLONG**	11	12	address of data
&&		unary	any-type	**XLONG**	11	12	address of handle
NOT	~	unary	integer	same-type	10	12	not - bitwise
!		unary	numeric	**TRUE/FALSE**	9	12	not - logical : true if 0 / empty11
!!		unary	numeric	**TRUE/FALSE**	9	12	test - logical : false if 0 / empty
+		unary	numeric	same-type	8	12	positive - sign
-		unary	numeric	same-type	8	12	negative - sign
<<<		binary	integer	left-type	7	11	up-shift - arithmetic / signed
>>>		binary	integer	left-type	7	11	down-shift - arithmetic / signed
<<		binary	integer	left-type	7	11	up-shift - logical / shift in zero
>>		binary	integer	left-type	7	11	down-shift - logical / shift in zero
******		binary	numeric	high-type	4	10	power - raise to power
/		binary	numeric	high-type	4	9	divide
*****		binary	numeric	high-type	4	9	multiply
		binary	numeric	Integer	6	9	divide - integer
MOD		binary	numeric	Integer	6	9	modulus - integer remainder

Xbasic operators - continued

op.	alt.	kind	operands	returns	class	precedence	comment
+		binary	numeric	high-type	5	8	addition
+		binary	string	**STRING**	5	8	concatenate strings
-		binary	numeric	high-type	4	8	subtraction
AND	**&**	binary	integer	high-type	3	7	and - bitwise
XOR	**^**	binary	integer	high-type	3	6	xor - bitwise
OR	**|**	binary	integer	high-type	3	6	or - bitwise
>	**!<=**	binary	Num. - string	**TRUE/FALSE**	2	5	greater-than
>=	**!<**	binary	Num. - strring	**TRUE/FALSE**	2	5	greater-or-equal
<=	**!>**	binary	Num. - string	**TRUE/FALSE**	2	5	L ess-or-equal
<	**!>=**	binary	Num. - string	**TRUE/FALSE**	2	5	less-than
<>	**!=**	binary	Num. - strring	**TRUE/FALSE**	2	4	not-equal
=	**==**	binary	Num. - string	**TRUE/FALSE**	2	4	equal
&&		binary	integer	**TRUE/FALSE**	1	3	and - logical
^^		binary	integer	**TRUE/FALSE**	1	2	xor - logical
||		binary	integer	**TRUE/FALSE**	1	2	or - logical
=		binary	Num. - string	right-type		1	assignment

You can pass arguments in functions by value or by reference. All strings & arrays MUST be passed by reference & must use the @ prefix. EG: MyFunction (v0, @v1, @string$, @array[]) Only v0 is passed by value : Vi, String$,& array[] by reference.

Xbasic was written by Max Reason & is now in the public domain. **Operators & and && are prefixed to variables when calling 3rd party DLL FUNCTIONS**

The unary address operator & returns the memory address of: an array, array node, array data element, or composite element

operator + variable	example	operator and variable	example
&numeric-variable	&count	**&string-array-data**	&name$[dept, stationNumber]
&string-variable	&name$	**&& returns the handle address of string variable, composite variable, etc**	
&whole-array	&token[]	**&&string-variable(non-AUTO)**	&&name$
&whole-string-array	&symbols$[]	**&&whole-array (non-AUTO)**	&&token[]
&array-node	&token[func,]	**&&whole-string-array (non-AUTO)**	&&symbols$[]
&array-data	&token[func, line, element]	**&&string-array-data**	&&name$[dept, stationNumber]
&string-array-node	&name$[dept,]	**...(same result as above)**	&name$[dept, stationNumber,]

comments

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 composite variables is always valid.

&& returns the **handle address** of a **string variable, composite variable, whole array, or string array element.** Numeric variables and components of composite variables have no handles. 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.

XBasic DATA TYPES - suffix Note the XLONG is the default, not the short integer as in other programming languages

type suffix	data type	size	example	minimum value	maximum value	coersion aka type conversion	
@	SBYTE	8-bit signed byte integer	mySbyte@	-128	+127	mySbyte@ = SBYTE(a$)	Convert to SBYTE
@@	UBYTE	8-bit unsigned byte integer	myUbyte@@	0	+255	myUbyte@@ = UBYTE(a$)	Convert to UBYTE
%	SSHORT	16-bit signed short integer	mySshort%	-32,768	+32,767	mySshort% = SSHORT(a$)	Convert to SSHORT
%%	USHORT	16-bit unsigned short integer	myUshort%%	0	+65,535	myUshort%% = USHORT(a$)	Convert to USHORT
&	SLONG	32-bit signed long integer	mySlong&	-2147483648	+2147483647	mySlong& = SLONG(a$)	Convert to SLONG
&&	ULONG	32-bit unsigned long integer	myUlong&&	0	+4294967395	myUlong&& = ULONG(a$)	Convert to ULONG
~	**XLONG**	**32/64-bit signed machine integer**	**myXlong~**	**MIN SLONG 32b / GIANT 64b**	**MAX SLONG 32b / GIANT 64b**	**myXlong~ = XLONG(a$)**	**Convert to XLONG**
	GOTOADDR	32/64 -bit computed GOTO address	myGoHome	MIN SLONG	MAX SLONG	myGoHome = GOADDR(a$)	Convert to GOADDR
	SUBADDR	32/64 -bit computed GOSUB address	myGoSubOne	MIN SLONG	MAX SLONG	myGoSubOne = SUBADDR(a$)	Convert to SUBADDR
	FUNCADDR	32/64 -bit computed FUNCTION address	myFuncTwo	MIN SLONG	MAX SLONG	myFuncTwo = FUNCADDR(a$)	Convert to FUNCADDR
$$	GIANT	64-bit signed giant (financial) integer	myGiant$$	-9223372036854775808	+9223372036854775807	myGiant$$ = GIANT(a$)	Convert to GIANT
!	SINGLE	32-bit IEEE single precision floating point	mySingle!	-le38	le38	mySingle! = SINGLE(a$)	Convert to SINGLE
#	DOUBLE	64-bit IEEE double precision floating point	myDouble#	ld308	ld308	myDouble = DOUBLE(a$)	Convert to DOUBLE
$	STRING	String of unsigned bytes	myString$	Zero characters	2147483647 characters	myString$ = STRING(num&)	Convert to STRING
	SCOMPLEX	64 – bit IEEE Single Complex				x$ = STRING$(numb&)	Ditto
	DCOMPLEX	128 - bit IEEE Double Complex					

Linux uses only a new line character, **"\n" =CHR$(10)** to end a text line, many word processors can accept that. **DOS** and other programs require a form feed & a new line **"\r"+"\n" = CHR$(13) + CHR$(10)**.at the end of a text line.
When creating a path string for a file use **$$PathSlash$** which automatically converts to the correct path separator **"\\"** or = **'\\'** for Windows, and **"/"** or **'/'** for LINUX. Many but not all programs can be written that will ru n on both Windows and Linux platforms. Third party Dynamic Link Libraries can be used if some one has created a .DEC file for them. Or you can create your own libraries and .DEC files.

Join the new support group: https://groups.io/g/MaxReasonsxBasic/messages LATEST Versions: xbasic-6.3.26-win32-src-180610.zip & xbasic-6.3.26-linux32-src-180610-tar-gz June. 2021