x16lib v0.1

by Unartic

x16lib has subroutines to extend BASIC and aid assembly developers with commonly used commands.

General
X16lib is loaded in a rambank of choice. The rambank in which x16lib is loaded is not available to the main program anymore. X16lib uses the 16 bit registers r0 to r16 for various purposes and depending on which function is used, a set of these registers is used. Also address range $22 to $34 is used.

Notes for BASIC programmers
X16lib taps directly into the variable space of basic for fast use of variable data. You as a programmer have to make sure the rambank on which x16lib is loaded is active when calling a function from x16lib. The subdirectoy /basic contains examples, both in untokenized basic files (*.bas) as in tokenized basic files (*.prg).

The content of .bas files can by pasted directly into the emulator window. The .prg files can be loaded via basic: LOAD “EXAMPLE.PRG”

X16lib uses dedicated variable names: XF$,X1$,X1%,X2$,X2%,XS$,XS%
These variables should not be used in your basic program for other purposes.

As you’ll see in several examples x16lib can use BASIC string variables in two ways:

1. You can directly set a x16lib string variable:
```
X1$=”THIS IS MY STRING”
```
2. Or you can indirectly set a string variable:
```
A$=”THIS IS MY STRING”
X1$=”A$”
```

In both cases the string “THIS IS MY STRING” is passed to x16lib.

Before the first use of x16lib, it must be loaded into highram like this:

```
BANK 1:BLOAD "X16LIB.BIN",8,1,$A000
```

This will assume x16lib.bin is in the current directory and loads it into rambank 1. You can select any rambank you like.

After setting the appropriate variables (based on the function) the command: SYS $A000 will execute the function.

Notes for Assembly programmers
By including x16lib.inc into your project, you have direct access to all functions in the library. Before first use the library must be loaded to a rambank of choice. X16lib.inc contains a subroutine to

do so. You as a programmer have to make sure the rambank on which x16lib is loaded is active when calling a function from x16lib. All strings have to be terminated with a $0 byte.

PrintCenter

Prints a string in the center of the screen on a selected line.

BASIC

Function: XF$=”PRINTCENTER”

Variables: X1$ -> contains the text to print
X2% -> line number to print the text on

Example: /basic/print.bas or /basic/print.prg

Assembly

Function: jsr fPrintCenter

Setup: - Address of the symbol containing a string which and with a zero-byte in r0
- Line to print on in r1H

```
MyString:  .asciiz “this is my string”
lda #<MyString
sta r0
lda #>MyString
sta r0+1

lda #5
sta r1H

jsr fPrintCenter
```

Example: /asm/print.s

SwapVERALayers

Changes the z-order in VERA of layer0 and layer1.

BASIC

Function:	XF$=”SWAPLAYERS”
Variables:	{none}
Example:	/basic/swapveralayers.bas or /basic/swapveralayers.prg

Assembly

Function:	jsr fSwapVERALayers
Setup:	{none}
Example:	/asm/swaplayers.s

FadeOut

Fades the screen to black by decrementing the palette colors to zero.

BASIC

Function:	XF$=”FADEOUT”
Variables:	X1% -> fade speed (higher is slower). Typical use: 3
Example:	/basic/fade.bas or /basic/fade.prg

Assembly

Function:	jsr fFadeOut
Setup:	ldy #3 -> fade speed (higher is slower)
Example:	/asm/fade.s

FadeIn

Fades the screen from all black to normal operating mode.
NOTE: FadeOut must be executed before FadeIn. FadeOut makes a copy of the current palette which is used to restore the palette to by FadeIn.

BASIC

Function:	XF$=”FADEIN”
Variables:	X1% -> fade speed (higher is slower). Typical use: 3
Example:	/basic/fade.bas or /basic/fade.prg

Assembly

Function:	jsr fFadeIn
Setup:	ldy #3 -> fade speed (higher is slower)
Example:	/asm/fade.s

Highmem functions

The following two functions make using highram available to BASIC programs in a simple and fast way. With it you can save 255 bytes at a time to highram on a specified rambank with a specified identifier. And read back the data based on the specified identifier. The identifier is 1 to 4 chars long.

Allthough these function can be used from assembler, it is better (faster and probably easier) to write your own routines in asm.

Example for basic: /basic/highram.bas or /basic/highram.prg. For assembler: /asm/highmem.s

SaveToHighmem

Saves the content of a string to a specifief rambank with a specified identifier. If an identifier is used again for the same rambank, the value in the rambank will be overwritten.

BASIC

```
Function:        XF$="WRITE HIGHRAM"
Variables/setup:POKE $BFFF,5 -> use rambank 5
                 X1$ -> string to store in highmem
                 X2$ -> identifier/variable name
Returns:         PEEK($14) -> If not 0, then an error occurred:
                 -  0 -> no error
                 -  1 -> rambank is full
```

Assembly

```
Function:        jsr fSaveToHighmem
Setup:           r0 -> address of data to be written to highmem. The
                 data should end with a $0 byte.
                 r2 -> identifier/variable name
                 $BFFF contains rambank to  write to
Returns:         lda $14 -> if not zero, then an error occurred
```

ReadFromHighmem

Reads a value from highmem based on a identifier.

BASIC

```
Function:        XF$="READ HIGHRAM"
Variables/setup:POKE $BFFF,5 -> use rambank 5
                 X1$ -> must be set to the minimal length of the data
                 that is being returned. If unsure, set it to 255
                 bytes.
                 X2$ -> identifier/variable name
Returns:         X1$ -> the data that has been returned
```

Assembly

```
Function:        jsr fReadFromHighmem
Setup:           r0 -> address of memory allocation to where the data
                 is going to be written. There should be enough
                 continues memory to store the data in.
                 r2 -> identifier/variable name
                 $BFFF contains rambank to  write to
Returns:         The data will be returned to r0 with a $0 byte
                 terminating the string.
```

Trim

The Trim, RTrim and LTrim all work exactly the same way. It removes any spaces from the right-, left or right and left-part of a string.

BASIC

Function:	XF$=”RTRIM” (or LTRIM or TRIM)
Variables:	X1$ -> the string to be trimmed
Returns:	The trimmed string is returned in X1$
Example:	/basic/trim.bas or /basic/trim.prg

Assembly

Function:	jsr fTrim (or fLTrim or fRTrim)
Setup:	r0 -> address of the string to be trimmed, ending with a $0 byte.
Returns:	The string is trimmed in place, setting the $0 byte to the end of the trimmed string.
Example:	/asm/trim.s

Instr

Finds the first occurrence of a string within another string.

BASIC	
Function:	XF$=”INSTR”
Variables:	X1$ -> the string in which to search
	X2$ -> the string to search for
Returns:	PEEK($16) -> position of the string. 0 (zero) means found at first position. If $FF is returned, the string has not been found.
Example:	/basic/instr.bas or /basic/instr.prg

Assembly	
Function:	jsr fInstr
Setup:	r0 -> the string in which to search
	r2 -> the string to search for
Returns:	r10L -> the position found. If $FF is returned, the string has not been found.
Example:	/asm/instr.s

Split

Returns an item of a string which is separated by a specific char. You can think of the split function as returning an element of an array.

BASIC

Function:	XF$="SPLIT"
Variables:	X1$ -> the string which is separated by a char
	X2$ -> the separator (for example a comma)
	X1% -> the item number to be returned (starting at 0)
Returns:	X3$ -> the substring. The length of X3$ should be big enough to fit the result in.
Example:	/basic/split.bas or /basic/split.prg

Assembly

Function:	jsr fSplit
Setup:	r0 -> string to split
	r2 -> separator (1 byte)
	$BE00 -> index to retrieve
Returns:	r6 -> the substring. The length should be big enough to fit the result in.
Example:	/asm/split.s

Tally

Counts the times a specific string occurs within another string.

BASIC

Function:	XF$=”TALLY”
Variables:	X1$ -> string to search in X2$ -> string to search for
Returns:	PEEK($16) -> count of times X2$ occurs in X1$ (max 255)
Example:	/basic/tally.bas or /basic/tally.prg

Assembly

Function:	jsr fTally
Setup:	r0 -> address of string to search in r2 -> address of string to search for
Returns:	r10L -> count of times r2 occurs in r0 (max 255)

Keyboard handler

With the custom keyboard handler from x16lib you can check for each key it is down or not down. Also when multiple keys are down. First you have to initialize the custom keyboard handler, then you can do your code in which you check the keys, when you're finished (or quit your program) you should reset the custom keyboard handler, to restore it to the default one.

The custom keyboard handler uses address region $0400 - $494

BASIC example: /basic/keyboard.bas or /basic/keyboard.prg
Assembler example: /asm/keyboard.s

SetCustomKeyboardHandler

Replaces the default handler and initialize the $0400 memory area.

BASIC

Function:	XF$="INIT KBHANDLER"
Variables:	{none}
Returns:	{none}

Assembly

Function:	jsr fSetCustomKeyboardHandler
Setup:	{none}
Returns:	{none}

CheckKey

Checks if a specified key is down or not.

BASIC

Function:	XF$="CHECKKEY"
Variables:	X1% -> the keycode to check (0-128)
Returns:	X1% -> 1=down, 0=up

Assembly

Function:	jsr fSetCarryOnKeyDown
Setup:	ldx with the keycode to check
Returns:	Carry flag is set if key is down, not set if key is up.

ResetCustomKeyboardHandler

Resets the keyboard handler to its default.

BASIC

Function:	XF$="RESET KBHANDLER"
Variables:	{none}
Returns:	{none}

Assembly

Function:	jsr fResetCustomKeyboardHandler
Setup:	{none}
Returns:	{none}