

SymbOS C Compiler Manual

v1.2

Contents

Documentation contents	12
Introduction	13
Why C?	13
Features	14
Limitations	14
Using the compiler	16
Contents	16
Basic operation	16
Multi-file projects	16
Installation	17
Windows	17
Linux/MacOS	17
SymbOS	17
Compiler options	18
SymbOS executable options	18
Other options	18
Interfacing with assembly	19
Some advice on workflow	21
Testing on an emulator	21
Editors	21
Native compilation	22
SymbOS programming	23
Contents	23
Console applications	23
Windowed applications	24
Memory segments	24
Windows	25
Controls	27
Desktop commands	28
Handling events	29
Next steps	31
Advanced windows	31
Menus	31
Toolbars	32
Resizing calculations	32
Modal windows	34
Control reference	35
C_AREA	35
C_TEXT	35

C_TEXT_FONT	35
C_TEXT_CTRL	36
C_FRAME	37
C_TFRAME	37
C_GRID	38
C_PROGRESS	38
C_IMAGE	39
C_IMAGE_EXT	39
C_IMAGE_TRANS	39
C_ICON	39
C_BUTTON	40
C_CHECK	40
C_RADIO	41
C_HIDDEN	42
C_TABS	42
C_SLIDER	43
C_COLLECTION	43
C_INPUT	44
C_TEXTBOX	45
C_LISTBOX	47
C_LISTFULL	49
C_LISTTITLE	49
C_DROPDOWN	49
C_TREE	49
Event reference	52
MSR_DSK_WCLICK	52
MSR_DSK_WFOCUS	52
MSR_DSK_CFOCUS	52
MSR_DSK_WRESIZ	53
MSR_DSK_WSCROLL	53
MSR_DSK_MENCTX	53
MSR_DSK_EVTCLK	53
MSR_DSK_WMODAL	54
System call reference	55
Subsections	55
Related libraries	56
System variables	56
Kernel routines	58
Contents	58
Messaging	58
Msg_Send()	58
Msg_Receive()	58

Msg_Sleep()	58
Msg_Wait()	59
Idle()	59
Memory management	59
Mem_Reserve()	59
Mem_Release()	60
Mem_Resize()	60
Mem_Longest()	60
Mem_Free()	61
Mem_Banks()	61
Memory read/write	61
Bank_ReadWord()	61
Bank_WriteWord()	61
Bank_ReadByte()	61
Bank_WriteByte()	62
Bank_Copy()	62
Bank_Get()	62
Bank-Decompress()	62
System status	63
Sys_Counter()	63
Sys_Counter16()	63
Sys_IdleCount()	63
Shell functions	64
Contents	64
Interacting with SymShell	64
A note on exiting	65
Shell functions	65
Shell_CharIn()	65
Shell_CharOut()	65
Shell_CharTest()	65
Shell_StringIn()	66
Shell_StringOut()	66
Shell_Print()	66
Shell_Locate()	67
Shell_Exit()	67
Shell_PathAdd()	67
Shell_CharWatch()	68
Shell_StopWatch()	68
Shell control codes	68
Window routines	70
Contents	70
Window management	70
Win_Open()	70

Win_Close()	70
Win_Redraw()	70
Win_Redraw_Ext()	70
Win_Redraw_Area()	71
Win_Redraw_Toolbar()	71
Win_Redraw_Menu()	71
Win_Redraw_Title()	71
Win_Redraw_Status()	71
Win_Redraw_Slider()	72
Win_Redraw_Sub()	72
Win_ContentX()	72
Win_ContentY()	72
Win_Focus()	72
Win_Maximize()	73
Win_Minimize()	73
Win_Restore()	73
Win_Move()	73
Win_Resize()	73
Win_Width()	73
Win_Height()	74
Window status	74
Win_X()	74
Win_Y()	74
TextBox_Pos()	75
TextBox_Redraw()	75
TextBox_Select()	75
Desktop features	75
Contents	76
Popup dialogs	76
MsgBox()	76
InputBox()	77
FileBox()	77
Context menus	78
Menu_Context()	78
Rubber band select	78
Select_Pos()	78
Select_Size()	79
Clipboard	79
Clip_Put()	79
Clip_Get()	80
Clip_Type()	80
Clip_Len()	80
System tray	80
Systray_Add()	80
Systray_Remove()	81

Filesystem routines	82
Contents	82
File access	82
File_Open()	82
File_New()	82
File_Close()	83
File_Read()	83
File_ReadLine()	83
File_ReadComp()	83
File_Write()	84
File_Seek()	84
File_ErrMsg()	85
Directory access	85
Dir_Read()	85
Dir_ReadRaw()	86
Dir_ReadExt()	86
Dir_New()	86
Dir_Rename()	86
Dir_Move()	87
Dir_Delete()	87
Dir_DeleteDir()	87
Dir_GetAttrib()	87
Dir_SetAttrib()	87
Dir_GetTime()	88
Dir_SetTime()	88
Dir_PathAdd()	88
Multitasking routines	90
Processes	90
App_Run()	90
App_End()	90
App_Search()	91
App_Service()	91
App_Release()	92
Proc_Add()	92
Proc_Delete()	93
Proc_Sleep()	93
Proc_Wake()	93
Proc_Priority()	94
Timers	94
Counter_Add()	94
Counter_Delete()	94
Counter_Clear()	95
Timer_Add()	95
Timer_Wake()	95

Timer_Delete()	96
Multithreading	96
thread_start()	96
thread_quit()	97
Thread safety	97
Thread-safe messaging	97
Device routines	98
Contents	98
Screen status	98
Screen_Mode()	98
Screen_Mode_Set()	98
Screen_Colors()	99
Screen_Width()	99
Screen_Height()	99
Screen_Redraw()	99
Color_Get()	99
Color_Set()	100
Text_Width()	100
Text_Height()	100
Mouse status	100
Mouse_X()	100
Mouse_Y()	100
Mouse_Buttons()	101
Mouse_Dragging()	101
Keyboard status	101
Key_Down()	101
Key_Status()	101
Key_Put()	102
Key_Multi()	102
Time functions	102
Time_Get()	102
Time_Set()	103
Time2Obj()	103
Obj2Time()	103
System configuration	103
Sys_Path()	103
Sys_Type()	103
Sys_GetDrives()	104
Sys_DriveInfo()	104
Sys_DriveFree()	105
Sys_GetConfig()	105
Sound	106
Contents	106

Creating/getting sounds	106
Sound functions	106
Sound_Init()	106
Music_Load()	107
Music_Load_Mem()	107
Music_Free()	107
Music_Start()	107
Music_Stop()	108
Music_Continue()	108
Music_Volume()	108
Effect_Load()	108
Effect_Load_Mem()	108
Effect_Free()	109
Effect_Play()	109
Effect_Stop()	109
Printer routines	110
Contents	110
Printing via the daemon	110
Printing via PrintIt	111
Direct printer functions	111
Print_Busy()	111
Print_Char()	112
Print_String()	112
Reference tables	113
Contents	113
Keyboard scancodes	113
Keyboard ASCII codes	113
Colors	114
Error codes	115
Graphics Library	117
Contents	117
Using the library	117
Using canvases	118
Creating canvases	118
Initializing canvases	118
Refreshing the display	119
Drawing functions	120
Gfx_Pixel()	120
Gfx_Safe_Pixel()	120
Gfx_Value()	120
Gfx_Line()	120
Gfx_LineB()	120

Gfx_HLine()	120
Gfx_VLine()	121
Gfx_Box()	121
Gfx_BoxF()	121
Gfx_Circle()	121
Gfx_Text()	121
Gfx_ScrollX()	122
Gfx_ScrollY()	122
Gfx_Clear()	122
Sprite functions	123
Converting images	123
Image sets and masks	123
Gfx_Load()	124
Gfx_Load_Set()	124
Gfx_Put()	125
Gfx_Put_Set()	125
Gfx_Get()	125
Gfx_Save()	125
Gfx_Prep()	126
Gfx_Prep_Set()	126
Gfx_TileAddr()	126
Raw images	126
Raw image controls	127
Gfx_Load_Raw()	127
Gfx_Load_Set_Raw()	128
Gfx_Prep_Raw()	128
Gfx_Prep_Set_Raw()	128
Gfx_TileAddr_Raw()	128
Advanced topics	128
Moving sprites	128
Memory problems	129
Reference	130
Color palette	130
Network Library	132
Contents	132
Using the library	132
Net_Init()	132
HTTP functions	133
HTTP_GET()	133
HTTP_POST()	133
Proxy servers	134
Tracking progress	134
DNS functions	134
DNS_Resolve()	134

DNS_Verify()	135
TCP functions	135
TCP_OpenClient()	135
TCP_OpenServer()	136
TCP_Event()	136
TCP_Status()	136
TCP_Send()	137
TCP_Receive()	137
TCP_ReceiveToEnd()	138
TCP_Skip()	138
TCP_Flush()	138
TCP_Disconnect()	139
TCP_Close()	139
UDP functions	139
UDP_Open()	139
UDP_Event()	140
UDP_Status()	140
UDP_Send()	140
UDP_Receive()	140
UDP_Skip()	141
UDP_Close()	141
FTP functions	141
FTP_Open()	141
FTP_Upload()	142
FTP_Download()	142
FTP_Listing()	143
FTP_ChDir()	143
FTP_Command()	144
FTP_Response()	144
FTP_GetPassive()	144
FTP_Disconnect()	145
FTP_Close()	145
Helper functions	145
Net_ErrMsg()	145
Net_SplitURL()	145
Net_PublicIP()	145
Net_SkipMsg()	146
iptoat()	146
Reference	146
Error codes	146
Special considerations	148
Contents	148
The malloc() heap	148
Native preprocessor	148

Quirks of <code>stdio</code>	149
File sizes	149
<code>fseek()</code>	149
<code>printf()</code>	149
Using <code>as</code> as a standalone assembler	149
Building SCC	150
SCC versus SDCC	150
Porting tips	151
Contents	151
Common problems	151
“Unknown symbol” errors on compilation	151
“Out of memory” error when starting the compiled app	151
Problems with <code>malloc()</code>	152
<code>printf()</code> formatting looks wrong	152
Can’t <code>fopen()</code> a file that should be openable	152
<code>fread()</code> etc. won’t read the end of a file (or reads extra junk at the end of a file)	152

Documentation contents

- [Introduction](#) (this page)
- [Using the compiler](#)
- [SymbOS programming guide](#)
 - **Start here** for a quickstart guide to writing your first SymbOS app (both console apps to run in SymShell and windowed apps to run on the desktop).
- [System call reference](#)
- [Control reference](#)
- [Event reference](#)
- [Graphics library](#)
- [Network library](#)
- [Special considerations](#) (compiler quirks, etc.)
- [Porting tips/troubleshooting](#)

Introduction

SCC is an ANSI C compiler that produces executables for [SymbOS](#).

SCC is not the only way to write code for SymbOS; for a more Visual Basic-like experience with a GUI form editor and event-driven programming (albeit with sparse documentation and a lot of quirks), check out the [Quigs IDE](#). There is also [extensive documentation and example code](#) for writing SymbOS applications in pure Z80 assembler. But if you already know C and want to write SymbOS software, or you want to port existing C code to SymbOS, or you just want something more powerful than Quigs but less mind-numbing than assembly, read on.

(This documentation assumes you are familiar with standard C syntax, particularly structs, pointers, and typecasting, as well as the various weird ways you can combine them.)

Why C?

Writing C will rot your brain. Only a true C programmer would look at the following code and think it was an “elegant” solution to anything:

```
addr = ((_MemStruct)membank).ptr->addr & (ver > 4 ? 7 : attr);
*((char*)(++addr)) += 'A';
```

C is an awful language that is nonetheless ideal for many kinds of 8-bit programming, where we want to express ideas in a relatively high-level way while retaining the knowledge of where every byte is going and why. C is particularly good at dealing with byte-level structured data and pointers, which makes it a good fit for SymbOS. For example, SymbOS GUI controls are defined by exact blocks of bytes laid out in the correct order. These can be conveniently represented in C as structs, allowing us to reference the control’s properties by name:

```
typedef struct {
    char* text;           // address of text buffer
    unsigned short scroll; // scroll position, in bytes
    unsigned short cursor; // cursor position, in bytes
    signed short selection; // number of selected characters relative to cursor
    unsigned short len;    // current text length, in bytes
    unsigned short maxlen; // maximum text length, in bytes
    unsigned char flags;   // flags
    unsigned char textcolor; // text color
    unsigned char linecolor; // line color
} Ctrl_Input;
```

With this type of infrastructure in place, we can express complex ideas in just a few lines of code:

```
// declare a 256-byte static buffer in the SymbOS "data" segment
_data char buffer[256];

// subroutine: convert the text in the specified GUI textbox (passed by reference)
// to a number, then AND it with every byte in buffer[].
```

```
void filter_buffer(Ctrl_Input* textbox) {  
    int i = 0;  
    char mask = atoi(textbox->text);  
    while (i < sizeof(buffer)) {  
        buffer[i] &= mask;  
        i++;  
    }  
}
```

C is also a mature language with a vast amount of open-source code written in it, so there is ample material for porting or adapting and it is possible to find code examples for just about every algorithm imaginable.

Features

- Runs natively on Windows, Linux, and SymbOS.
- Full build chain with preprocessor, object files, linker, etc.
- Standard ANSI C syntax, with good support for most typical usage.
- Standard data types and structures: `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, `struct`, `union`, `enum`, `auto`, `static`, `register`, `extern`, `typedef`.
- A proper `libc` port so existing code can be compiled unmodified.
- Headers, `typedefs`, and support functions for most SymbOS system calls (`symbos.h`).
- Clean handling of SymbOS segments, with data, buffers, and literals located correctly in the executable without duplication (other compilers struggle with this). Keywords `_data` and `_transfer` allow specifying the segment of globals.
- Multithreading (yes, [really!](#))

Limitations

- **Not all `libc` functions are available, well-tested, and/or correctly implemented yet.** The `libc` implementation is robust enough for SCC to compile itself, but you may encounter subtle incompatibilities and any code requiring less-common system functions should be tested carefully.
- Symbol matching only considers the first 15 characters (to conserve memory on native builds).
- No high-level optimizations like subexpression merging. (Just write efficiently.)
- The `libc` implementation is not very fast and favors portability over speed.
- For the usual Z80 reasons, 8-bit arithmetic (`char`) will always be much faster than 16-bit (`int`) and particularly floating-point (`float`, `double`) arithmetic. Declaring variables `unsigned` may also improve efficiency where applicable.
- `double` is currently treated as synonymous with `float` (32-bit IEEE 754 floating-point).
- The usual SymbOS limits apply, such as 64KB and 16KB limits on particular segments (see the [SymbOS programming guide](#)).
- No named static struct initializers like `{.x = 1, .y = 2}`—use `{1, 2}` instead and just be careful about member order.
- No K&R-style function declarations.

- Not all C99 and C11 additions are supported.
- `const` and `volatile` are accepted but don't actually do anything.
- Local variables are technically scoped to their function, not their block.
- If you run into type-mismatch problems with unusual pointer data types (like struct members that are typed as pointers to another typedef'd struct—this can happen when working with complex data structures for windows), try `void*` instead.

Using the compiler

Contents

- [Basic operation](#)
- [Multi-file projects](#)
- [Installation](#)
 - [Windows](#)
 - [Linux/MacOS](#)
 - [SymbOS](#)
- [Compiler options](#)
 - [SymbOS executable options](#)
 - [Other options](#)
- [Interfacing with assembly](#)
- [Some advice on workflow](#)

Basic operation

To compile a single source file into a single SymbOS executable, just run it through `cc` on the command line:

```
cc windemo.c
```

That's it! Enjoy your executable.

Multi-file projects

For more complex projects, it is helpful to understand how the C build chain works. Under the hood, SCC consists of a multi-stage toolchain typical for C compilers: a preprocessor (`cpp`), multiple compiler stages (`cc0`, `cc1`, and `cc2`), an assembler (`as`), linker (`ld`), optimizer (`copt`), and a relocation table builder (`reloc`). Generally speaking, after preprocessing, source files (`.c`) are compiled to assembly files (`.s`), which are then assembled into object files (`.o`). The linker (`ld`) then links these object files together with each other (and with the relevant functions from the standard library) to produce a single executable.

For organization and to improve compilation speed (particularly when compiling natively on SymbOS), projects can be broken up into multiple source files (`.c` or `.s` for assembly-language source files). If we pass multiple source files to `cc`, it will compile or assemble them into object files, then link the resulting object files into an executable:

```
cc file1.c file2.c asmfile.s
```

However, we can also skip the linking step by using the `-c` command-line option. This lets us only recompile the specific modules we have changed, saving a lot of time when building large projects natively on SymbOS. For example, to compile just the single source file `file1.c` into the object file `file1.o`:

```
cc -c file1.c
```


Then, to link all the relevant object files into a single executable:

```
cc -o file.exe file1.o file2.o asmfile.o
```

As seen in the above example, the `-o` option can be used to manually specify what the resulting executable should be named (e.g., `file.exe`).

In the C world this type of modular build is usually done with a Makefile. SCC does not currently have its own `make` utility, but we can use the one from MinGW (not documented here). In practice SymbOS projects are usually small enough that, for normal desktop cross-compilation, we can just maintain a single main source file (potentially with `#include` directives to merge in subsidiary files) and compile it directly with `cc`.

The `cc` app is usually the most convenient way to organize modular builds, but we can also run the stages separately if we know what we are doing. (A good way to determine what `cc` is doing under the hood is to run it with the `-v` option, which outputs each subcommand as it is run.)

Installation

Windows

To install on Windows, just download the latest [binary release](#) for Windows and unzip it to a convenient folder. SCC has been tested to work on Windows 98SE through Windows 11.

Linux/MacOS

Using SCC on Linux is possible, but currently requires building from source:

1. Download the latest [source release](#) and unzip it to a convenient folder.
2. Ensure you have `gcc` installed.
3. On the command line, navigate to the `src` subfolder and run `./makeLinux.sh`. (It may be necessary to run `chmod +x makeLinux.sh` first.)

This process currently builds SCC in-place (in the source repository's `bin` subfolder) rather than installing to some universally accessible directory like `/usr/bin`. Once built, `cc` expects to be run directly from this `bin` subfolder, or it will not be able to find its own libraries.

SymbOS

SCC can run natively on SymbOS! This is cool, but much too slow for serious development—expect multiple minutes to compile a simple “hello world” program. For this reason, it’s currently recommended to [set up a good cross-compilation environment](#) rather than trying to do all your development on SymbOS. But if you insist, see [below](#) for some suggestions.

To install on SymbOS, just download the latest [binary release](#) for SymbOS, unzip it, and copy the entire directory tree onto a sufficiently large FAT-formatted drive (such as a mass storage device). Once installed, you can just run `cc` from SymShell like you would on other platforms.

Warning: The directory structure must remain exactly as it is in the archive! SCC will not run if all the files are placed in a single folder.

Warning: SCC **must** be run from a FAT-formatted drive! The filesystem used on CP/M and Amstrad floppies internally pads files to the nearest 128 bytes, which will cause problems for utilities (like `ld`) that need to know the length of binary files to the exact byte. This also means that transferring the object and library files in the `lib` subfolder (`.o`, `.a`) via an AMSDOS floppy or floppy image **will corrupt them**. If there is no other way to transfer these files to their final FAT-formatted drive, a good workaround is to compress them into a `.zip` file and decompress them at their final destination using the SymbOS `unzip.com` utility.

It is recommended to install SCC to a relatively “root” subfolder (such as `C:\SCC`) because `cc.com` will use this absolute path repeatedly in long SymShell commands, which may overflow if the base path is very long. To allow CC to be run directly in SymShell without specifying its full path, the base SCC folder can be the SymbOS system folder (e.g., `C:\SYMBOS`, so the path of `cc` is `C:\SYMBOS\CC.COM` and SCC’s libraries are in subfolders like `C:\SYMBOS\LIB`).

Compiler options

SymbOS executable options

SymbOS executables include several special resources used by the desktop. These can be specified using command-line arguments passed to `cc`:

- `-N "appname"` - specifies the application name shown in the task manager.
- `'-G iconfile` - specifies the 4-color application icon (see below).
- `'-g iconfile` - specifies the 16-color application icon (see below).

For example:

```
cc file.c -N "Application Name" -G icon4.sgx -g icon16.sgx
```

Icons are 24x24 images in `.sgx` format. The default icon resembles the SymShell icon and is suitable for console apps, but we can create our own using software such as [MSX Viewer 5](#) (classic version) or the `gfx2sgx` tool included with the [graphics library](#). For example, to convert a 24x24-pixel image called `icon4.png` into a 4-color `.sgx` icon:

```
gfx2sgx icon4.png -4
```

A library of generic icons can also be found on the [SymbOS website](#).

Note: MSX Viewer 5 generates an incorrect header for 16-color icons. This can be fixed easily with a hex editor by deleting the first eight bytes of the file and replacing them with the ten bytes: `0C 18 18 00 00 00 00 20 01 05`. The resulting file should be exactly 298 bytes long. 4-color icons generated with MSX Viewer 5 will work unmodified.

Other options

- `-c` - compile to object modules only (`.o` files) without linking; see [Multi-File Projects](#) for details.
- `-D x` - define the macro `x` for the preprocessor; e.g., `-D TESTBUILD` will act like the preprocessor directive `#define TESTBUILD`.
- `-E` - preprocess only, do not compile. Output will be saved to the temporary file `$stream0.c`.

- `-I x` - adds a directory to the include path; e.g., `-I C:\MYPROJ` will add `C:\MYPROJ` to the list of directories searched when resolving an `#include` directive.
- `-h x` - set the `malloc()` heap size to `x` bytes (default 4096); see [here](#) for discussion.
- `-l x` - link against the library `x` (physically stored as the file `libx.a` in SCC's `lib` subfolder); e.g., `-lnet` links against `libnet.a`, giving us access to functions in the [network library](#).
- `-L x` - adds a directory to the library path; e.g., `-L C:\MYPROJ` will add `C:\MYPROJ` to the list of directories searched when resolving a `-l` command-line option.
- `-M x` - create a map file showing the relative memory addresses of each exported symbol (variable or function); e.g., `-M mapfile.txt` will create a map file called `mapfile.txt`. For more detailed information, the utility `sortmap.exe` in SCC's `bin` subfolder can be used to sort this map and show the amount of memory used by each symbol; e.g., `sortmap mapfile.txt`.
- `-o x` - set the output filename of the SymbOS executable to `x`; e.g., `-o file.com` would name the output file `file.com`.
- `-Ox` - set the optimization level, where `x` is one of `0`, `1`, `2`, or `s`. This controls whether various short operations will be inlined (for speed) or delegated to subroutines (for size). Not all optimization levels are currently stable, so only `-O1` (the default) or `-Os` (optimize for size) are currently recommended.
- `-s` - build standalone, without including `libc` or system libraries. For experts only.
- `-S` - compile to Z80 assembly source only (`.s` files); do not assemble.
- `-T x` - set the assembler name of the **code** segment to `x`. For experts only.
- `-v` - verbosely print the full command being run for each compiler pass.
- `-X` - do not erase temporary files (for example, generated Z80 assembly files). Mainly useful for low-level debugging.

Interfacing with assembly

SCC does not currently support inline assembly. However, Z80 assembly files can be passed as arguments to `cc` to be linked into the main executable. For example:

```
cc cfile.c asmfile.s
```

The usual system of `.export` and `extern` applies for sharing symbols between assembly and C objects. Assembly files must export any shared symbols with the `.export` directive:

```
.code                ; emit to code segment
.export _asmadd      ; export symbol _asmadd
_asmadd:
    pop de           ; pop return address
    pop hl           ; pop first argument (8-bit, so in L)
    ld a,(_asmbuf+0)
    add l
    ld l,a           ; 8-bit return value goes in L
    push hl          ; restore stack (caller cleans up)
    push de          ; restore return address
    ret
```

```
.symtrans          ; emit to transfer segment
.export _asmbuf    ; export symbol _asmbuf
_asmbuf:
    .ds 256        ; 256 bytes of 0x00
```

The exported symbols can then be declared in C with the `extern` keyword:

```
extern char asmbuf[256];
extern char asmadd(char v);

int main(int argc, char* argv[]) {
    asmbuf[0] = 1;
    asmbuf[1] = asmadd(2); // asmbuf[1] will contain 2 + 1 = 3
}
```

The assembled version of shared symbols is assumed to start with an underscore, so the C symbol `main` assembles to `_main` and the assembly symbol `_asmfunc` is referenced in C as `asmfunc`. If a symbol is not exported, it remains local to its own assembly file.

SCC uses an approximately cdecl calling convention: all arguments are passed on the stack, right to left, before the function is called with `call`. On function entry, the top word of the stack will be the return address, followed by the leftmost argument, the next leftmost argument, and so on. The caller cleans up the stack, so the stack should be returned to this state (at least in quantity, if not in content) before calling `ret`. 8-bit values are passed as the low byte of the 16-bit value pushed to stack. 32-bit values are passed as two successive 16-bit values on the stack. 8-bit values are returned in the L register. 16-bit values are returned in the HL register pair. 32-bit values are returned with the low word in HL and the high word in (`__hireg`), a globally defined symbol in `libz80.a` that will be linked into all SymbOS executables.

Note that assembly routines should preserve the values of the IX, IY, and (ideally) BC registers on entry/exit. SCC uses IX and/or IY to track the local variables of the calling function, and may use BC as a register variable. (If no register variables are used, it is safe to destroy BC.)

as supports standard Z80 opcodes, with notable directives including:

- `.code`: output subsequent code to the SymbOS **code** segment
- `.symdata`: output subsequent code to the SymbOS **data** segment
- `.symtrans`: output subsequent code to the SymbOS **transfer** segment
- `.abs`: output subsequent code at the absolute address specified by `.org` (this may not link correctly into a SymbOS executable)
- `.org ____`: set absolute address of subsequent code to _____. **Must be preceded by .abs!**
- `.export ____`: export symbol ____ for linking
- `.byte ____`: emit the raw byte ____ (also `.db` or `db` or `defb`). Multiple values can be separated by commas.
- `.word ____`: emit the raw 2-byte word ____ (also `.dw` or `dw` or `defw`). Multiple values can be separated by commas.
- `.ds ____`: emit ____ bytes filled with 0x00 (also `ds` or `defs` or `.blkb`)

- `.ascii "____"`: emit the text string "____" as raw ASCII text (also `defm`). Single quotes `'` can also be used to delimit the string if the string includes double quotes.

In assembly generated during compilation, you may see additional segment directives like `.data` and `.bss` (the internal “data” and “bss” segments, actually linked as part of the SymbOS **code** segment) or `.literal` (the internal “literal” segment, actually linked as part of the SymbOS **data** segment). **Do not confuse `.data` with `.symdata`!** To put data in the SymbOS **data** segment, use `.symdata`, not `.data`.

Note that `as` uses assembly syntax similar to—but simpler and not 100% identical with—the Maxam-style syntax supported by the WinApe assembler (the most common assembler for SymbOS programming). If something isn’t working as expected, try examining some of the assembler files in the SCC source repository to see what syntax they use. To see what assembly code `cc` is producing, run it with the `-x` option to preserve intermediate files.

Some advice on workflow

Development will be much, **much** easier if you cross-compile on a PC and test on an emulator. Working natively on SymbOS sounds cool—and improving the native experience is a long-term goal—but for now, the slow compilation speeds and lack of a good IDE quickly get frustrating. Setting up an efficient cross-compilation workflow will dramatically improve your experience with SCC.

Testing on an emulator

Some example workflows using [WinApe](#) on Windows:

1. Easy: Run SymbOS in WinApe (e.g., from a floppy or hard disk image). Insert a blank, formatted disk image into one of WinApe’s virtual floppy drives, and keep the “Edit Disk” window open. Whenever you compile a new version of your app, just drag-and-drop it into the Edit Disk window and run it from SymbOS.
2. Advanced: Create a FAT16 or FAT32 partition on your hard drive and mount it as a hard drive within WinApe (Settings > Other > Logical Drive). Unpack the SymbOS installation packages to this partition and install the SymbOS CPC ROM images to slots Upper 1/2/3/4 in WinApe. You should now be able to run SymbOS directly off of your hard drive, with both Windows and SymbOS seeing the same files simultaneously. Just compile on Windows and run the resulting executable in-place on SymbOS!

Similar workflows are possible with other emulators, depending on setup. For developing [network](#) apps, note especially that [CPCEMU](#) can emulate the M4 Board’s WiFi capabilities.

Editors

SCC code tends to use a lot of SymbOS-specific system calls, so it’s worth using a good modern IDE with code-completion—it will drastically reduce the number of times you have to look up syntax in the manual. ([Code::Blocks](#) and Visual Studio Code are both popular.) In Code::Blocks, you should add SCC’s `lib/include` directory to the search path to enable code-completion for SCC-specific functions (Settings > Compiler > Search directories > Compiler > Add), or just keep a copy of `symbos.h` open.

Native compilation

If you insist on running SCC natively on SymbOS, a few tips to improve the experience:

Run `cc` with the `-v` option. (e.g.: `cc -v test.c`) This shows the commands used for each compilation stage as they are executed, giving some sense of progress. (Otherwise, `cc` will just appear to do nothing for multiple minutes—pretty boring.)

Break large programs into small modules and only recompile the ones that have changed. See [above](#) for discussion of how to do this.

Only include the minimum number of necessary header files in each module (e.g., `symbos/shell.h` instead of the entire `symbos.h`). This may also be necessary to prevent SCC from running out of memory.

Run SCC in an accelerated platform/emulator. Compilation is much faster on SymbOSVM, although there are currently still bottlenecks relating to disk access that should be improved in future versions. Emulators like WinApe also allow ramping the CPU speed from 100% (Shift+F4) to 1000% (Shift+F5). (Whether this defeats the whole purpose of running SCC natively on SymbOS is, of course, a matter of taste.)

Create batch files to automate the build. SCC currently lacks a native “make” utility, but SymShell does support simple batch files. For example, we can create a file called `make.bat` in our project’s folder that contains the commands necessary to build the app, one per line. Typing `make` in the project’s folder will then run the commands in order from the batch file.

SymbOS programming

Start here if you are new to SymbOS programming. This section provides a complete guide to writing your first console and windowed SymbOS applications.

Contents

- [Console applications](#)
- [Windowed applications](#)
 - [Memory segments](#)
 - [Windows](#)
 - [Controls](#)
 - [Desktop commands](#)
 - [Handling events](#)
- [Next steps](#)
- [Advanced windows](#)
 - [Menus](#)
 - [Toolbars](#)
 - [Resizing calculations](#)
 - [Modal windows](#)

See also:

- [Control reference](#)
- [Event reference](#)
- [System call reference](#)

Console applications

For the most part, console applications meant to run in SymShell can be written in normal C style using the functions in `stdio.h` (`printf()`, `fgets()`, etc.). No additional headers are necessary:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello world!\n");
}
```

The only subtlety is that SymbOS expects console applications to have the file extension `.com` rather than `.exe`. We can rename the executable file after compilation, or tell `cc` to use a custom output name:

```
cc -o condemo.com condemo.c
```

While SCC's `stdio` functions are meant to work relatively seamlessly in SymShell, they are somewhat slow and bulky and are not entirely optimized for SymShell's display model. When writing code from scratch for SymbOS, consider designing it around the [system shell functions](#) (`Shell_StringOut()`, `Shell_CharIn()`, etc.) rather than the standard `stdio` functions.

Windowed applications

Writing windowed applications in SCC is not inherently difficult, but requires a thorough knowledge of SymbOS's desktop model. SCC provides the necessary headers and data types to interact with the SymbOS desktop manager, but there are no shortcuts here; writing a windowed application requires careful coding, testing, and regularly cross-referencing the documentation to ensure that every byte is where it needs to be. The most important data structures and functions are summarized here, but you may also find it useful to consult the [SymbOS developer documentation](#).

This section has a lot to take in, but don't worry! To keep you from getting overwhelmed, a good place to start is to try compiling `windemo.c` (in the `sample` folder). Then, read through this section with the goal of learning what each line in `windemo.c` actually does. Once you understand it, you can use the code in `windemo.c` as a skeleton for building your own graphical application, referring to this documentation as needed to see how SymbOS handles the controls and events you want to add. Once you understand the basic concepts of segments, data structures, and messaging, writing windowed applications isn't actually that hard.

(If you want a more Visual Basic-like experience with a graphical form editor, try the useful—if quirky—[Quigs](#) language.)

The first thing `windemo.c` does is include the header file `symbos.h`, which defines many of the features referenced in the following sections:

```
#include <symbos.h>
```

Memory segments

A SymbOS executable is divided into three memory segments. For console apps written with normal libc functionality, it is not generally necessary to think about this, but many SymbOS system calls make important distinctions between memory segments:

1. **Code:** Contains code and globals (unless manually placed into a different segment—see below). Can contain up to 64KB of code and data.
2. **Data:** Stores data that must not cross a 16KB page boundary. This is required by many system functions, particularly those dealing with text or image data, so SCC places string literals in this segment by default. Can contain up to 16KB of data.
3. **Transfer:** Stores data that must be relocated to the upper 16KB of address space so it can be accessed by (mainly) the desktop manager. Most data structures relating to desktop windows and controls should be placed here. Can contain up to 16KB of data.

Globals are placed in the **code** segment by default, but we can manually assign a global to the **data** or **transfer** segments by adding `_data` or `_transfer` to its declaration:

```
_transfer char imgdata[256];
```

Many system calls also make important distinctions between memory *banks*. Because the Z80 can only address 64KB of memory at a time, extended memory (up to 1MB) is divided into multiple “banks” of 64KB each. We do not generally need to worry about this because all three segments of the executable will be loaded into the same bank. However, there are two exceptions:

1. Some system calls will need to know which bank an address is in. In most cases, this is just the application's main bank (stored in the `_symbank` global).
2. If we need to handle more than 64KB of data in our application, it is possible to [reserve and indirectly address](#) memory in other banks.

Windows

SymbOS defines windows and controls using carefully structured data, which must typically be located in the **transfer** segment. To make it easier to define and reference these data structures, `symbos.h` defines a set of struct types that give convenient member names to the different components. Defining a window consists of carefully defining and arranging the necessary structs in the **transfer** segment.

The format of a window's primary data structure is as follows (see also `symbos.h`):

```
typedef struct {
    unsigned char state;      // (see below)
    unsigned short flags;    // (see below)
    unsigned char pid;       // process ID of owner (set by Win_Open)
    unsigned short x;        // window X position
    unsigned short y;        // window Y position
    unsigned short w;        // window width
    unsigned short h;        // window height
    unsigned short xscroll;  // X offset of window content
    unsigned short yscroll;  // Y offset of window content
    unsigned short wfull;    // full width of window content
    unsigned short hfull;    // full height of window content
    unsigned short wmin;     // minimum width of window
    unsigned short hmin;     // minimum height of window
    unsigned short wmax;     // maximum width of window
    unsigned short hmax;     // maximum height of window
    char* icon;              // address of the 8x8 window icon, in SGX format
    char* title;             // address of the title text
    char* status;            // address of the status text
    void* menu;              // address of the Menu struct (see Menus)
    void* controls;          // address of the control group struct
    void* toolbar;           // address of the control group struct for the toolbar
    unsigned short toolbarheight; // toolbar height, in pixels
    char reserved1[9];
    unsigned char modal;     // modal window ID + 1
    char reserved2[140];
} Window;
```

Some commentary on these elements is useful.

`state` is one of the following: `WIN_CLOSED`, `WIN_NORMAL`, `WIN_MAXIMIZED`, or `WIN_MINIMIZED`.

`flags` is an OR'd bitmask which may contain one or more of the following flags:

- WIN_ICON = show window icon
- WIN_RESIZABLE = window can be resized
- WIN_CLOSE = show close button
- WIN_TOOLBAR = show toolbar
- WIN_TITLE = show titlebar
- WIN_MENU = show menubar
- WIN_STATUS = show statusbar
- WIN_ADJUSTX = disable horizontal slider in resizable windows (i.e., the program will adjust its own X scroll, not the system)
- WIN_ADJUSTY = disable vertical slider in resizable windows (i.e., the program will adjust its own Y scroll, not the system)
- WIN_NOTTASKBAR = do not display in taskbar
- WIN_NOTMOVEABLE = window cannot be moved
- WIN_MODAL = modal window ([see below](#))

For windows that have the WIN_RESIZABLE flag, there is an important distinction between the size of the *window* (given by `.w` and `.h`) and the size of its *content* (given by `.wfull` and `.hfull`). The window will have scroll bars at the edges, and if the size of the window is smaller than the size of the content, the user will be able to use these scroll bars to move around and view the whole content of the window. This can be used to accomplish a variety of interesting techniques by adjusting the content of the window in response to the user scrolling or resizing the window; see [resizing calculations](#) for some discussion of this.

The window icon is an 8x8 4-color SGX image. The `.icon` member can be set to 0 if there is no icon, but this will also prevent the window from being displayed correctly in the taskbar. Images can be converted to SGX format using software such as [MSX Viewer 5](#) (classic version) or the `gfx2sgx` tool shipped as part of SCC's [graphics library](#), and their raw data examined using a hex editor. A simple default icon is:

```
_transfer char icon[19] = {0x02, 0x08, 0x08, 0xFF, 0xFF, 0xF8, 0xF1, 0xF8, 0xF1,
                          0x8F, 0x1F, 0x8F, 0x1F, 0x8F, 0x1F, 0x8F, 0x1F, 0xFF, 0xFF};
```

The content of the window data structure can be defined using a static initializer. Note that SCC does not currently support named initializers such as `.x = 10`, so we will instead need to list all necessary elements in order, being careful not to omit any intervening elements. Here's an example window definition:

```
_transfer Window form1 = {
    WIN_NORMAL,
    WIN_CLOSE | WIN_TITLE | WIN_ICON,
    0,          // PID
    10, 10,     // x, y
    178, 110,   // w, h
    0, 0,       // xscroll, yscroll
    178, 110,   // wfull, hfull
    178, 110,   // wmin, hmin
    178, 110,   // wmax, hmax
```

```

icon,      // icon
"Form 1",  // title text
0,         // no status text
0,         // no menu
&ctrls};  // controls

```

(The line `&ctrls` will be explained in the next section.)

Controls

Window controls (buttons, text, etc.) are defined using `Ctrl` structs, which must themselves be linked to an overarching `Ctrl_Group` struct. The definition of a `Ctrl_Group` struct is:

```

typedef struct {
    unsigned char controls; // total number of controls in the group
    unsigned char pid;      // process ID of owner (set by Win_Open for main group)
    void* first;            // address of first control's data
    void* calcrule;         // address of first resizing calculation (see below), or 0 for none
    unsigned short unused1;
    unsigned char retctrl;  // control ID to "click" on pressing Return
    unsigned char escctrl;  // control ID to "click" on pressing Escape
    char reserved1[4];
    unsigned char focusctrl; // control ID to select on window focus
    unsigned char reserved2;
} Ctrl_Group;

```

Each control is an instance of the struct `Ctrl`, which has the definition:

```

typedef struct {
    unsigned short value; // control ID or return value
    unsigned char type;   // control type
    char bank;            // bank of extended data record (-1 for default)
    unsigned short param; // parameter, or address of extended data record
    unsigned short x;     // X location relative to window content
    unsigned short y;     // Y location relative to window content
    unsigned short w;     // width in pixels
    unsigned short h;     // height in pixels
    unsigned short unused;
} Ctrl;

```

The controls in a control group must be defined in order immediately after each other, with the encompassing `Ctrl_Group`.`first` pointing to the address of the first control. (This will also be the order in which they are drawn onscreen.)

The member `.param` requires some explanation. For the simplest type of control, it is simply a 16-bit number containing various values and flags required by the control. For instance, an important control type is `C_AREA`, which fills the control's area with a solid color; in this case `.param` is the color code (e.g., `COLOR_BLACK`, `COLOR_RED`, `COLOR_ORANGE` or `COLOR_YELLOW` in the default Amstrad CPC 4-color mode). This

control is very important because SymbOS will not fill the window background automatically! Starting the main control group with a C_AREA control ensures that the window background is filled in.

For more complicated control types, .param holds the address of the control's extended data record. (Note that we will need to cast this address to (unsigned short) or a compatible type for SCC to accept it, since SCC does not normally allow assigning a pointer directly to a scalar variable.) Each control type has a different extended data record format. For example, to declare a text label, we would use Ctrl_Text:

```
typedef struct {
    char* text;           // address of text string
    unsigned char color; // (foreground << 2) | background
    unsigned char flags; // ALIGN_LEFT, ALIGN_CENTER, or ALIGN_RIGHT
} Ctrl_Text;
```

Combining this into a set of controls for our example window:

```
// extended data record for c_text1
_transfer Ctrl_Text cd_text1 = {
    "Hello world!",           // text
    (COLOR_BLACK << 2) | COLOR_ORANGE, // color
    ALIGN_LEFT};             // flags

// two controls, immediately after each other
_transfer Ctrl c_area = {
    0, C_AREA, -1,           // control ID, type, bank
    COLOR_ORANGE,           // param (color)
    0, 0,                   // x, y
    10000, 10000};          // width, height
_transfer Ctrl c_text1 = {
    0, C_TEXT, -1,           // control ID, type, bank
    (unsigned short)&cd_text1, // param (extended data record)
    20, 10,                  // x, y
    100, 8};                 // width, height

// control group
_transfer Ctrl_Group ctrls = {
    2, 0,                   // number of controls, process ID
    &c_area};               // address of first control
```

Desktop commands

To display a window, we need to tell the desktop manager to open it. `symbos.h` provides many helper functions for managing windows (see the [system call reference](#)); the first one we need is `Win_Open()`:

```
char Win_Open(char bank, void* addr);
```

This opens the window defined at address `addr` in memory bank `bank`, and returns a window ID that will be needed to identify the window in subsequent commands. Usually, this is the main bank the application is running in, which can be found in the global variable `_symbank`. So, to open the window we defined earlier:

```
char winID;  
winID = Win_Open(_symbank, &form1);
```

Another important call is `Win_Close()`, which closes an open window (windows will also be closed automatically on app exit):

```
void Win_Close(char winID);
```

The last really important call is `Win_Redraw()`, which redraws a portion of the window content. SymbOS will not automatically redraw the window after we change something (for instance, changing the text of a button), so we need to do it manually:

```
void Win_Redraw(unsigned char winID, signed char what, unsigned char first);
```

Here, `what` can be either (1) -1, meaning “redraw entire window content”; (2) the control ID of a single control to redraw; or (3) a negative number (from -2 to -16) indicating how many controls should be redrawn (from 2 to 16), in which case `first` indicates the control ID of the first control to redraw. (There are separate commands for redrawing the menu, toolbar, titlebar, and statusbar; see the [system call reference](#).)

Handling events

SymbOS apps communicate with the desktop manager via interprocess messages. `symbos.h` provides helper functions that handle most of the necessary messages automatically (as with `Win_Open()` above). However, we still need to understand how messaging works internally in order to set up the main loop that will receive event notifications from the desktop manager.

Messages are passed via a small buffer which must be placed in the **transfer** segment. The standard message buffer in SCC is `_symmsg`, which is suitable for most tasks:

```
char _symmsg[14];
```

The core functions for sending and receiving messages are:

```
unsigned char Msg_Send(char rec_pid, char send_pid, char* msg);  
unsigned short Msg_Receive(char rec_pid, char send_pid, char* msg);  
unsigned short Msg_Sleep(char rec_pid, char send_pid, char* msg);
```

- `Msg_Send()` sends the message in the buffer `*msg` to the process ID `send_pid`.
- `Msg_Receive()` checks whether the process ID `send_pid` has sent us a message, and if so, places it in the buffer `*msg`.
- `Msg_Sleep()` is similar to `Msg_Receive()`, but if there is no message, it will sleep, releasing CPU time back to SymbOS until such a message arrives.

In each case `rec_pid` is the “inbound” process ID that should receive the response (that is, our own process ID, which can be found in the global variable `_sympid`). The process ID of the desktop manager is always 2, but to receive messages from any process, we can use -1.

After opening our starting window, we need to go into a loop that:

1. Calls `Msg_Sleep()` to wait for a message;
2. Checks the message type in the first byte, `msg[0]`; and
3. Takes any necessary action.

Message type codes are documented in the [event reference](#). The most immediately useful message is `MSR_DSK_WCLICK`, which is sent for most interactions with a window or form. The return message then has the format:

- `msg[0]`: `MSR_DSK_WCLICK`
- `msg[1]`: Window ID
- `msg[2]`: Action type, one of:
 - `DSK_ACT_CLOSE`: Close button has been clicked, or the user has typed Alt+F4.
 - `DSK_ACT_MENU`: A menu option has been clicked, with:
 - * `msg[8]` = Value of the clicked menu entry
 - `DSK_ACT_CONTENT` = A control has been clicked or modified, with:
 - * `msg[3]`: Sub-action, one of:
 - `DSK_SUB_MLCLICK`: Left mouse button clicked
 - `DSK_SUB_MRCLICK`: Right mouse button clicked
 - `DSK_SUB_MDCLICK`: Left mouse button double clicked
 - `DSK_SUB_MMCLICK`: Middle mouse button clicked
 - `DSK_SUB_KEY`: Key pressed, with key ASCII value in `msg[4]`
 - * `*(int*)&msg[4]` = Mouse X position relative to window content
 - * `*(int*)&msg[6]` = Mouse y position relative to window content
 - * `msg[8]` = control ID
 - `DSK_ACT_TOOLBAR`: Equivalent to `DSK_ACT_CONTENT`, but for controls in the toolbar.
 - `DSK_ACT_KEY`: A key has been pressed without modifying any control:
 - * `msg[4]` = key ASCII value

Putting this all together, an example of a skeleton main loop is below. This opens our previously-defined window `form1` and loops, waiting for an event. In this example, the only event implemented is the most important one: exiting the application when the user tries to close the window!

```
char winID;

int main(int argc, char *argv[]) {
    winID = Win_Open(_symlink, &form1);

    while (1) {
        // handle events
        _symmsg[0] = 0;
        Msg_Sleep(_sympid, -1, _symmsg);
        if (_symmsg[0] == MSR_DSK_WCLICK) {
```

```

    switch (_symmsg[2]) {
        case DSK_ACT_CLOSE: // Alt+F4 or click close
            exit();
            // more event cases go here...
    }
}
}
}

```

That's it, our first windowed application! (The complete code for this demo is available as `windemo.c` in the `samples` folder.) Everything else we might want to do—more controls, more windows, more interactivity—can be accomplished by adding more data records and event-handling code to this skeleton.

Next steps

Complete references to all available controls, event messages, and system calls are available in the following sections:

- [Control reference](#)
- [Event reference](#)
- [System call reference](#)

Information about advanced window features (menus, toolbars, etc.) is available in the following section.

Advanced windows

Menus

Compared to lists (see `C_LISTBOX`), menus are comparatively simple. We define a menu using a `Menu` struct directly followed by a series of `Menu_Entry` structs:

```

typedef struct {
    unsigned short entries; // number of Menu_Entry structs to follow
} Menu;

typedef struct {
    unsigned short flags; // flags (see below)
    char* text; // address of entry text
    unsigned short value; // value to return when clicked, or address of submenu
    unsigned short unused;
} Menu_Entry;

```

`flags` is an OR'd bitmask that may consist of one or more of the following flags:

- `MENU_ACTIVE`: entry is active and can be clicked (we usually want this).

- **MENU_CHECKED**: entry has a checkmark. Note that **MENU_CHECKED** will not be updated automatically—it is our responsibility to receive menu events and take the necessary action, such as toggling the **MENU_CHECKED** flag for a given entry.
- **MENU_SUBMENU**: entry opens a submenu. If **MENU_SUBMENU** is set, value points to the **Menu** struct defining the submenu to open. Submenus can be nested up to 5 levels deep.
- **MENU_SEPARATOR**: entry is a separator line.
- **MENU_ICON**: entry begins with an icon (SymbOS 4.0 and up only, and only allowed in drop-down menus, not a window's top-level menu bar.) If **MENU_ICON** is set, the entry's text string must begin with a five-byte inline image **control code** representing an 8x8 icon that will be plotted before the entry. (i.e., 0x06 0x80 banknum [canvas] (#graphics-library) address - 1).

Windows can have main menus if their **WIN_MENU** flag is set and their **menu** property points to a **Menu** struct similar to the above. Menus can also be opened independently (see the [Menu_Context\(\)](#) system call).

// example

```
_transfer Menu submenu = {2};
_transfer Menu_Entry submenu_row1 = {MENU_ACTIVE, "Entry 1", 1};
_transfer Menu_Entry submenu_row2 = {MENU_ACTIVE, "Entry 2", 2};

_transfer Menu mainmenu = {3};
_transfer Menu_Entry mainmenu_row1 = {MENU_ACTIVE | MENU_SUBMENU, "Submenu", (unsigned short)&submenu};
_transfer Menu_Entry mainmenu_row2 = {MENU_ACTIVE | MENU_CHECKED, "Option", 3};
_transfer Menu_Entry mainmenu_row3 = {0, "Inactive", 4};
```

Toolbars

Windows can have a separate “toolbar” region if their property **.toolheight** is greater than 0. This specifies the vertical height (in pixels) of the toolbar region, which will be plotted just below the titlebar or menubar (if present). This region is distinct from the window content in two ways: it does not scroll with the window content in a resizable window, and it has a separate control group data record.

The control group data record for a toolbar is structurally identical to the main control group data record for a window (including the need for a **C_AREA** control to fill in the background of the toolbar), but it is registered in the window's **.toolbar** property instead of its **.controls** property. Refreshing the toolbar must be done with **Win_Redraw_Toolbar** instead of **Win_Redraw**, and user events will be passed back with the **DSK_ACT_TOOLBAR** type instead of the **DSK_ACT_CONTENT** type.

Resizing calculations

The default behavior of resizable windows is to show only a portion of the main window content (optionally with scrollbars to allow the user to scroll to the rest of the content). However, we can also tell the desktop manager to automatically move and resize the controls on a window using “calculation rules.” Calculation rules are defined by a series of **Calc_Rule** structs, one struct per control in the window's main control group, with the format:


```
typedef struct {
    signed short xbase; // x base
    unsigned char xmult; // x multiplier
    unsigned char xdiv; // x divisor
    signed short ybase; // y base
    unsigned char ymult; // y multiplier
    unsigned char ydiv; // y divisor
    signed short wbase; // width base
    unsigned char wmult; // width multiplier
    unsigned char wdiv; // width divisor
    signed short hbase; // height base
    unsigned char hmult; // height multiplier
    unsigned char hdiv; // height divisor
} Calc_Rule;
```

The calculation rules for a control group should be placed immediately after one another in the **transfer** segment, one `Calc_Rule` per control, and the `calcrule` property of the window's main `Ctrl_Group` set to the address of the first `Calc_Rule`. When the window is resized, the `x`, `y`, `w`, and `h` properties of each `Ctrl` in the window's main `Ctrl_Group` will be automatically recalculated according to the formula:

- $x = xbase + (window_width * xmult / xdiv)$
- $y = ybase + (window_height * ymult / ydiv)$
- $w = wbase + (window_width * wmult / wdiv)$
- $h = hbase + (window_height * hmult / hdiv)$

An easy way to think about these formulas is in terms of which control margins you want to remain a constant distance from the corresponding side of the window area:

- To keep a control's left margin constant, use `xbase = left_margin`, `xmult = 0`, `xdiv = 1`.
- To keep a control's right margin constant, use `xbase = -(control_width + right_margin)`, `xmult = 1`, `xdiv = 1`.
- To keep both left and right margins constant, stretching the control to fit, use `xbase = left_margin`, `xmult = 0`, `xdiv = 1`, `wbase = -(left_margin + right_margin)`, `wmult = 1`, `wdiv = 1`.
- To keep a control's top margin constant, use `ybase = top_margin`, `ymult = 0`, `ydiv = 1`.
- To keep a control's bottom margin constant, use `ybase = -(control_height + bottom_margin)`, `ymult = 1`, `ydiv = 1`.
- To keep both top and bottom margins constant, stretching the control to fit, use `ybase = top_margin`, `ymult = 0`, `ydiv = 1`, `hbase = -(top_margin + bottom_margin)`, `hmult = 1`, `hdiv = 1`.

For example, to resize a single control to fill the entirety of the window content except for a 12-pixel area at the top of the window content (left margin = 0, right margin = 0, top margin = 12, bottom margin = 0):

```
_transfer Calc_Rule calcrule = {0, 0, 1, 12, 0, 1, 0, 1, 1, -12, 1, 1};
```

When using calculation rules, we should be careful about setting the maximum and minimum window sizes in the `Window` record to sizes where we know the calculations will be valid. For example, if we

have a control with a left margin of 25 and a right margin of 100 and we let the user resize the window to only 50 pixels wide, the calculated width of the control will be negative! We can avoid this situation by defining an appropriate minimum width, such as 150.

(An alternative approach is to recognize the “window resized” event `MSR_DSK_WRESIZ`, manually recalculate the `x`, `y`, `w`, and `h` properties of the affected controls, and redraw any controls that have moved or been resized by this calculation. However, using calculation rules will result in a smoother experience because the desktop manager can recalculate controls as the window is being resized, rather than redrawing them twice.)

Modal windows

Windows can be “modal”, that is, unable to be focused or interacted with as long as another window (here called the “topmost” window) is open. To establish a modal relationship, we need to set two values:

- Set the `WIN_MODAL` flag for the topmost window, i.e.: `form2.flags |= WIN_MODAL`.
- Set the `.modal` property of the lower window to the window ID of the topmost window, plus 1, i.e.: `form1.modal = form1id + 1`.

When the topmost window closes, unset both properties. If the user clicks the lower window while this relationship has been established, the desktop manager sends a `MSR_DSK_WMODAL` message to notify the application. (This is particularly useful to create dialogs that should be closed automatically if the user clicks the main window.)

Control reference

This section contains a reference to all available window controls (buttons, text, etc.). For details on how to use these controls in windows, see [SymbOS Programming](#).

C_AREA

Displays a rectangular area filled with the specified color.

Parameter: Color, e.g., COLOR_ORANGE. The parameter may optionally be OR'd with AREA_16COLOR (to enable 16-color mode) and AREA_XOR (to enable XOR mode, to be used only with 16-color mode): e.g., COLOR_WHITE | AREA_16COLOR.

```
// example
_transfer Ctrl c_area1 = {1, C_AREA, -1, COLOR_ORANGE, 0, 0, 100, 80};
```

C_TEXT

Displays a line of text in the default system font.

Parameter: Address of extended data record:

```
typedef struct {
    char* text;           // address of text
    unsigned char color;  // 4-color mode: (foreground << 2) | background
                        // 16-color mode: (foreground << 4) | background
    unsigned char flags;  // one of: ALIGN_LEFT, ALIGN_CENTER, ALIGN_RIGHT
                        // OR with TEXT_16COLOR for 16-color mode
} Ctrl_Text;
```

In “fill” mode, the background of the control is first filled in with the specified background color and the text is clipped to the size of the control rather than being allowed to overflow. To activate this mode in 4-color mode, OR .color with TEXT_FILL: e.g., (COLOR_BLACK << 2) | COLOR_ORANGE | TEXT_FILL. To activate this mode in 16-color mode, OR .flags with TEXT_FILL16: e.g., ALIGN_LEFT | TEXT_16COLOR | TEXT_FILL16.

```
// example
_transfer Ctrl_Text cd_text1 = {"Text", (COLOR_BLACK << 2) | COLOR_ORANGE, ALIGN_LEFT};
_transfer Ctrl c_text1 = {1, C_TEXT, -1, (unsigned short)&cd_text1, 10, 10, 80, 8};
```

C_TEXT_FONT

Displays a line of text in an alternative font.

Parameter: Address of extended data record:

```
typedef struct {
    char* text;           // address of text
    unsigned char color;  // 4-color mode: (foreground << 2) | background
```

```

// 16-color mode: (foreground << 4) | background
unsigned char flags; // one of: ALIGN_LEFT, ALIGN_CENTER, ALIGN_RIGHT
// OR with TEXT_16COLOR for 16-color mode
char* font; // address of font
} Ctrl_Text_Font;

```

The control height should be equal to the height of the font, and the font data must be stored in the same 16KB segment as the text (usually the **data** segment). A description of the font format can be found in the SymbOS Developer Documentation; fonts can be created using the SymbOS Font Editor application.

```

// example
_data char fontbuf[1538]; // a font would be loaded into this buffer
_transfer Ctrl_Text_Font cd_text_font1 =
    {"Text", (COLOR_BLACK << 2) | COLOR_ORANGE, ALIGN_LEFT, fontbuf};
_transfer Ctrl c_text_font1 =
    {1, C_TEXT_FONT, -1, (unsigned short)&cd_text_font1, 10, 10, 80, 8};

```

C_TEXT_CTRL

Displays a line of “rich” text with optional control codes that can change the appearance of the text mid-line. This renders more slowly than C_TEXT, but allows for very complex effects.

Parameter: Address of extended data record:

```

typedef struct {
    char* text; // address of text
    unsigned short maxlen; // maximum length (in bytes) of the text
    char* font; // address of starting font (-1 for default)
    unsigned char color; // (foreground << 4) | background
    unsigned char underline; // 1 = start with underline on
} Ctrl_Text_Ctrl;

```

The following control bytes can be included in the text string:

- 0x00 = end of string
- 0x01 0xNN = change text color, where NN is (foreground « 4) | background
- 0x02 0xAAAA = change font, where AAAA is the address of the font, or -1 for the default font. The font must be in the same 16KB segment as the text (usually the **data** segment).
- 0x03 = switch underlining on
- 0x04 = switch underlining off
- 0x05 0xNN = insert NN pixels of extra space before the next character
- 0x06 0xNN 0xBB 0xAAAA = plot a [graphics canvas](#) as an inline image, where NN is the Y position relative to the line (0 = 128 pixels up, 128 = in line with text, 255 = 128 pixels down); BB is the bank number of the canvas, or -1 for “same bank as text”; and AAAA is the address of the canvas + 1. (This feature is only available in SymbOS 4.0 and up.)
- 0x08 to 0x0B = skip next (code - 8) * 2 + 1 bytes
- 0x0C to 0x1F = insert (code - 8) pixels of extra space before the next character

```
// example
_transfer Ctrl_Text_Ctrl cd_text_ctrl1 =
    {"Text \x03underlined\x04 and not", 100, -1, (COLOR_BLACK << 2) | COLOR_ORANGE, 0};
_transfer Ctrl_c_text_ctrl1 =
    {1, C_TEXT_CTRL, -1, (unsigned short)&cd_text_ctrl1, 10, 10, 80, 8};
```

C_FRAME

Displays a rectangular frame.

Parameter: Color and flags:

- 4-color mode: (area_color << 4) | (lower_right_color << 2) | upper_left_color; OR with AREA_FILL to fill interior.
- 16-color mode: (lower_right_color << 12) | (upper_right_color << 8) | area_color | AREA_16COLOR; OR with AREA_FILL to fill interior.

An optional XOR mode inverts the colors underneath the control, like a rubber-band selection. In XOR mode, the parameter is just FRAME_XOR.

```
// example
_transfer Ctrl_c_frame1 =
    {1, C_FRAME, -1, (COLOR_ORANGE << 4) | (COLOR_RED << 2) | COLOR_BLACK | AREA_FILL,
    10, 10, 64, 64};
```

C_TFRAME

Displays a rectangular frame with a line of text at the top.

Parameter: Address of extended data record:

```
typedef struct {
    char* text;           // address of text
    unsigned char color;  // 4-color mode: (foreground << 2) | background
                        // 16-color mode: line color
                        // OR with AREA_16COLOR for 16-color mode
    unsigned char color2; // 16-color mode only: (foreground << 4) | background
} Ctrl_TFrame;
```

```
// example
_transfer Ctrl_TFrame cd_tframe1 =
    {"Title", COLOR_BLACK | AREA_16COLOR, (COLOR_BLACK << 2) | COLOR_LBLUE};
_transfer Ctrl_c_tframe1 =
    {1, C_TFRAME, -1, (unsigned short)&cd_tframe1, 10, 10, 64, 64};
```

C_GRID

Displays repeated horizontal or vertical lines. By overlapping two C_GRID controls, one horizontal and one vertical, it is also possible to draw a grid of boxes. (This control is only available in SymbOS 4.0 and up.)

Parameter: Address of extended data record:

```
typedef struct {
    unsigned char color;    // line color, OR'd with either GRID_HORIZONTAL or GRID_VERTICAL
    unsigned char lines;    // number of lines to draw
    unsigned char spacing; // (spacing << 1) | GRID_FIXED
} Ctrl_Grid;
```

Grid lines will start being drawn at the control's location *x*, *y* and continue down (for horizontal lines) or right (for vertical lines) for the specified number of lines. The *spacing* parameter requires some explanation. If this value is OR'd with GRID_FIXED (e.g., (8 << 1) | GRID_FIXED), all the lines will be drawn with the specified spacing (8 pixels in the preceding example). If GRID_FIXED is omitted, however, each line will have its own spacing. In this case, the *spacing* parameter indicates the spacing for the first line, and the *spacing* parameter for subsequent lines will be drawn from a char array that must immediately follow the Ctrl_Grid record (one char per remaining line). (The values in this array are treated exactly like the original spacing parameter, so once an array element with GRID_FIXED is encountered, the desktop manager will stop reading the array and draw all subsequent lines with the same spacing.)

```
// example
_transfer Ctrl_Grid cd_grid1 = {COLOR_BLACK | GRID_HORIZONTAL, 10, (8 << 1)};
_transfer char grid1_sp[9] = {(8 << 1), (10 << 1), (12 << 1),
                             (8 << 1), (10 << 1), (12 << 1),
                             (8 << 1), (10 << 1), (12 << 1)};

_transfer Ctrl c_grid1 = {1, C_GRID, -1, (unsigned short)&cd_grid1, 10, 10, 1000, 1000};
```

C_PROGRESS

Displays a progress bar.

Parameter: Color and progress: (progress << 8) | (empty_color << 6) | (filled_color << 4) | (lower_right_color << 2) | upper_left_color. Progress is measured from 0 (empty) to 255 (full).

```
// example
_transfer Ctrl c_progress1 =
    {1, C_PROGRESS, -1,
     (119 << 8) | (COLOR_ORANGE << 6) | (COLOR_RED << 4) | (COLOR_BLACK << 2) | COLOR_BLACK,
     10, 10, 64, 8};
```

C_IMAGE

Displays a standard image (4-color SGX format only). (See the [graphics library](#) for how to use this in practice.)

Control type: C_IMAGE.

Parameter: Address of the image data.

```
// example
_data char imgbuf[198];
_transfer Ctrl c_image1 = {1, C_IMAGE, -1, (unsigned short)imgbuf, 10, 10, 24, 24};
```

C_IMAGE_EXT

Displays an image with an extended graphics header. (See the [graphics library](#) for how to use this in practice.)

Parameter: Address of the extended graphics header. Extended graphics are complicated, but allow plotting 16-color images and breaking up an image that is larger than 256 pixels wide/tall into multiple blocks that can be displayed side by side. The details of the graphics format are described in the SymbOS Developer Documentation. In practice, it is usually easier to use the [graphics library](#) rather than trying to deal with extended graphics headers directly. However, if needed, SCC provides a struct type (Img_Header) to implement the header itself:

```
typedef struct {
    unsigned char bytew; // width of the complete graphic in bytes (must be even)
    unsigned char w;    // width of this block in pixels
    unsigned char h;    // height of this block in pixels
    char* addrData;     // address of graphic data + offset
    char* addrEncoding; // address of encoding byte just before the start of the whole graphic
    unsigned short len; // size of the complete graphic in bytes
} Img_Header;
```

```
// example
_data Img_Header imghead; // fill with the appropriate data
_transfer Ctrl c_image_ext1 = {1, C_IMAGE_EXT, -1, (unsigned short)&imghead, 10, 10, 24, 24};
```

C_IMAGE_TRANS

Same as C_IMAGE_EXT, except that color 0 will be transparent.

Parameter: Address of the extended graphics header, as above.

C_ICON

Displays a 24x24 icon with up to two lines of text below it.

Parameter: Address of extended data record:

```
typedef struct {
    char* icon;           // address of graphic or graphic header (e.g., a canvas)
    char* line1;          // address of first line of text
    char* line2;          // address of second line of text
    unsigned char flags;   // 4-color mode: (foreground_color << 2) | background_color
    unsigned char color16; // 16-color mode: (foreground_color << 4) | background_color
    unsigned char extflags; // extended mode flags
} Ctrl_Icon;
```

The following flags can be OR'd with .flags:

- ICON_STD: .icon points to standard graphics data (4-color SGX)
- ICON_EXT: .icon points to extended graphics header, e.g., a (canvas)[graphics.md#using-canvases] (see C_IMAGE_EXT)
- ICON_16COLOR: use 16-color mode for text colors (this does **not** mean the icon itself is 16-color, which is determined by the image header).
- ICON_MOVEABLE: icon can be moved by the user
- ICON_EXTOPTS: icon has extended options

When .flags includes ICON_EXTOPTS, the following flags can be OR'd with .extflags:

- ICON_MARKABLE: icon can be marked (selected) by user
- ICON_MARKED: icon is currently marked (selected) by user

The width of the control must be 48 and the height 40.

```
// example
_data char imgdata[198]; // store an icon image here
_transfer Ctrl_Icon cd_icon1 =
    {imgdata, "Line 1", "Line 2", (COLOR_BLACK << 2) | COLOR_YELLOW | ICON_STD | ICON_4COLOR};
_transfer Ctrl c_icon1 = {1, C_ICON, -1, (unsigned short)&cd_icon1, 10, 10, 48, 40};
```

C_BUTTON

Displays a button. Control height must always be 12.

Parameter: Address of the button text.

```
// example
_transfer Ctrl c_button1 = {1, C_BUTTON, -1, (unsigned short)"Text", 10, 10, 48, 12};
```

C_CHECK

Displays a checkbox.

Parameter: Address of extended data record:

```
typedef struct {
    char* status;         // address of status byte
    char* text;           // address of text
```



```
    unsigned char color; // (foreground << 2) | background
} Ctrl_Check;
```

The value of the checkbox (0 = unchecked, 1 = checked) will be stored in the byte pointed to by `status`, which should be in the **transfer** segment. The control height should always be 8.

```
// example
_transfer char check1 = 0;
_transfer Ctrl_Check cd_check1 = {&check1, "Label text", (COLOR_BLACK << 2) | COLOR_ORANGE};
_transfer Ctrl c_check1 = {1, C_CHECK, -1, (unsigned short)&cd_check1, 10, 10, 32, 8};
```

C_RADIO

Displays a radio button (circular checkbox). Selecting one radio button in a group will unselect all the others, allowing the user to select just one option.

Parameter: Address of extended data record:

```
typedef struct {
    char* status; // address of status byte
    char* text; // address of text
    unsigned char color; // (foreground << 2) | background
    unsigned char value; // value to load into status when selected
    char* buffer; // address of 4-byte coordinate buffer (see below)
} Ctrl_Radio;
```

When a radio button is selected, its `value` property will be loaded into the byte pointed to by `status`, which should be in the **transfer** segment. Usually this is an ID of some kind (1, 2, 3...), but it can technically be anything. To determine which radio button is selected, read the value of the byte pointed to by `status` and match it to the known `value` properties of the possible radio buttons.

`buffer` points to a 4-byte static buffer used internally by the desktop manager. It should initially contain the values -1, -1, -1, -1 and be in the **transfer** segment.

All radio buttons in a group should use the same `status` and `buffer`; to create multiple groups that do not interact, give them different `status` bytes and coordinate buffers.

The control height should always be 8.

```
// example
_transfer char radio = 0;
_transfer char radiocoord[4] = {-1, -1, -1, -1};

_transfer Ctrl_Radio cd_radio1 =
    {&radio, "First button", (COLOR_BLACK << 2) | COLOR_ORANGE, 1, radiocoord};
_transfer Ctrl_Radio cd_radio2 =
    {&radio, "Second button", (COLOR_BLACK << 2) | COLOR_ORANGE, 2, radiocoord};
_transfer Ctrl_Radio cd_radio3 =
    {&radio, "Third button", (COLOR_BLACK << 2) | COLOR_ORANGE, 3, radiocoord};
```

```
_transfer Ctrl c_radio1 = {1, C_RADIO, -1, (unsigned short)&cd_radio1, 10, 10, 32, 8};
_transfer Ctrl c_radio2 = {2, C_RADIO, -1, (unsigned short)&cd_radio2, 10, 20, 32, 8};
_transfer Ctrl c_radio3 = {3, C_RADIO, -1, (unsigned short)&cd_radio3, 10, 30, 32, 8};
```

C_HIDDEN

Nothing will be displayed, but any clicks to the area of the control will be sent as events for this control ID.

// example

```
_transfer Ctrl c_hidden1 = {1, C_HIDDEN, -1, 0, 10, 30, 32, 8};
```

C_TABS

Displays a row of tabs.

Parameter: Address of extended data record:

```
typedef struct {
    unsigned char tabs;      // number of tabs
    unsigned char color;     // (lower_right << 6) | (upper_left << 4) | (fore << 2) | back
    unsigned char selected;  // number of selected tab
} Ctrl_Tabs;
```

This data structure should be immediately followed by the number of Ctrl_Tab structs indicated by the tabs property:

```
typedef struct {
    char* text;              // address of text
    signed char width;       // width in pixels, or -1 to autocalculate
} Ctrl_Tab;
```

The control height must be 11. SymbOS will keep track of the selected tab for us, but it is our responsibility to decide what a click event on a tab actually does (for example, hiding one page of controls and revealing another). The event-handling code is also responsible for refreshing the tab control so the updated selection is actually displayed. A good way to switch between several pages of controls is to create each page as a separate Ctrl_Group and have a C_COLLECTION control pointing to the Ctrl_Group of the visible page. When the tab bar is clicked, change which Ctrl_Group the collection's controls property points to and redraw it.

// example

```
_transfer Ctrl_Tabs cd_tabs =
    {2, (COLOR_BLACK << 6) | (COLOR_RED << 4) | (COLOR_BLACK << 2) | COLOR_ORANGE, 1};
_transfer Ctrl_Tab cd_tab1 = {"Tab 1", -1};
_transfer Ctrl_Tab cd_tab2 = {"Tab 2", -1};

_transfer Ctrl c_tabs = {1, C_TABS, -1, (unsigned short)&cd_tabs, 10, 10, 64, 11};
```

C_SLIDER

Displays a slider (scrollbar or value selector).

Parameter: Address of extended data record:

```
typedef struct {
    unsigned char type;        // type (see below)
    unsigned char unused;
    unsigned short value;      // current value/position
    unsigned short maxvalue;   // maximum value/position (minimum is 0)
    signed char increment;     // value change when clicking up/right button
    signed char decrement;     // value change when clicking down/left button
} Ctrl_Slider;
```

Type is an OR'd bitmask consisting of one or more of the following flags:

- SLIDER_H: horizontal slider
- SLIDER_V: vertical slider
- SLIDER_VALUE: value slider
- SLIDER_SCROLL: scrollbar slider

For a horizontal slider, the control height must be 8 and the control width must be at least 24. For a vertical slider, the control width must be 8 and the control height must be at least 24. We are responsible for reading the slider's value and deciding what to do with it.

```
// example
_transfer Ctrl_Slider cd_slider1 = {SLIDER_H | SLIDER_SCROLL, 0, 15, 30, 1, -1};
_transfer Ctrl c_tabs = {1, C_SLIDER, -1, (unsigned short)&cd_slider1, 10, 10, 100, 8};
```

C_COLLECTION

Contains a collection of subcontrols. A collection behaves similarly to the main window content: controls will be clipped to the bounds of the collection, and if the full size of the content is larger than the size of the collection, the user will be able to scroll around the collection's content with scrollbars.

Parameter: Address of extended data record:

```
typedef struct {
    void* controls;            // address of Ctrl_Group defining the controls in the collection
    unsigned short wfull;      // full width of content, in pixels
    unsigned short hfull;      // full height of content, in pixels
    unsigned short xscroll;     // horizontal scroll position, in pixels
    unsigned short yscroll;     // vertical scroll position, in pixels
    unsigned char flags;        // flags (see below)
} Ctrl_Collection;
```

If scrollbars are enabled, the control size must be greater than 32x32. `flags` is one of the following:

- CSCROLL_NONE: display no scrollbars

- C_SCROLL_H: display horizontal scrollbar
- C_SCROLL_V: display vertical scrollbar
- C_SCROLL_BOTH: display both scrollbars

// example

```
_transfer Ctrl cc_area = {1, C_AREA, -1, COLOR_ORANGE, 0, 0, 100, 100};
_transfer Ctrl_Group cg_collect = {1, 0, &cc_area};
_transfer Ctrl_Collection cd_collect = {&cg_collect, 200, 100, 0, 0, C_SCROLL_H};
_transfer Ctrl c_collect = {1, C_COLLECTION, -1, (unsigned short)&cd_collect, 10, 10, 100, 100};

int main(int argc, char* argv[]) {
    cg_collect.pid = _sympid; // ensure collection's Ctrl_Group has the correct
                             // process ID for sending back events

    /* ... */
}
```

C_INPUT

Displays a single-line text input field.

Parameter: Address of extended data record:

```
typedef struct {
    char* text;           // address of text buffer (in data segment)
    unsigned short scroll; // scroll position, in bytes
    unsigned short cursor; // cursor position, in bytes
    signed short selection; // number of selected characters relative to cursor
    unsigned short len;    // current text length, in bytes
    unsigned short maxlen; // maximum text length, in bytes (not including 0-terminator)
    unsigned char flags;   // flags (see below)
    unsigned char textcolor; // (foreground << 4) | background
    unsigned char linecolor; // (lower_right_color << 4) | upper_left_color
} Ctrl_Input;
```

flags is an OR'd bitmask which may contain one or more the following:

- INPUT_PASSWORD: show all chars as *
- INPUT_READONLY: input is read-only
- INPUT_ALTCOLS: use alternate colors (if this flag is not specified, the textcolor and linecolor options will be ignored)
- INPUT_MODIFIED: represents a bit set when the text is modified

selection = 0 when no characters are selected, greater than 0 when the cursor marks the start of the selection, and less than 0 when the cursor marks the end of the selection.

The control height should always be 12. Note that, if we wish the input box to be prefilled, we must be sure to set the properties (cursor, len, etc.) correctly. (For input by the user, SymbOS will update these properties automatically.)

```
// example
_data char cd_input_buf[25];
_transfer Ctrl_Input cd_input1 =
    {cd_input1_buf, 0, 0, 0, 0, 24, INPUT_ALTCOLS,
     (COLOR_RED << 4) | COLOR_YELLOW, (COLOR_RED << 4) | COLOR_BLACK};
_transfer Ctrl c_input1 = {1, C_INPUT, -1, (unsigned short)&cd_input1, 10, 10, 100, 12};
```

C_TEXTBOX

Displays a multi-line text input box (text editor).

Parameter: Address of extended data record:

```
typedef struct {
    char* text;                // address of text buffer (in data segment)
    unsigned short unused1;
    unsigned short cursor;    // cursor position, in bytes
    signed short selection;   // number of selected characters relative to cursor
    unsigned short len;       // current text length, in bytes
    unsigned short maxlen;    // maximum text length, in bytes (not including 0-terminator)
    unsigned char flags;      // flags (see below)
    unsigned char textcolor;  // (foreground << 4) | background, when using INPUT_ALTCOLS flag
    unsigned char unused2;
    char* font;               // font address, when using INPUT_ALTFONT flag (in data segment)
    unsigned char unused3;
    unsigned short lines;     // number of lines of text
    signed short wrapwidth;   // wrapping width, in pixels (-1 for no wrapping)
    unsigned short maxlines;  // maximum number of lines
    signed short xvisible;    // (used internally, set to -8 initially to force reformatting)
    signed short yvisible;    // (used internally, set to 0)
    void* self;               // address of this data record
    unsigned short xtotal;    // (used internally, set to 0)
    unsigned short ytotal;    // (used internally, set to 0)
    unsigned short xoffset;   // (used internally, set to 0)
    unsigned short yoffset;   // (used internally, set to 0)
    unsigned char wrapping;   // WRAP_WINDOW or WRAP_WIDTH
    unsigned char tabwidth;   // tab stop width (1-255, or 0 for no tab stop)
    int column;               // TextBox_Pos() returns values here
    int line;                 // TextBox_Pos() returns values here
    char reserved[4];         // (used internally, set to 0)
} Ctrl_TextBox;
```

The data record must be immediately followed by a buffer consisting of as many 16-bit words (e.g., unsigned short) as there are the maximum number of lines in the text data (maxlines). This buffer will hold the length (in bytes) of each line in the textbox, with the high bit set if this length includes a Windows-style `\r\n` end-of-line marker at the end of the line. Due to a quirk in SCC's linker, which

currently treats initialized and uninitialized arrays differently, this buffer must be given an initial value (such as `{0}`) to ensure that it is placed directly after the `Ctrl_TextBox` data structure in the **transfer** segment. (This may be improved in future releases.) For example:

```
_transfer unsigned short lines[1000] = {0};
```

`flags` is an OR'd bitmask which may contain one or more the following:

- `INPUT_READONLY`: input is read-only
- `INPUT_ALTCOLS`: use alternate colors (if this flag is not specified, the `textcolor` option will be ignored)
- `INPUT_ALTFONT`: use alternate font (if this flag is not specified, the `font` option will be ignored)
- `INPUT_MODIFIED`: represents a bit set when the text is modified

`wrapmode` must be either `WRAP_WINDOW` (wrap at window border) or `WRAP_WIDTH` (wrap at the width specified in `wrapwidth`). To use no wrapping, set `wrapmode = WRAP_WIDTH` and `wrapwidth = -1`.

`self` is a bit tricky: this must be the address of the data record itself, but this cannot be set in a static initializer because, at the time the compiler is parsing the initializer, the data record's symbol has not yet been fully defined! Instead, we must set this at runtime with a statement like:

```
cd_textbox1.self = &cd_textbox1;
```

`selection = 0` when no characters are selected, greater than 0 when the cursor marks the start of the selection, and less than 0 when the cursor marks the end of the selection.

To update the contents, cursor, or selection of the textbox, it is recommended to use the special system calls `TextBox_Redraw()` and `TextBox_Select()` rather than trying to update all the relevant properties manually.

Note that, because the buffer must be stored in a continuous 16KB segment (usually the **data** segment), this control is effectively limited to no more than 16KB of text. Note also that, if we wish the textbox to be prefilled, we must be sure to set the properties (`cursor`, `len`, etc.) correctly. (For input by the user, SymbOS will update these properties automatically.)

// example

```
_data char textbuf[4096];
_transfer Ctrl_TextBox cd_textbox1 = {
    textbuf,          // text address
    0, 0, 0, 0,       // unused1, cursor, selection, len
    4095, 0, 0, 0,     // maxlen, flags, textcolor, unused2
    0, 0, 0, -1,      // font, unused3, lines, wrapwidth
    1000, -8, 0,       // maxlines, xvisible, yvisible
    0,                // self
    200, 100, 0, 0,    // xtotal, ytotal, xoffset, yoffset
    WRAP_WIDTH, 20}; // wrapping, tabwidth
_transfer unsigned short textbox1_lines[1000] = {0}; // line-length buffer
_transfer Ctrl c_textbox1 = {1, C_TEXTBOX, -1, (unsigned short)&cd_textbox1, 0, 0, 200, 100};

int main(int argc, char* argv[]) {
```

```

    cd_textbox1.self = &cd_textbox1; // fill "self" property at runtime
    /* ... */
}

```

C_LISTBOX

Displays a list box, which may have one or more columns. The structure of lists is somewhat complicated, and is described below.

Parameter: Address of a List data record:

```

typedef struct {
    unsigned short lines;      // number of list rows
    unsigned short scroll;     // index of first shown row
    void* rowdata;            // address of the row data (see below)
    unsigned short status;     // (only used by tree views, see C_TREE)
    unsigned char columns;     // number of columns (from 1 to 64)
    unsigned char sorting;     // sorting flags (see below)
    void* coldata;            // address of the column data (see below)
    unsigned short clicked;    // index of last clicked row
    unsigned char flags;       // flags (see below)
    unsigned char resorted;    // (SymbOS 4.0+ sets this to 1 when the user re-sorts the list)
    unsigned short treelines;  // (only used by tree views, see C_TREE)
    unsigned short treefirst;  // (only used by tree views, see C_TREE)
} List;

```

flags for List is an OR'd bitmask which may contain one or more of the following:

- LIST_SCROLL: show scrollbar
- LIST_MULTI: allow multiple selections

sorting is the index of the column to sort by (from 1 to 64), optionally OR'd with SORT_AUTO to automatically sort on first display and/or SORT_REVERSE to sort in descending order (rather than the default, ascending order).

rowdata and coldata must point to additional data structures in a specific format and sequence. Column definitions consist of a series of List_Column structs directly after one another in the **transfer** segment, as many structs are there are columns:

```

typedef struct {
    unsigned char flags;       // flags (see below)
    unsigned char sortskip;    // number of characters to skip when sorting (normally 0)
    unsigned short width;     // width in pixels
    char* text;                // address of title text
    unsigned short unused2;
} List_Column;

```

flags for List_Column controls how the column is aligned and sorted, as well as what is displayed on each row. It consists of one of ALIGN_LEFT, ALIGN_RIGHT, or ALIGN_CENTER; OR'd with one of LTYPE_TEXT

(for text data), `LTYPE_IMAGE` (for image data), `LTYPE_16` (for a 16-bit number), or `LTYPE_32` (for a 32-bit number). Alternatively, on SymbOS 4.0 and up, `flags` may just be set to `LTYPE_CTRL`, with no other flags (indicating text with control codes—see [C_TEXT_CTRL](#) for what codes are available).

`sortskip` (only available in SymbOS 4.0 and up) indicates the index of the first character to compare when sorting text rows. Normally this should be 0 (i.e., “sort from the first character”), but we can set it >0 to skip leading characters (like control characters).

The format of row definitions is a bit trickier. Internally, each row consists of a 16-bit flags word, followed by as many 16-bit value words as there are columns; these store either the value of the row in that column (for `LTYPE_16`) or the address of the data shown in that column. For a 1-column list, we can represent this structure with a series of `List_Row` structs, directly after one another:

```
typedef struct {
    unsigned short flags; // flags (see below)
    char* value;          // address of row content (or value for LTYPE_16 - cast to char*)
} List_Row;
```

`flags` for `List_Row` (and its variants) is a 14-bit numeric value associated with the row. If the row is selected, the high bit of `flags` will be set; this can be tested with `row.flags & ROW_MARKED`.

Additional struct types for multi-column rows (up to 4 columns) are defined in `symbos/windows.h`; for example:

```
typedef struct {
    unsigned short flags;
    char* value1;
    char* value2;
} List_Row2Col;
```

The control width must be at least 11, and the control height must be at least 16. Note that column names will not be displayed in the base `C_LISTBOX` control type; to show them, use `C_LISTFULL` instead.

```
// example
_transfer List_Column cd_list1_col1 = {ALIGN_LEFT | LTYPE_TEXT, 0, 80, "Column 1"};
_transfer List_Column cd_list1_col2 = {ALIGN_LEFT | LTYPE_TEXT, 0, 80, "Column 2"};

_transfer List_Row2Col cd_list1_r1 = {1, "Row 1", "Value 1"};
_transfer List_Row2Col cd_list1_r2 = {2, "Row 2", "Value 2"};
_transfer List_Row2Col cd_list1_r3 = {3, "Row 3", "Value 3"};

_transfer List cd_list1 =
    {3, 0, &cd_list1_r1, 0, 2, 1 | SORT_AUTO, &cd_list1_col1, 0, LIST_MULTI};

_transfer Ctrl c_list1 = {1, C_LISTBOX, -1, (unsigned short)&cd_list1, 0, 0, 160, 100};
```


C_LISTFULL

Equivalent to C_LISTBOX, but also displays column titles. Clicking a column title will sort by that column. Control width must be at least 11, control height must be at least 26.

Parameter: Same as for C_LISTBOX.

C_LISTTITLE

Displays the column titles of a listbox or tree view, in isolation. Control height must always be 10.

Parameter: Same as for C_LISTBOX.

C_DROPDOWN

Equivalent to C_LISTBOX, but displays a dropdown list selector instead of a full list box. The control width must be at least 11, and the control height must be 10. The LIST_SCROLL flag should be used whenever the list is longer than 10 entries, and the LIST_MULTI flag should never be used. The .clicked property indicates the currently selected row.

Note that, even though dropdown lists generally only have one column and do not display a column title, we must still define a valid column struct as described under C_LISTBOX.

Parameter: Same as for C_LISTBOX.

C_TREE

Equivalent to C_LISTBOX, but displays a tree view listbox where the user can fold or expand rows as groups of nested nodes. (This control is only available in SymbOS 4.0 and up.)

Parameter: Same as for C_LISTBOX, with the following differences:

- .sorting should be set to SORT_TREE.
- .flags should never have LIST_MULTI set.

The format for columns and rows is also slightly different. Columns are defined in the usual way, except that flags for the first List_Column record should either be LTYPE_TREE (for normal text) or LTYPE_TREE | LTYPE_CTRL (for [text with control codes](#)). Rows are defined slightly differently, using Tree_Row structs instead of List_Row structs:

```
typedef struct {
    unsigned char indent; // indentation level (see below)
    unsigned char flags;  // flags (see below)
    char* value;          // address of row content (or value for LTYPE_16 - cast to char*)
    unsigned short id;    // numerical ID of row
} Tree_Row;
```

flags for Tree_Row (and its variants) is an OR'd bitmask of one or more of the following:

- TREE_NODE - indicates that this row is a node (and starts a group of collapsible child rows).
- TREE_EXPANDED - indicates that subsequent child rows are currently expanded and visible.

- `TREE_HIDDEN` - indicates that this row is currently hidden.
- `TREE_MARKED` - indicates that this row is currently marked/selected.

Conceptually, a tree consists of a series of rows, each of which can be either a node (which will be followed by more child rows) or a leaf (which has no children). The first node has `indent = 0`, and any children of a node should have `indent` set to 1 greater than the `indent` of the parent. When the user collapses a node, all of its children (i.e., all contiguous rows following it with a greater `indent` than the node that was clicked) will be hidden.

Additional struct types for multi-column rows (up to 4 columns) are defined in `symbos/windows.h`; for example:

```
typedef struct {
    unsigned char indent;
    unsigned char flags;
    char* value1;
    char* value2;
    unsigned short id;
} Tree_Row2Col;
```

Note that all visibility flags, etc. must be initialized correctly for how we want the tree to initially look. (Any subsequent modifications by the user will be tracked automatically.) Indentation levels must be correct, and all collapsed lines must have `TREE_HIDDEN` set. In addition, the text strings for the first column must begin with a special character followed by a space, which will be used to display the node symbols:

- `\x7F` - for a leaf with no children.
- `\x81` - for a collapsed node.
- `\x82` - for an expanded node.

In addition to being dispatched as a normal `DSK_ACT_CONTENT` event, any user events that collapse or expand a node will be recorded in the `status` property of the tree's `List` struct. If `status & 0x8000` is nonzero, a node event has taken place, and the relevant information can be extracted as:

- `status & 0x0FFF` - the numerical ID of the node row
- `status & 0x4000` - nonzero if expanded, otherwise collapsed

Control width must be at least 11, and control height must be at least 16. Note that column titles will not be displayed automatically; to show them, use a separate `C_LISTTITLE` control, as in the example below.

Tip: By using the `LTYPE_TREE | LTYPE_CTRL` setting and using [control codes](#) to draw inline images, it is possible to include icons in the tree view.

```
// example
_transfer List_Column cd_tree1_col1 = {LTYPE_TREE, 0, 80, "Column 1"};
_transfer List_Column cd_tree1_col2 = {ALIGN_LEFT | LTYPE_TEXT, 0, 80, "Column 2"};

_transfer Tree_Row2Col cd_tree1_r1 = {0, TREE_NODE | TREE_EXPANDED, "\x82 Node 1", "Value 1", 1};
_transfer Tree_Row2Col cd_tree1_r2 = {1, 0, "\x7F Leaf 1", "Value 2", 2};
```

```
_transfer Tree_Row2Col cd_tree1_r3 = {1, TREE_NODE | TREE_EXPANDED, "\x82 Node 2", "Value 3", 3};
_transfer Tree_Row2Col cd_tree1_r4 = {2, 0, "\x7F Leaf 2", "Value 4", 4};
_transfer Tree_Row2Col cd_tree1_r5 = {0, TREE_NODE, "\x81 Node 3", "Value 5", 5};
_transfer Tree_Row2Col cd_tree1_r6 = {1, TREE_HIDDEN, "\x7F Leaf 3", "Value 6", 6};

_transfer List cd_tree1 = {6, 0, &cd_tree1_r1, 0, 2, SORT_TREE, &cd_tree1_col1, 0, LIST_SCROLL};

_transfer Ctrl c_treetitle1 = {1, C_LISTTITLE, -1, (unsigned short)&cd_tree1, 0, 0, 160, 10};
_transfer Ctrl c_treecontent1 = {2, C_TREE, -1, (unsigned short)&cd_tree1, 0, 10, 160, 100};
```

Event reference

This section contains a reference to all event messages sent by the desktop service (button clicks, etc.). For details on how to trap and respond to these events, see [SymbOS Programming](#).

MSR_DSK_WCLICK

The primary event sent for most interactions with a window's controls.

- msg[0]: MSR_DSK_WCLICK
- msg[1]: Window ID
- msg[2]: Action type, one of:
 - DSK_ACT_CLOSE: Close button has been clicked, or the user has typed Alt+F4.
 - DSK_ACT_MENU: A menu option has been clicked, with:
 - * msg[8] = Value of the clicked menu entry
 - DSK_ACT_CONTENT = A control has been clicked or modified, with:
 - * msg[3]: Sub-action, one of:
 - DSK_SUB_MLCLICK: Left mouse button clicked
 - DSK_SUB_MRCLICK: Right mouse button clicked
 - DSK_SUB_MDCLICK: Left mouse button double clicked
 - DSK_SUB_MMCLICK: Middle mouse button clicked
 - DSK_SUB_KEY: Key pressed, with key ASCII value in msg[4]
 - * *(int*)&msg[4] = Mouse X position relative to window content
 - * *(int*)&msg[6] = Mouse y position relative to window content
 - * msg[8] = control ID
 - DSK_ACT_TOOLBAR: Equivalent to DSK_ACT_CONTENT, but for controls in the toolbar.
 - DSK_ACT_KEY: A key has been pressed without modifying any control:
 - * msg[4] = key ASCII value

MSR_DSK_WFOCUS

The focus status of a window has changed.

- msg[0]: MSR_DSK_WFOCUS
- msg[1]: Window ID
- msg[2]: Event type, one of:
 - 0 = window lost focus
 - 1 = window received focus

MSR_DSK_CFOCUS

Sent when the focus status of a control has changed.

- msg[0]: MSR_DSK_CFOCUS
- msg[1]: Window ID
- msg[2]: Number of newly focused control (not the control ID/value, but an index starting from 1)

- `msg[3]`: Event type, one of:
 - 0 = User navigated with mouse
 - 1 = User navigated with Tab key

MSR_DSK_WRESIZ

A window has been resized by the user (including maximizing or restoring a maximized or minimized window). The new window size can be determined using `Win_Width()` and `Win_Height()`. (We can also read the `h` and `w` properties of the window directly, but these may not match the actual window size in all cases, e.g., when the window is maximized.)

- `msg[0]`: `MSR_DSK_WRESIZ`
- `msg[1]`: Window ID

MSR_DSK_WSCROLL

The user has scrolled the main window content of a window. The new scroll position can be read from the window's `xscroll` and `yscroll` properties.

- `msg[0]`: `MSR_DSK_WSCROLL`
- `msg[1]`: Window ID

MSR_DSK_MENCTX

The user has clicked or cancelled an open context menu.

- `msg[0]`: `MSR_DSK_MENCTX`
- `msg[1]`: Event type, one of:
 - 0 = menu cancelled
 - 1 = entry clicked
- `*(int*)&msg[2]`: Value associated with the clicked entry
- `msg[4]`: Menu entry type, one of:
 - 0 = normal entry
 - 1 = checked entry

MSR_DSK_EVTCLK

The user has clicked a system tray icon associated with this application.

- `msg[0]`: `MSR_DSK_EVTCLK`
- `msg[1]`: Value associated with the system tray icon
- `msg[2]`: Mouse button pressed, one of:
 - `SYSTRAY_LEFT` = left click
 - `SYSTRAY_RIGHT` = right click
 - `SYSTRAY_DOUBLE` = double left click

MSR_DSK_WMODAL

The user has clicked a window that is modal and cannot be focused. (This is useful for creating windows that disappear if the user clicks the main window.)

- `msg[0]`: MSR_DSK_WMODAL
- `msg[1]`: Modal window ID

System call reference

The system calls in this reference are all available after including `symbolos.h`:

```
#include <symbolos.h>
```

When cross-compiling on a modern computer, it is usually easiest to just include the entirety of `symbolos.h`. However, this is a large file, so when compiling on SymbOS, we can reduce compilation time and memory usage by only including the sub-headers that are actually needed (e.g., `symbolos/shell.h` or `symbolos/windows.h`). See the subsection introductions for what functions are in which sub-header.

These headers are not 100% comprehensive; SymbOS provides some additional system calls not implemented in `symbolos.h`, mainly low-level calls for dealing with storage devices, system configuration, and complicated applications that alter system functionality or execute code in multiple banks (SCC is not well-suited for this). These calls are discussed in the [SymbOS developer documentation](#). It is assumed that, if you need these calls, you are probably already doing something complicated enough that a few extra wrapper functions won't be useful.

Subsections

- [System variables](#)
- [Kernel routines](#)
 - [Messaging](#)
 - [Memory management](#)
 - [Memory read/write](#)
 - [System status](#)
- [Shell routines](#)
 - [Interacting with SymShell](#)
 - [Shell functions](#)
 - [Shell control codes](#)
- [Window routines](#)
 - [Window management](#)
 - [Window status](#)
 - [Control reference](#)
 - [Event reference](#)
- [Desktop features](#)
 - [Popup dialogs](#)
 - [Context menus](#)
 - [Rubber band select](#)
 - [System tray](#)
 - [Clipboard](#)
- [Filesystem routines](#)
 - [File access](#)
 - [Directory access](#)
- [Multitasking routines](#)
 - [Processes](#)

- [Timers](#)
 - [Multithreading](#)
- [Device routines](#)
 - [Screen status](#)
 - [Mouse status](#)
 - [Keyboard status](#)
 - [Time functions](#)
 - [System configuration](#)
- [Sound routines](#)
 - [Creating/getting sounds](#)
 - [Sound functions](#)
- [Printer routines](#)
 - [Printing via the daemon](#)
 - [Printing via PrintIt](#)
 - [Direct printer functions](#)
- [Reference tables](#)
 - [Keyboard scancodes](#)
 - [Keyboard ASCII codes](#)
 - [Colors](#)
 - [Error codes](#)

Related libraries

- [Graphics library](#)
- [Network library](#)

System variables

```
char* _symmsg;
```

A 14-byte buffer for sending messages. This is used internally by most system calls, but can be used for our own purposes when manually sending messages with `Msg_Send()` and similar functions.

Several other global variables and constants contain useful information about the system:

```
unsigned char _sympid;           // process ID of the current app
unsigned char _symappid;        // application ID of the current app
unsigned char _symlink;         // main bank number of the current app
unsigned short _symversion;     // SymbOS version, as major/minor decimal (i.e., SymbOS 3.1 = 31)
char* _segcode;                // start address of the code segment + 0x100
char* _segdata;                // start address of the data segment
char* _segtrans;               // start address of the transfer segment
unsigned short _segcodelen;    // length of the code segment
unsigned short _segdatalen;    // length of the data segment
unsigned short _segtranslen;   // length of the transfer segment
```


(The other contents of the application header can be accessed directly with the struct `_symheader`, not documented here; see definition in `symsos/header.h`.)

Kernel routines

In addition to `symbol.h`, these functions can be found in `symbol/core.h`.

Contents

- [Messaging](#)
- [Memory management](#)
- [Memory read/write](#)
- [System status](#)

Messaging

Msg_Send()

```
unsigned char Msg_Send(char rec_pid, char send_pid, char* msg);
```

Sends the message in `*msg` to process ID `send_pid`. `rec_pid` is the process ID that should receive the response, if any; usually this should be our own process ID (`_sympid`). `*msg` must be in the **transfer** segment.

Return value: 0 = message queue is full; 1 = message sent successfully; 2 = receiver process does not exist.

SymbOS name: `Message_Send (MSGSEND)`.

Msg_Receive()

```
unsigned short Msg_Receive(char rec_pid, char send_pid, char* msg);
```

Checks for a message sent from process ID `send_pid` to process ID `rec_pid` and, if one is waiting, stores it in `*msg`. Usually `rec_pid` should be our own process ID (`_sympid`). If `send_pid` is -1, checks for messages from any process. `*msg` must be in the **transfer** segment.

Return value: Low byte: 0 = no message available, 1 = message received. High byte: sender process ID. Extract with, e.g.,

```
rec = result & 0xFF;
pid = result >> 8;
```

SymbOS name: `Message_Receive (MSGGET)`.

Msg_Sleep()

```
unsigned short Msg_Sleep(char rec_pid, char send_pid, char* msg);
```

Checks for a message sent from process ID `send_pid` to process ID `rec_pid`. If one is waiting, stores it in `*msg`. If there is no message, returns CPU time to SymbOS and waits until a message is available or the process is woken up for another reason. Usually `rec_pid` should be our own process ID (`_sympid`). If `send_pid` is -1, checks for messages from any process. `*msg` must be in the **transfer** segment.

Return value: Low byte: 0 = no message available, 1 = message received. High byte: sender process ID. Extract with, e.g.,

```
rec = result & 0xFF;
pid = result >> 8;
```

Note that processes can be “woken up” for multiple reasons, so returning from `Msg_Sleep()` does not necessarily mean that the desired message has been received. We must check the return value or the contents of `*msg` to be sure. For example, to loop until a message is actually received:

```
while (!(Msg_Sleep(_sympid, -1, _symmsg) & 0x01));
```

SymbOS name: `Message_Sleep_And_Receive (MSGSLP)`.

Msg_Wait()

Currently only available in development builds of SCC.

```
unsigned char Msg_Wait(char rec_pid, char send_pid, char* msg, char id);
```

A utility function that idles on `Msg_Sleep()` until a message from process ID `send_pid` with the first byte `msg[0] = id` arrives. Any other messages received in the meantime will remain on the queue. (This is useful for command/response pairs where we want to send a message to a service and wait for its response.)

Return value: sender process ID.

Idle()

```
void Idle(void);
```

Return CPU time to SymbOS and idle until something wakes it up—for example, an incoming message.

SymbOS name: `Multitasking_SoftInterrupt (RST #30)`.

Memory management

Applications are able to address more than 64KB of memory by reserving additional blocks of banked memory. These blocks cannot be addressed directly using C pointers and variables, but we can read/write/copy data to them using system functions.

In addition to `symbols.h`, these functions can be found in `symbols/memory.h`.

Mem_Reserve()

```
unsigned char Mem_Reserve(unsigned char bank, unsigned char type, unsigned short len,
                          unsigned char* bankVar, char** addrVar);
```

Reserve a block of banked memory in bank `bank` of length `len`, in bytes. `bank` may be from 0 to 15; 0 means “any bank can be used.” `type` may be one of: 0 = located anywhere; 1 = reserve within a

16KB address block (like the **data** segment); 2 = reserve within the last 16KB address block (like the **transfer** segment).

Two variables must be passed by reference to store the address of the resulting block of banked memory: `bankVar` (type `unsigned char`), which stores the bank, and `addrVar` (type `char*`), which stores the address.

Note that, to avoid memory leaks, memory reserved with `Mem_Reserve()` must be manually released with `Mem_Release()` before program exit!**** SymbOS does not have the resources to track this automatically; it is up to us.

Return value: 0 = success, 1 = out of memory.

SymbOS name: `Memory_Get` (MEMGET).

Mem_Release()

```
void Mem_Release(unsigned char bank, char* addr, unsigned short len);
```

Releases a block of banked memory previously reserved with `Mem_Reserve()`. `bank` is the bank of the reserved memory, which must be from 1 to 15; `addr` is the address; and `len` is the length of the reserved block, in bytes.

Be careful to ensure that `bank`, `addr`, and `len` exactly match a contiguous block of memory that was previously reserved with `Mem_Reserve()`! SymbOS does not keep track of this independently, so we can corrupt memory if we pass invalid information.

SymbOS name: `Memory_Free` (MEMFRE).

Mem_Resize()

```
unsigned char Mem_ResizeX(unsigned char bank, unsigned char type, char* addr,
                          unsigned short oldlen, unsigned short newlen,
                          unsigned char* bankVar, char** addrVar);
```

Attempts to resize a block of banked memory previously reserved with `Mem_Reserve()`. (This is accomplished manually by reserving a new block of the desired size, copying the old block to the new block, and releasing the old block.) `bank` is the bank of the reserved memory, which must be from 1 to 15; `addr` is the address; `oldlen` is the previous length of the reserved block, in bytes; and `newlen` is the requested new length, in bytes.

Two variables must be passed by reference to store the address of the resulting block: `bankVar` (type `unsigned char`), which stores the bank, and `addrVar` (type `char*`), which stores the address. Note that the new location of the block may be in any bank, not just the same bank as the previous block.

Return value: 0 = success, 1 = out of memory.

Mem_Longest()

```
unsigned short Mem_Longest(unsigned char bank, unsigned char type);
```

Returns (in bytes) the longest area of contiguous memory within bank bank that could be reserved with Mem_Reserve(). bank may be from 0 to 15; 0 means “any bank can be used.” type may be one of: 0 = located anywhere; 1 = reserve within a 16KB address block (like the **data** segment); 2 = reserve within the last 16KB address block (like the **transfer** segment).

SymbOS name: Memory_Information (MEMINF).

Mem_Free()

```
unsigned long Mem_Free(void);
```

Returns the total amount of free memory, in bytes.

SymbOS name: Memory_Summary (MEMSUM).

Mem_Banks()

```
unsigned char Mem_Banks(void);
```

Returns the total number of existing 64KB extended RAM banks.

SymbOS name: Memory_Summary (MEMSUM).

Memory read/write

In addition to `symbol.h`, these functions can be found in `symbol/memory.h`.

Bank_ReadWord()

```
unsigned short Bank_ReadWord(unsigned char bank, char* addr);
```

Returns the two-byte word at bank bank, address addr. bank must be from 1 to 15.

SymbOS name: Banking_ReadWord (BNKRWD).

Bank_WriteWord()

```
void Bank_WriteWord(unsigned char bank, char* addr, unsigned short val);
```

Writes the two-byte word val to memory at bank bank, address addr. bank must be from 1 to 15.

SymbOS name: Banking_WriteWord (BNKWWD).

Bank_ReadByte()

```
unsigned char Bank_ReadByte(unsigned char bank, char* addr);
```

Returns the byte at bank bank, address addr. bank must be from 1 to 15.

SymbOS name: Banking_ReadByte (BNKRBT).

Bank_WriteByte()

```
void Bank_WriteByte(unsigned char bank, char* addr, unsigned char val);
```

Writes the byte `val` to memory at bank `bank`, address `addr`. `bank` must be from 1 to 15.

SymbOS name: Banking_WriteByte (BNKWBt).

Bank_Copy()

```
void Bank_Copy(unsigned char bankDst, char* addrDst,  
               unsigned char bankSrc, char* addrSrc, unsigned short len);
```

Copies `len` bytes of memory from bank `bankSrc`, address `addrSrc` to bank `bankDst`, address `addrDst`.

SymbOS name: Banking_Copy (BNKCOP).

Bank_Get()

```
unsigned char Bank_Get(void);
```

Returns the bank number in which the app's main process is running. (Normally it is easier to use the `_symlink` global for this purpose.)

SymbOS name: Banking_GetBank (BNKGET).

Bank-Decompress()

```
void Bank-Decompress(unsigned char bank, char* addrDst, char* addrSrc);
```

Decompresses the compressed data block located at bank `bank`, address `addrSrc` into memory at bank `bank`, address `addrDst`. The addresses must be arranged such that the last address of the decompressed data will be the same as the last address of the original compressed data. That is, we need to know the length of the uncompressed data ahead of time, and load the compressed data into the end of this buffer, with `addrSrc = addrDest + (uncompressed length) - (compressed length)`. The data will then be decompressed "in place" to fill the buffer completely from start to finish.

The structure of a compressed data block is as follows:

- 2 bytes (unsigned short): length of the block, minus these two bytes
- 4 bytes: the last four bytes of the data (uncompressed)
- 2 bytes (unsigned short): the number of uncompressed bytes before the compressed data begins (e.g., for metadata; usually 0)
- (some amount of uncompressed data, or nothing)
- (some amount of data compressed using the [ZX0 algorithm](#), minus the last four bytes given above)

This function is only available in SymbOS 4.0 and higher.

SymbOS name: Banking-Decompress (BNKCPR).

System status

Sys_Counter()

```
unsigned long Sys_Counter(void);
```

Returns the system counter, which increments 50 times per second. This can be used to calculate time elapsed for regulating framerates in games, etc.

SymbOS name: Multitasking_GetCounter (MTGCNT).

Sys_Counter16()

```
unsigned short Sys_Counter16(void);
```

Equivalent to `Sys_Counter()`, but returns only the low 16 bits of the system counter. (This wraps around every 22 minutes, but can be useful when [multithreading](#) because **long** is not thread-safe.)

SymbOS name: Multitasking_GetCounter (MTGCNT).

Sys_IdleCount()

```
unsigned short Sys_IdleCount(void);
```

Returns the idle process counter, which increments every 64 microseconds. This can be used to calculate CPU usage.

SymbOS name: Multitasking_GetCounter (MTGCNT).

Shell functions

For simple shell I/O, we can just use standard C stdio functions (`printf()`, etc.). However, for more direct control (or to avoid the overhead of importing stdio), we can interact directly with SymShell using the functions described below.

SymShell functions will only be available if the application is associated with a running instance of SymShell. To ensure that an application is started in SymShell, make sure that it has the file extension `.com` instead of `.exe`.

In addition to `symbolos.h`, these functions can be found in `symbolos/shell.h`.

Contents

- [Interacting with SymShell](#)
- [Shell functions](#)
- [Shell control codes](#)

Interacting with SymShell

Several globals provide useful information about the SymShell instance:

```
unsigned char _shellpid;    // SymShell process ID
unsigned char _shellwidth; // console width, in characters
unsigned char _shellheight; // console height, in characters
unsigned char _shellver;    // SymShell version
unsigned char _shellerr;    // error code of last shell command
```

If `_shellpid = 0`, there is no SymShell instance. `_shellver` is a two-digit number where the tens digit is the major version and the ones digit is the minor version, e.g., 21 = 2.1.

Most shell functions allow specifying a *channel*. In general, channel 0 is the standard input/output, which is usually the keyboard (in) and text window (out) but may also be a file or stream if some type of redirection is active. This is similar to the behavior of `stdin/stdout` in standard C (although note that there is no direct equivalent to `stderr`). Channel 1 is always the physical keyboard (in) or text window (out), even if redirection is active on channel 0. Usually we want channel 0.

Note that SymShell returns the Windows-style ASCII character 13 (`\r`) for the “Enter” key, *not* the Unix-style ASCII character 10 (`\n`), as is more common in C. Likewise, when sending text to the console, note that SymShell expects the Windows-style line terminator `\r\n` rather than the Unix-style `\n` that is more common in C. If we only send `\n`, SymShell will take this literally, only performing a line feed (`\n`, going down a line) but not a carriage return (`\r`, going back to the start of the next line)! The stdio implementation (`printf()`, etc.) includes some logic to paper over these differences and understand the Unix-style convention, but when working with SymShell functions directly, we will need to be more careful.

Most shell functions will set `_shellerr` to an error code on failure; see [error codes](#).

A note on exiting

When an app's host SymShell instance is forcibly closed by clicking the "X" button, SymShell will send message 0 to the app (i.e., `msg[0] = 0`) to tell it to exit. It is important that the app obey this message and exit, or else it will end up "orphaned", running in the background and forever waiting for responses from a SymShell window that no longer exists.

The default behavior of the shell routines below is to watch for message 0 and call `exit(0)` to quit immediately if detected. This is usually what we want and does not require any special consideration of message 0 on our part. However, if the app needs to shut down more gracefully (e.g., by calling `Mem_Release()` to free allocated memory), we can override this behavior by setting the global variable `_shellexit = 1` at the beginning of `main()`. When `_shellexit` is nonzero, shell functions will instead respond to message 0 by setting `_shellexit = 2` and behaving as if there is no available shell (i.e., `_shellerr = ERR_NOSHELL`). We can then watch for `_shellexit = 2` and perform any necessary shutdown manually.

Shell functions

Shell_CharIn()

```
int Shell_CharIn(unsigned char channel);
```

Requests an input character from the specified `channel`. If this is the console keyboard and there is no character waiting in the keybuffer, SymShell will pause until the user presses a key.

Return value: On success, returns the ASCII value of the character (including [extended ASCII codes](#) for special keys). If we have hit EOF on an input stream, returns -1. If another error has occurred, returns -2 and sets `_shellerr`.

SymbOS name: `SymShell_CharInput_Command (MSC_SHL_CHRINP)`.

Shell_CharOut()

```
signed char Shell_CharOut(unsigned char channel, unsigned char val);
```

Sends ASCII character `val` to the specified `channel`.

While this is the standard way to output a single character to the console, note that outputting long strings by repeatedly calling `Shell_CharOut()` will be very slow, because for every character sent, SymShell must (1) receive the message, (2) redraw the screen, and (3) send a response message. Sending a single longer string with `Shell_StringOut()` only requires one set of messages and one redraw and is therefore much more efficient.

Return value: On success, returns 0. If another error has occurred, returns -2 and sets `_shellerr`.

SymbOS name: `SymShell_CharOutput_Command (MSC_SHL_CHROUT)`.

Shell_CharTest()

```
int Shell_CharTest(unsigned char channel, unsigned char lookahead);
```

Behaves like `Shell_CharIn()`, except that if there is no character waiting in the keybuffer, it will return 0 immediately without waiting for input. If `lookahead = 0`, any character found will be returned but left in the keybuffer; if `lookahead = 1`, the character will be returned and removed from the keybuffer.

This function requires SymShell 2.3 or greater and will always return 0 on earlier versions. This function currently only works for physical keyboard input, not redirected streams.

Return value: If a key is waiting, returns the ASCII value of the character (including [extended ASCII codes](#) for special keys). If no key is waiting, returns 0. If another error has occurred, returns -2 and sets `_shellerr`.

SymbOS name: `SymShell_CharTest_Command (MSC_SHL_CHRTST)`.

Shell_StringIn()

```
signed char Shell_StringIn(unsigned char channel, unsigned char bank, char* addr);
```

Requests a line of input from the specified `channel`, terminated by the Enter key. If this is the console keyboard, SymShell will pause and accept input until the user presses Enter. The input will be written to memory (zero-terminated) at bank `bank`, address `addr`.

Input may be up to 255 characters in length, plus a zero-terminator, so the write buffer should always be at least 256 bytes long.

Return value: On success, returns 0. If we have hit EOF on an input stream, returns -1. If another error has occurred, returns -2 and sets `_shellerr`.

SymbOS name: `SymShell_StringInput_Command (MSC_SHL_STRINP)`.

Shell_StringOut()

```
signed char Shell_StringOut(unsigned char channel, unsigned char bank,  
                           char* addr, unsigned char len);
```

Sends the string at bank `bank`, address `addr` to the specified `channel`. The string can be up to 255 bytes long and must be zero-terminated. `len` must contain the length of the string (without the zero-terminator); the `string.h` function `strlen()` is a good way to determine this.

Return value: On success, returns 0. If another error has occurred, returns -2 and sets `_shellerr`.

SymbOS name: `SymShell_StringOutput_Command (MSC_SHL_STROUT)`.

Shell_Print()

```
signed char Shell_Print(char* addr);
```

A convenience function that calls `Shell_StringOut(0, _symbank, addr, strlen(addr))`. This saves some bytes and hassle from the most common use-case for `Shell_StringOut()`.

Return value: On success, returns 0. If another error has occurred, returns -2 and sets `_shellerr`.

Shell_Locate()

Currently only available in development builds of SCC.

```
signed char Shell_Locate(unsigned char col, unsigned char row);
```

Set the cursor to column `col`, row `row`. (Internally, this just calls `Shell_Print()` with the appropriate [control codes](#).) Row and column numbers start at 1.

Return value: On success, returns 0. If another error has occurred, returns -2 and sets `_shellerr`.

Shell_Exit()

```
void Shell_Exit(unsigned char type);
```

Informs SymShell that the app is closing down, so it can stop waiting for input/output messages from the app. (Normally this is handled automatically by `exit()`, but we can also do it manually.) If `type = 0`, the app is exiting normally and should be unregistered with SymShell. If `type = 1`, the app is going into “blur” mode: it is still running in the background, but no longer plans to output anything to the shell.

SymbOS name: `SymShell_Exit_Command (MSC_SHL_EXIT)`.

Shell_PathAdd()

```
void Shell_PathAdd(unsigned char bank, char* path, char* addition, char* dest);
```

A utility function that constructs an absolute file path from a base path (at bank `bank`, address `path`) and a relative path addition (at bank `bank`, address `addition`), storing the result in bank `bank`, address `dest`. This is mainly used to turn relative paths into absolute paths for the file manager functions.

Any relative path elements in the addition (`..`, `\`, etc.) will be resolved. If `path = 0`, the absolute path will be relative to the current shell path (i.e., the path set by the `CD` command). The base path should not end with a slash or backslash.

Examples:

```
char abspath[256];
```

```
Shell_PathAdd(_symlink, "C:\SYMBOS\APPS", "..\MUSIC\MP3\LALALA.MP3", abspath);  
// yields: C:\SYMBOS\MUSIC\MP3\LALALA.MP3
```

```
Shell_PathAdd(_symlink, "A:\GRAPHICS\NATURE", "\SYMBOS", abspath);  
// yields: A:\SYMBOS
```

```
Shell_PathAdd(_symlink, "C:\ARCHIVE", "*.ZIP", abspath);  
// yields: C:\ARCHIVE\*.ZIP
```

```
Shell_PathAdd(_symlink, "A:\ARCHIVE", "C:\SYMBOS", abspath);  
// yields: C:\SYMBOS
```

SymbOS name: SymShell_PathAdd_Command (MSC_SHL_PTHADD).

Shell_CharWatch()

```
signed char Shell_CharWatch(unsigned char bank, char* addr);
```

Creates a “character watch byte” that we can read directly to determine if there is keyboard input pending in SymShell, without the overhead of repeatedly calling Shell_CharTest(). The watch byte will be established at bank *bank*, address *addr*. As soon as a key is pressed, its value will be written to the byte; if the keyboard buffer is empty, 0 will be written. We can then call Shell_CharIn() to read the key from the buffer.

This function requires SymShell 2.3 or greater and will fail with `_shellerr = ERR_RINGFULL` on earlier versions.

Return value: On success, returns 0. On failure, returns -2 and sets `_shellerr`.

SymbOS name: SymShell_CharWatch_Command (MSC_SHL_CHRWTC).

Shell_StopWatch()

```
signed char Shell_StopWatch(unsigned char bank, char* addr);
```

Deactivates the “character watch byte” at bank *bank*, address *addr*, which was previously established using Shell_CharWatch().

This function requires SymShell 2.3 or greater and will fail with `_shellerr = ERR_RINGFULL` on earlier versions.

Return value: On success, returns 0. On failure, returns -2 and sets `_shellerr`.

SymbOS name: SymShell_CharWatch_Command (MSC_SHL_CHRWTC).

Shell control codes

SymShell interprets ASCII characters 1-31 as control characters; these are summarized below.

Sequence	Meaning
0x02	Switch cursor off
0x03	Switch cursor on
0x04	Save current cursor position
0x05	Restore saved cursor position
0x06	Reactivate text output (see 0x15)
0x08	Move cursor one character to the right
0x09	Move cursor one character to the left
0x0A	Move cursor one character down
0x0B	Move cursor one character up
0x0C	Clear screen and place cursor at 1,1

Sequence	Meaning
0x0D	Carriage return (move cursor to start of line)
0x0E n	Move cursor multiple characters.(1)
0x10	Clears the character under the cursor.
0x11	Clear line from cursor left.
0x12	Clear line from cursor right.
0x13	Clear column from cursor up.
0x14	Clear column from cursor down.
0x15	Deactivate text output (see 0x06)
0x16	Set a tab at the current cursor column
0x17	Clear a tab at the current cursor column
0x18	Clear all tabs
0x19	Jump to next tab
0x1C c r	Resize the terminal window to c columns by r rows.(2)
0x1D 0x01	Scroll window up one line, without affecting cursor
0x1D 0x02	Scroll window down one line, without affecting cursor
0x1E	Reset cursor to 1,1
0x1F c r	Move cursor to column c, row r

(1) Characters to move is specified as follows:

- n = 1 to 80 = move n characters right
- n = 81 to 160 = move (n - 80) characters left
- n = 161 to 185 = 'move (n' - 160) characters down
- n = 186 to 210 = move (n - 185) characters up.

The cursor will not cross any borders.

(2) Maximum terminal size is 80x25 (or 80x24 on MSX). Depending on platform and context, this command may not do anything.

Window routines

In addition to `symbol.h`, these functions can be found in `symbol/windows.h`.

Contents

- [Window management](#)
- [Window status](#)
- [Clipboard](#)
- [System tray](#)

Window management

Win_Open()

```
signed char Win_Open(unsigned char bank, void* addr);
```

Opens the window whose data record (a struct of type `Window`) is at bank `bank`, address `addr`. This must be in the **transfer** segment.

Return value: On success, returns the window ID. On failure, returns -1.

SymbOS name: `Window_Open_Command (MSC_DSK_WINOPN)`.

Win_Close()

```
void Win_Close(unsigned char winID);
```

Closes the window with the ID `winID`.

SymbOS name: `Window_Close_Command (MSC_DSK_WINCLS)`.

Win_Redraw()

```
void Win_Redraw(unsigned char winID, signed char what, unsigned char first);
```

Redraws one or more controls in the main window content of window `winID`. `what` can be either (1) -1, meaning “redraw entire window content”; (2) the control ID of a single control to redraw; or (3) a negative number (from -2 to -16) indicating how many controls should be redrawn (from 2 to 16), in which case `first` indicates the control ID of the first control to redraw.

For performance reasons the window will only actually be redrawn if it has focus. To force redraw of the window even when it does not have focus, see `Win_Redraw_Ext()`.

SymbOS name: `Window_Redraw_Content_Command (MSC_DSK_WININH)`.

Win_Redraw_Ext()

```
void Win_Redraw_Ext(unsigned char winID, signed char what, unsigned char first);
```

Equivalent to `Win_Redraw()`, but redraws controls whether or not the window has focus. (This is slightly slower than `Win_Redraw()` because SymbOS must check for window overlap.)

SymbOS name: `Window_Redraw_ContentExtended_Command` (`MSC_DSK_WINDIN`).

Win_Redraw_Area()

```
void Win_Redraw_Area(unsigned char winID, unsigned char what, unsigned char first,  
                    unsigned short x, unsigned short y, unsigned short w, unsigned short h);
```

Equivalent to `Win_Redraw()`, but only redraws controls within the box specified by the upper left coordinates `x` and `y`, width `w`, and height `h` (in pixels). This command is particularly useful for redrawing only part of a large graphic area (such as a game playfield), since it is much faster than redrawing the entire area.

Note that these coordinates are relative to the window content, including any scroll. Note that the behavior for resizable windows changed in SymbOS 4.0 to be more consistent with non-resizable windows; in prior versions, the calculation was `x = xpos - form1.xscroll + 8` (i.e., offset by 8 and not including scroll).

SymbOS name: `Window_Redraw_ContentArea_Command` (`MSC_DSK_WINPIN`).

Win_Redraw_Toolbar()

```
void Win_Redraw_Toolbar(unsigned char winID, signed char what, unsigned char first);
```

Equivalent to `Win_Redraw()`, but redraws controls in the window's toolbar control group instead of main window content.

SymbOS name: `Window_Redraw_Toolbar_Command` (`MSC_DSK_WINTOL`).

Win_Redraw_Menu()

```
void Win_Redraw_Menu(unsigned char winID);
```

If the window has focus, redraws the menu of window `winID`.

SymbOS name: `Window_Redraw_Menu_Command` (`MSC_DSK_WINMEN`).

Win_Redraw_Title()

```
void Win_Redraw_Title(unsigned char winID);
```

If the window has focus, redraws the titlebar of window `winID`.

SymbOS name: `Window_Redraw_Title_Command` (`MSC_DSK_WINTIT`).

Win_Redraw_Status()

```
void Win_Redraw_Status(unsigned char winID);
```

If the window has focus, redraws the statusbar of window `winID`.

SymbOS name: `Window_Redraw_Status_Command (MSC_DSK_WINSTA)`.

Win_Redraw_Slider()

```
void Win_Redraw_Slider(unsigned char winID);
```

If the window has focus and is resizable, redraws the main content area's scrollbars on window `winID`.

SymbOS name: `Window_Redraw_Status_Command (MSC_DSK_WINSTA)`.

Win_Redraw_Sub()

```
void Win_Redraw_Sub(unsigned char winID, unsigned char collection, unsigned char control);
```

Redraws the single control with the ID `control` inside the control collection with the ID `collection` on window `winID`.

SymbOS name: `Window_Redraw_SubControl_Command (MSC_DSK_WINSIN)`.

Win_ContentX()

```
void Win_ContentX(unsigned char winID, unsigned short newX);
```

Changes the horizontal scroll position (`.xscroll`) of window `winID` to `newX` pixels. If the window has focus, the relevant portions of the content will be redraw.

This command is faster than updating the `.xscroll` record of the window and redrawing manually because only the newly visible portion of the window will be redrawn from scratch. The scroll position can be changed even if the window is not resizable by the user.

SymbOS name: `Window_Set_ContentX_Command (MSC_DSK_WINMVX)`.

Win_ContentY()

```
void Win_ContentY(unsigned char winID, unsigned short newY);
```

Changes the vertical scroll position (`.yscroll`) of window `winID` to `newY` pixels. If the window has focus, the relevant portions of the content will be redraw.

This command is faster than updating the `.yscroll` record of the window and redrawing manually because only the newly visible portion of the window will be redrawn from scratch. The scroll position can be changed even if the window is not resizable by the user.

SymbOS name: `Window_Set_ContentY_Command (MSC_DSK_WINMVY)`.

Win_Focus()

```
void Win_Focus(unsigned char winID);
```


Gives the window `winID` focus, bringing it to the front of the desktop.

SymbOS name: `Window_Focus_Command (MSC_DSK_WINTOP)`.

Win_Maximize()

```
void Win_Maximize(unsigned char winID);
```

Maximizes the window `winID`.

SymbOS name: `Window_Size_Maximize_Command (MSC_DSK_WINMAX)`.

Win_Minimize()

```
void Win_Minimize(unsigned char winID);
```

Minimizes the window `winID`.

SymbOS name: `Window_Size_Minimize_Command (MSC_DSK_WINMIN)`.

Win_Restore()

```
void Win_Restore(unsigned char winID);
```

Restores the window `winID` to normal size, if maximized.

SymbOS name: `Window_Size_Restore_Command (MSC_DSK_WINMID)`.

Win_Move()

```
void Win_Move(unsigned char winID, unsigned short newX, unsigned short newY);
```

If the window is not maximized, moves the window `winID` to horizontal position `newX`, vertical position `newY` on the screen (in pixels).

SymbOS name: `Window_Set_Position_Command (MSC_DSK_WINMOV)`.

Win_Resize()

```
void Win_Resize(unsigned char winID, unsigned short newW, unsigned short newH);
```

Resizes the window `winID` so that the main content of the window has the width `newW` and the height `newH` (in pixels). Note that this is the size of the main *content*, not the size of the entire window; the titlebar, menubar, toolbar, statusbar, and scrollbars (if any) may add additional size.

SymbOS name: `Window_Set_Size_Command (MSC_DSK_WINSIZ)`.

Win_Width()

```
unsigned short Win_Width(Window* win);
```

A utility function that returns the width of the visible content area of the window `win`, in pixels. (Note that the window is passed as the address of the relevant `Window` record, *not* as the window ID!)

When determining the visible size of a resizable window, this function should be used instead of directly reading the `Window.w` record. This is because the `Window.w` record contains the width the window “wants” to be, not necessarily its true current size; for example, when a window is maximized, `Window.w` will be the original “restored” width rather than the true “maximized” width. This function handles all the necessary calculations for determining the true width automatically.

Win_Height()

```
unsigned short Win_Height(Window* win);
```

A utility function that returns the height of the visible content area of the window `win`, in pixels. (Note that the window is passed as the address of the relevant `Window` record, *not* as the window ID!)

When determining the visible size of a resizable window, this function should be used instead of directly reading the `Window.h` record. This is because the `Window.h` record contains the height the window “wants” to be, not necessarily its true current size; for example, when a window is maximized, `Window.h` will contain the original “restored” height rather than the true “maximized” height. This function handles all the necessary calculations for determining the true height automatically.

Window status

Win_X()

Currently only available in development builds of SCC.

```
int Win_X(Window* win);
```

A utility function that returns the absolute screen X position of the visible content area of the window `win`, in pixels. (Note that the window is passed as the address of the relevant `Window` record, *not* as the window ID!)

This is mainly useful for translating between absolute and relative screen position when using functions such as [Select_Pos\(\)](#). This function is more reliable than simply reading the `Window.x` record because it accounts for how SymbOS handles maximized windows.

Win_Y()

Currently only available in development builds of SCC.

```
int Win_Y(Window* win);
```

A utility function that returns the absolute screen Y position of the visible content area of the window `win`, in pixels. (Note that the window is passed as the address of the relevant `Window` record, *not* as the window ID!)

This is mainly useful for translating between absolute and relative screen position when using functions such as [Select_Pos\(\)](#). This function is more reliable than simply reading the `Window.y` record

because it accounts for how SymbOS lays out the titlebar, menubar, and toolbar, as well as maximized windows.

TextBox_Pos()

```
unsigned char TextBox_Pos(Window* win, Ctrl* textbox);
```

Requests the current cursor position of a multiline textbox control as a column and row (passed as a pointer to its `Ctrl` struct with the parameter `textbox`). For internal reasons, this command will only work if the textbox currently has keyboard focus, and we must also pass a pointer to its host window's `Window` struct as the parameter `win`.

Return value: On success, returns 1, and the cursor column and line (starting at 0) will be loaded into the `column` and `line` properties of the textbox's `Ctrl_TextBox` struct. On failure, returns 0. (Yes, this is convoluted. If we just need the cursor position in bytes from the start of the file, we can examine the cursor position of the textbox's `Ctrl_TextBox` struct directly.)

SymbOS name: `KEYPUT 29`.

TextBox_Redraw()

```
void TextBox_Redraw(Window* win, Ctrl* textbox);
```

Tells the specified multi-line textbox (passed as a pointer to its `Ctrl` struct with the parameter `textbox`) that its content has been modified and it should reformat and redraw its text. For internal reasons, this command will only work if the textbox currently has keyboard focus, and we must also pass a pointer to its host window's `Window` struct as the parameter `win`.

SymbOS name: `KEYPUT 30`.

TextBox_Select()

```
void TextBox_Select(Window* win, Ctrl* textbox, int cursor, int selection);
```

Updates the cursor position and selection of the specified multi-line textbox (passed as a pointer to its `Ctrl` struct with the parameter `textbox`), scrolling and redrawing the textbox as necessary. `cursor` is the new cursor position, in bytes from the start of the complete text; `selection` is the number of characters to select (0 for none, greater than 0 for the cursor to mark the start of the selection, and less than 0 for the cursor to mark the end of the selection). For internal reasons, this command will only work if the textbox currently has keyboard focus, and we must also pass a pointer to its host window's `Window` struct with the parameter `win`.

SymbOS name: `KEYPUT 31`.

Desktop features

Contents

- [Popup dialogs](#)
- [Context menus](#)
- [Rubber band select](#)
- [Clipboard\[#clipboard\]](#)
- [System tray](#)

Popup dialogs

In addition to `symbols.h`, these functions can be found in `symbols/popups.h`.

MsgBox()

```
unsigned char MsgBox(char* line1, char* line2, char* line3, unsigned int pen,
                    unsigned char type, char* icon, void* modalWin);
```

Opens a message box onscreen. `line1`, `line2`, and `line3` are three text strings that will be displayed. `pen` is the text color, one of `COLOR_BLACK`, `COLOR_RED`, `COLOR_ORANGE`, or `COLOR_YELLOW` (usually we want `COLOR_BLACK`). `type` is a OR'd bitmask of one or more of the following options:

- `BUTTON_OK`: display an “OK” button
- `BUTTON_YN`: display “Yes” and “No” buttons
- `BUTTON_YNC`: display “Yes”, “No”, and “Cancel” buttons
- `TITLE_DEFAULT`: title the box “Error!” if there is no custom icon, otherwise “Info”
- `TITLE_INFO`: title the box “Info”
- `TITLE_WARNING`: title the box “Warning”
- `TITLE_CONFIRM`: title the box “Confirmation”
- `MSGBOX_ICON`: use a custom icon

`icon` is the address of a 24x24 custom icon image, in 4-color SGX format. If `icon = 0`, a default icon will be used. The variable `_symicon` stores the address of the application's main 4-color icon.

`modalWin` specifies the address of a Window data record that should be declared modal, if any; this window will not be able to be focused until the message box is closed. If `modalWin = 0`, no window will be declared modal.

Note that only pure info messages (`BUTTON_OK`, not modal) can have multiple instances open on the screen at the same time. SymbOS implements more complex message boxes as a single window shared by all processes; if the message box cannot be opened because it is already in use by another process, this function will return `MSGBOX_FAILED (0)`. This function is also NOT thread-safe.

Return value: One of:

- `MSGBOX_FAILED (0)`: another process is already using the complex message box.
- `MSGBOX_OK`: the user clicked the “OK” button.
- `MSGBOX_YES`: the user clicked the “Yes” button.
- `MSGBOX_NO`: the user clicked the “No” button.
- `MSGBOX_CANCEL`: the user clicked the “Cancel” button or the window Close button.

SymbOS name: Dialogue_Infobox_Command (MSC_SYS_SYSWRN)

InputBox()

Currently only available in development builds of SCC.

```
signed char InputBox(char* title, char* line1, char* line2, char* buffer,
                    unsigned short buflen, void* modalWin);
```

Opens a text input box onscreen. `title` is the window title to display, `line1` and `line2` are two lines of text to display as the prompt, `buffer` is the buffer to store the inputted text, and `buflen` is the length of the buffer (including zero terminator, so the maximum length of the input string is `buflen - 1`). If `buffer` already contains text, it will be used as the default value of the text field.

`modalWin` specifies the address of a Window data record that should be declared modal, if any; this window will not be able to be focused until the message box is closed. If `modalWin = 0`, no window will be declared modal.

This function is thread-safe, but only one `InputBox` will be visible at a time (a semaphore is used to ensure other threads wait their turn).

Return value: 0 if the user clicked “OK” or -1 if the user clicked “Cancel”. In either case, the user’s text input will be stored in `buffer`.

FileBox()

```
unsigned char FileBox(char* path, char* filter, unsigned char flags, unsigned char attribs,
                    unsigned short entries, unsigned short bufsize, void* modalWin);
```

Opens a file selector box onscreen. `path` can be the absolute path of the directory to start in, the absolute path of a file to preselect, or 0 to start in the default location. If `path` is a directory, it should end with a backslash (\). `filter` is a three-byte string containing a file extension filter, which can contain Window-style wildcards (* or ?); for example, * to show all files or TXT to show only .txt files. (If shorter than 3 bytes, this string should be padded to 3 bytes with spaces.) ‘flags’ is an OR’d bitmask consisting of one or more of the following options:

- FILEBOX_OPEN: label box “open”
- FILEBOX_SAVE: label box “save”
- FILEBOX_FILE: select an individual file
- FILEBOX_DIR: select a directory

`attribs` is an OR’d bitmask consisting of one or more of the following options:

- ATTRIB_READONLY: do not include read-only files
- ATTRIB_HIDDEN: do not include hidden files
- ATTRIB_SYSTEM: do not include system files
- ATTRIB_VOLUME: do not include volume information files (recommended)
- ATTRIB_DIR: do not include directories
- ATTRIB_ARCHIVE: do not include archived files

`entries` specifies the maximum number of entries (recommended 128) and `bufLen` specifies the length of the directory buffer to create (recommended 8000). `modalWin` specifies the address of a `Window` data record that should be declared modal, if any; this window will not be able to be focused until the file selector is closed. If `modalWin = 0`, no window will be declared modal.

Note that SymbOS implements the file selector as a single window shared by all processes; if the file selector cannot be opened because it is already in use by another process, this function will return `FILEBOX_FAILED`.

Return value: One of:

- `FILEBOX_OK (0)`: the user has selected a file and clicked “OK”. The absolute path of the selected file or directory will be stored in the global string `FileBoxPath`.
- `FILEBOX_CANCEL`: the user canceled file selection.
- `FILEBOX_FAILED`: another process is already using the file selector.
- `FILEBOX_NOMEM`: not enough memory available to open the file selector.
- `FILEBOX_NOWIN`: no window ID is available to open the file selector.

SymbOS name: `Dialogue_FileSelector_Command (MSC_SYS_SELOPN)`

Context menus

In addition to `symbols.h`, these functions can be found in `symbols/popups.h`.

Menu_Context()

```
void Menu_Context(unsigned char bank, void* addr, int x, int y);
```

Opens a context menu at the specified `x` and `y` coordinates on the screen (in pixels). The menu data is a `Menu` struct (and associated `Menu_Entry` structs) located at bank `bank`, address `addr` in the **transfer** segment. If `x = -1`, the context menu will be opened at the current mouse position.

This function just opens the menu. Any user interactions with the menu will be passed back as a `MSR_DSK_MENCTX` message, which we should handle in our app’s main event loop:

- `msg[1] = 1` if entry clicked, 0 if menu cancelled
- `*(int*)&msg[2] = value of clicked entry`
- `msg[4] = 1` if entry had a checkmark, 0 otherwise

SymbOS name: `Menu_Context_Command (MSC_DSK_MENCTX)`.

Rubber band select

In addition to `symbols.h`, these functions can be found in `symbols/popups.h`.

Select_Pos()

```
char Select_Pos(unsigned short* x, unsigned short* y, unsigned short w, unsigned short h);
```

Opens a “rubber band” selector (dotted rectangle) at the specified absolute *x* and *y* position on the screen (in pixels), with width *w* and height *h* (in pixels). The user will be able to change the position (but not the size) of this selector by moving the mouse, until they either confirm their selection by releasing the left mouse button or cancel it by pressing ESC. (This is usually used for drag-and-drop operations triggered by first pressing the left mouse button.) The final position of the selector will be written back to the variables passed by reference in *x* and *y*.

Note that absolute screen coordinates are used, not coordinates relative to window content. To translate between the two, see [Win_X\(\)](#) and [Win_Y\(\)](#).

Return value: On successful completion, returns 1. If the user cancelled the operation by pressing ESC, returns 0.

SymbOS name: VirtualControl_Position_Command (MSC_DSK_CONPOS).

Select_Size()

```
char Select_Size(unsigned short x, unsigned short y, unsigned short* w, unsigned short* h);
```

Opens a “rubber band” selector (dotted rectangle) at the specified absolute *x* and *y* position on the screen (in pixels), with width **w* and height **h* (in pixels). The user will be able to change the size (but not the position) of this selector by moving the mouse, until they either confirm their selection by releasing the left mouse button or cancel it by pressing ESC. (This is usually used for drag-and-drop operations triggered by first pressing the left mouse button.) The final size of the selector will be written back to the variables passed by reference in *w* and *h*.

Note that absolute screen coordinates are used, not coordinates relative to window content. To translate between the two, see [Win_X\(\)](#) and [Win_Y\(\)](#).

Return value: On successful completion, returns 1. If the user cancelled the operation by pressing ESC, returns 0.

SymbOS name: VirtualControl_Size_Command (MSC_DSK_CONSIZ).

Clipboard

In addition to `symbol.h`, these functions can be found in `symbol/clip.h`.

Clip_Put()

```
unsigned char Clip_Put(unsigned char bank, char* addr, unsigned short len,  
                      unsigned char type);
```

Stores *len* bytes of data from bank *bank*, address *addr* into the system clipboard. *type* may be one of: 1 = text, 2 = extended graphic, 3 = item list, 4 = desktop icon shortcut.

Return value: 0 = success, 1 = out of memory.

SymbOS name: Clipboard_Put (BUFPUT).

Clip_Get()

```
unsigned short Clip_Get(unsigned char bank, char* addr, unsigned short len,  
                        unsigned char type);
```

Retrieves up to `len` bytes of data from the system clipboard and stores it in bank `bank`, address `addr`. type may be one of: 1 = text, 2 = extended graphic, 3 = item list, 4 = desktop icon shortcut. Data will only be retrieved if (1) the type of the data in the clipboard matches the requested type, and (2) the data length is not greater than `len`.

Return value: length of received data, in bytes.

SymbOS name: Clipboard_Get (BUFGET).

Clip_Type()

```
unsigned char Clip_Type(void);
```

Returns the type of data in the clipboard, if any (0 = empty, 1 = text, 2 = extended graphic, 3 = item list, 4 = desktop icon shortcut).

SymbOS name: Clipboard_Status (BUFSTA).

Clip_Len()

```
unsigned short Clip_Len(void);
```

Returns the length of data in the clipboard, in bytes.

SymbOS name: Clipboard_Status (BUFSTA).

System tray

In addition to `symbos.h`, these functions can be found in `symbos/systray.h`.

Systray_Add()

```
signed char Systray_Add(unsigned char bank, char* addr, unsigned char code);
```

Add the icon whose graphic is at bank `bank`, address `addr` to the system tray on the taskbar. These can be clicked by the user, which will generate an event (MSR_DSK_EVTCLK) with the reference value `code`. The icon must be an 8x8 4-color SGX graphics object.

Return value: On success, returns an icon ID, which can be used to later remove the icon with `Systray_Remove()`. If there are no more icon slots available, returns -1.

SymbOS name: SystrayIcon_Add_Command (MSC_DSK_STTADD).

Systray_Remove()

```
void Systray_Remove(unsigned char id);
```

Remove the icon with the ID `id` from the system tray.

SymbOS name: SystrayIcon_Remove_Command (MSC_DSK_STTREM).

Filesystem routines

Contents

- [File access](#)
- [Directory access](#)

File access

File functions in SymbOS work using numerical file handles. Due to system limits only 8 files can be open at one time, so it is easy to run out of file handles if we are not careful. **Be careful to close any files you open before exiting!** Files opened with `stdio.h` functions (`fopen()`, etc.) will be closed automatically on exit, but files opened with direct system calls will not.

SymbOS does not make a distinction between opening a file for reading or writing.

All file paths given to system calls must be absolute, e.g., `A:\DIR\FILE.TXT`, not `FILE.TXT`. For a convenient way to convert relative paths into absolute paths, see [Dir_PathAdd\(\)](#).

A quirk to know: Due to a limitation of the AMSDOS filesystem used by, e.g., CPC floppy disks, files on this filesystem only report their length to the nearest 128-byte block. The convention is to terminate the actual file with the AMSDOS EOF character (0x1A) and then pad out the file to the nearest 128-byte boundary with garbage.

As usual, the `stdio.h` implementation includes some logic to paper over this issue when using functions such as `fgets()`, but when accessing files directly with system calls we will need to keep this in mind. Files stored on FAT filesystems (i.e., most mass storage devices) do not have this limitation, although any files copied from an AMSDOS filesystem to a FAT filesystem may retain their garbage padding at the end.

Most file operations will set `_fileerr` to an error code on failure; see [error codes](#).

In addition to `symbolos.h`, these functions can be found in `symbolos/file.h`.

File_Open()

```
unsigned char File_Open(unsigned char bank, char* path);
```

Opens the file at the absolute path stored in bank `bank`, address `path` and returns the system file handle. The seek position will be set to the start of the file.

Return value: On success, returns the file handle (0 to 7). On failure, sets and returns `_fileerr` (which will always be greater than 7).

SymbOS name: `File_Open (FILOPN)`.

File_New()

```
unsigned char File_New(unsigned char bank, char* path, unsigned char attrib);
```

Creates a new file at the absolute path stored in bank bank, address path and returns the system file handle. If the file already existed, it will be emptied. The seek position will be set to the start of the file.

attrib is an OR'd list of any or none of the following: ATTRIB_READONLY, ATTRIB_HIDDEN, ATTRIB_SYSTEM, ATTRIB_ARCHIVE.

Return value: On success, returns the file handle (0 to 7). On failure, sets and returns _fileerr (which will always be greater than 7).

SymbOS name: File_New (FILNEW).

File_Close()

```
unsigned char File_Close(unsigned char id);
```

Closes the file handle id.

Return value: On success, returns 0. On failure, sets and returns _fileerr.

SymbOS name: File_Close (FILCLO).

File_Read()

```
unsigned short File_Read(unsigned char id, unsigned char bank,  
                        char* addr, unsigned short len);
```

Reads len bytes from the open file handle id into a buffer at bank bank, address addr. The seek position will be moved forward by len bytes, so the next call to File_Read() will read the next part of the file.

Return value: On success, returns the number of bytes read. On failure, sets _fileerr and returns 0.

SymbOS name: File_Input (FILINP).

File_ReadLine()

```
unsigned char File_ReadLine(unsigned char id, unsigned char bank, char* addr);
```

Reads a line of text (up to 254 bytes) from the open file handle id into a buffer at bank bank, address addr. Text lines can be terminated by character 13 (\r), character 10 (\n), a Windows-style combination of both (\r\n), or the AMSDOS EOF character 0x1A. To avoid overflow, the buffer should be at least 255 bytes long. The seek position will be moved forward to the start of the next line.

Return value: On success, returns 0. On failure, sets and returns _fileerr.

SymbOS name: File_LineInput (FILLIN).

File_ReadComp()

```
unsigned short File_ReadComp(unsigned char id, unsigned char bank,  
                             char* addr, unsigned short len);
```

Reads a compressed data block from the open file handle `id` into a buffer at bank `bank`, address `addr` and decompresses it in-place using `Bank-Decompress()`. `len` must contain the total resulting length of the *decompressed* data, in bytes. The seek position will be moved past the end of the block, so the next call to `File_ReadComp()` will read the next part of the file.

This function is only available in SymbOS 4.0 and higher. For details on the structure of a compressed data block, see the documentation for `Bank-Decompress()`.

Return value: On success, returns the number of bytes read. On failure, sets `_fileerr` and returns 0.

SymbOS name: `File_Compressed (FILCPR)`.

File_Write()

```
unsigned short File_Write(unsigned char id, unsigned char bank,
                          char* addr, unsigned short len)
```

Writes `len` bytes from the buffer at bank `bank`, address `addr` to open file handle `id`. The seek position will be moved forward by `len` bytes, so the next call to `File_Write()` will write to the next part of the file.

Return value: On success, returns the number of bytes written. On failure, sets `_fileerr` and returns 0.

SymbOS name: `File_Output (FILOUT)`.

File_Seek()

```
long File_Seek(unsigned char id, long offset, unsigned char ref);
```

Sets the seek position of the open file handle `id` to `offset` bytes relative to the reference point `ref`, which may be one of:

- `SEEK_SET`: relative to the start of the file
- `SEEK_CUR`: relative to the current seek position
- `SEEK_END`: relative to the end of the file

For `SEEK_CUR` and `SEEK_END`, `offset` can be positive or negative. Some examples:

```
File_Seek(f, 546, SEEK_SET);    // set seek position to byte 546
File_Seek(f, -16, SEEK_CUR);    // move seek position backwards 16 bytes
len = File_Seek(f, 0, SEEK_END); // get length by seeking to file end
```

(When using `File_Seek()` to determine the length of a file, as in the last example, note that the result may slightly overestimate the actual file length on AMSDOS filesystems—see the caveats at [the start of this chapter](#).)

Return value: On success, returns the new seek position. On failure, sets `_fileerr` and returns -1.

SymbOS name: `File_Pointer (FILPOI)`.

File_ErrMsg()

```
void File_ErrMsg(void* modalWin);
```

Displays a message box with the current error in `_fileerr`, if any. `modalWin` specifies the address of a window data record that should be declared modal, if any; this window will not be able to be focused until the message box is closed. If `modalWin = 0`, no window will be declared modal.

Directory access

In addition to `symsos.h`, these functions can be found in `symsos/file.h`.

Dir_Read()

```
int Dir_Read(char* path, unsigned char attrib, void* buf,
             unsigned short len, unsigned short skip);
```

Reads the contents of the directory indicated by the absolute path stored at address `path`. This path may contain Windows-style wildcards (* and ?, e.g., *.txt), in which case only matching filenames will be returned. `attrib` is a bitmask consisting of an OR'd list of attribute types that should *not* be included in the results:

- ATTRIB_READONLY: do not include read-only files
- ATTRIB_HIDDEN: do not include hidden files
- ATTRIB_SYSTEM: do not include system files
- ATTRIB_VOLUME: do not include volume information files (recommended)
- ATTRIB_DIR: do not include directories
- ATTRIB_ARCHIVE: do not include archived files

The first `skip` entries will be skipped—this is useful for reading a large number of filenames by performing multiple calls to `Dir_Read()`. Filenames will not be sorted.

The resulting directory information will be written to the buffer at address `buf`, with a maximum length of `len` bytes. This buffer is structured as an array of `DirEntry` structs:

```
typedef struct {
    long len;           // file length in bytes
    unsigned long time; // system timestamp - decode with Time2Obj()
    unsigned char attrib; // attribute bitmask
    char name[13];      // filename (zero-terminated)
} DirEntry;
```

An example of a complete call to `Dir_Read()`:

```
DirEntry dirbuf[64];
Dir_Read("A:\\FILES\\*.TXT", ATTRIB_VOLUME | ATTRIB_DIR,
         dirbuf, sizeof(dirbuf), 0);
printf("%s\n", dirbuf[0].name);
```

Return value: On success, returns the number of files found (which may be zero, if no matching files were found). On failure, sets `_fileerr` and returns -1.

SymbOS name: Internally, calls `Directory_Input (DIRINP)`, but does some extra formatting work on the output. For the raw version of this call, see `Dir_ReadRaw()`.

Dir_ReadRaw()

```
int Dir_ReadRaw(unsigned char bank, char* path, unsigned char attrib,
               unsigned char bufbank, void* addr, unsigned short len,
               unsigned short skip);
```

Equivalent to `Dir_Read()`, but executes the system call directly without converting the output into an array of `DirEntry` structs. For details on the raw output format, see the [SymbOS developer documentation](#).

Return value: On success, returns the number of files found (which may be zero, if no matching files were found). On failure, sets `_fileerr` and returns -1.

SymbOS name: `Directory_Input (DIRINP)`.

Dir_ReadExt()

```
int Dir_ReadExt(unsigned char bank, char* path, unsigned char attrib,
               unsigned char bufbank, void* addr, unsigned short len,
               unsigned short skip, unsigned char cols);
```

An extended version of `Dir_Read()` that automatically formats directory information into a set of list controls for the use of windowed file-manager-type applications. The behavior of this function is fairly complicated (and its uses are fairly esoteric), so see the [SymbOS developer documentation](#) for further details.

SymbOS name: `Directory_Input_Extended (DEVDIR)`.

Dir_New()

```
unsigned char Dir_New(unsigned char bank, char* path);
```

Creates a new directory from the absolute path stored in bank `bank`, address `path`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_New (DIRNEW)`.

Dir_Rename()

```
unsigned char Dir_Rename(unsigned char bank, char* path, char* newname);
```

Renames the file or directory indicated by the absolute path stored in bank `bank`, address `path` to the new name stored in bank `bank`, address `newname`. (The new name is just a name, e.g., `FILE.TXT`, and does not contain the full path.)

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_Rename (DIRREN)`.

Dir_Move()

```
unsigned char Dir_Move(unsigned char bank, char* pathSrc, char* pathDst);
```

Moves the file indicated by the absolute path stored in bank `bank`, address `pathSrc` to the new absolute path stored in bank `bank`, address `pathDst`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_Move (DIRMOV)`.

Dir_Delete()

```
unsigned char Dir_Delete(unsigned char bank, char* path);
```

Deletes the file indicated by the absolute path stored in bank `bank`, address `path`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_DeleteFile (DIRDEL)`.

Dir_DeleteDir()

```
unsigned char Dir_DeleteDir(unsigned char bank, char* path);
```

Deletes the directory indicated by the absolute path stored in bank `bank`, address `path`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_DeleteDirectory (DIRRMD)`.

Dir_GetAttrib()

```
signed char Dir_GetAttrib(unsigned char bank, char* path);
```

Retrieves the file attributes of the file at the absolute path stored in bank `bank`, address `path`. The attributes are an OR'd bitmask consisting of one or more of `ATTRIB_READONLY`, `ATTRIB_HIDDEN`, `ATTRIB_SYSTEM`, `ATTRIB_VOLUME`, `ATTRIB_DIR` and `ATTRIB_ARCHIVE`.

Return value: On success, returns a bitmask of the file's attributes. On failure, sets `_fileerr` and returns -1.

SymbOS name: `Directory_Property_Get (DIRPRR)`.

Dir_SetAttrib()

```
unsigned char Dir_SetAttrib(unsigned char bank, char* path, unsigned char attrib);
```

Sets the file attributes of the file at the absolute path stored in bank `bank`, address `path` to the bitmask `attrib`, which is an OR'd bitmask consisting of one or more of `ATTRIB_READONLY`, `ATTRIB_HIDDEN`, `ATTRIB_SYSTEM`, `ATTRIB_VOLUME`, `ATTRIB_DIR` and `ATTRIB_ARCHIVE`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_Property_Set` (DIRPRS).

Dir_GetTime()

```
unsigned long Dir_GetTime(unsigned char bank, char* path, unsigned char which);
```

Retrieves the system timestamp of the file at the absolute path stored in bank `bank`, address `path`. The option `which` can be `TIME_MODIFIED` or `TIME_CREATED`. Timestamps can be read with the utility function `Time2Obj()`.

Return value: On success, returns the timestamp. On failure, sets `_fileerr` and returns 0.

SymbOS name: `Directory_Property_Get` (DIRPRR).

Dir_SetTime()

```
unsigned char Dir_SetTime(unsigned char bank, char* path, unsigned char which,  
                          unsigned long timestamp);
```

Sets the system timestamp of the file at the absolute path stored in bank `bank`, address `path`. The option `which` can be `TIME_MODIFIED` or `TIME_CREATED`. Timestamps can be created with the utility function `Obj2Time()`.

Return value: On success, returns 0. On failure, sets and returns `_fileerr`.

SymbOS name: `Directory_Property_Set` (DIRPRS).

Dir_PathAdd()

```
char* Dir_PathAdd(char* path, char* addition, char* dest);
```

An SCC convenience function that constructs an absolute file path from a base path (at address `path`) and a relative path addition (at address `addition`), storing the result in address `dest`. This is mainly used to turn relative paths into absolute paths for the file manager functions. (This function is similar to the system function `Shell_PathAdd()`, but is available even in windowed applications that do not use `SymShell`.)

Any relative path elements in the addition (`..`, `\`, etc.) will be resolved. If `path = 0`, the absolute path will be relative to the absolute path in which the current application's `.exe` or `.com` file is located; this is useful for loading accompanying data files that should always be in the same directory as the `.exe`.

Examples:

```
char abspath[256];
```



```
Dir_PathAdd("C:\SYMBOS\APPS", "..\MUSIC\MP3\LALALA.MP3", abspath);  
// yields: C:\SYMBOS\MUSIC\MP3\LALALA.MP3
```

```
Dir_PathAdd("A:\GRAPHICS\NATURE", "\SYMBOS", abspath);  
// yields: A:\SYMBOS
```

```
Dir_PathAdd("C:\ARCHIVE", "*.ZIP", abspath);  
// yields: C:\ARCHIVE\*.ZIP
```

```
Dir_PathAdd("A:\ARCHIVE", "C:\SYMBOS", abspath);  
// yields: C:\SYMBOS
```

```
Dir_PathAdd(0, "FILE.DAT", abspath);  
// if the app is located at C:\APPS\DEMO.EXE, yields: C:\APPS\FILE.DAT
```

Return value: Returns dest.

Multitasking routines

In addition to `symbol.h`, these functions can be found in `symbol/proc.h`.

Processes

SymbOS makes a distinction between an *application ID*, which is associated with a single application, and a *process ID*, which is associated with a single process. An application may potentially open multiple processes, although most do not. *Pay close attention to which is required by a given function!*

App_Run()

```
unsigned short App_Run(char bank, char* path, char suppress);
```

Runs the app whose absolute path is at bank `bank`, address `path`. If `suppress = 1`, any errors that occur during loading will be suppressed; otherwise, they will appear in a message box. If the path points to a non-executable file of a known type (such as `.txt`), it will be opened with its associated application, exactly like double-clicking it in SymCommander. Paths starting with `%` will be understood as starting with the system path; e.g., if the system path were `C:\SYMBOS\`, the path `%CMD.EXE` would run `C:\SYMBOS\CMD.EXE`.

Return value: On success, returns the application ID as the low byte and the process ID of the application's main process as the high byte. These can be extracted with:

```
appid = result & 0xFF;
procid = result >> 8;
```

The process ID will always be greater than 3, so if the return value is greater than 3, the app was opened successfully.

On failure, returns an error code:

- `APPERR_NOFILE`: the file does not exist.
- `APPERR_UNKNOWN`: the file is not executable or of a known type.
- `APPERR_LOAD`: file system error (sets `_fileerr` with error code).
- `APPERR_NOMEM`: out of memory.

SymbOS name: `Program_Run_Command (MSC_SYS_PRGRUN)`.

App_End()

```
void App_End(char appID);
```

Forcibly closes the app with the application ID `appID`. SymbOS will close any processes, windows, and other system resources registered to the app, but since it is possible for apps to reserve some resources (particularly file handles and banked memory) without explicitly registering them to the app, memory leaks may occur. (Treat this command like force-killing an application in the Task Manager: safe if we control the app and know exactly what it is doing, but a last-resort otherwise.)

SymbOS name: `Program_End_Command (MSC_SYS_PRGEND)`.

App_Search()

```
unsigned short App_Search(char bank, char* idstring);
```

Searches for the running application with the first 12 bytes of the internal application name matching the string at bank `bank`, address `idstring`. (This is the application name listed in the Task Manager and which can be set with the `-N` command-line option to `cc`.) This function is mainly used for determining the process IDs of shared services such as network drivers so we can send messages to them.

Return value: If a matching app has been found, returns the application ID as the low byte and the process ID of the application's main process as the high byte. These can be extracted with:

```
appid = result & 0xFF;
procid = result >> 8;
```

If no matching app can be found, returns 0.

SymbOS name: `Program_SharedService_Command (MSC_SYS_PRGSRV)`.

App_Service()

```
unsigned short App_Service(char bank, char* idstring);
```

Similar to `App_Search()`, but used to connect with system shared services. The behavior is identical to `App_Search`, with two differences:

- 1) If the specified app is not currently running, SymbOS will attempt to start it from the system path. The `idstring` must have the exact format `%NNNNNNNN.EE` (%, eight ASCII characters for the filename, ., and two ASCII characters for the first part of the extension). The last character will be filled in automatically based on the current system type:
 - c for Amstrad CPC
 - p for Amstrad PCW
 - m for MSX

For example, if the system path were `C:\SYMBOS` and the system were an Amstrad CPC, `App_Service(_symlink, "%GAMEDRVR.EX")` would try to load `C:\SYMBOS\GAMEDRVR.EXC`.

- 2) After the app is successfully located or started, an internal counter will be incremented that indicates how many apps are using the service. This allows SymbOS to automatically decide whether the service is still being used and should remain open. (We can unregister with the service with `App_Release()`.)

Return value: If the application could be found or was successfully loaded, returns the application ID as the low byte and the process ID of the application's main process as the high byte. These can be extracted with:

```
appid = result & 0xFF;
procid = result >> 8;
```

The process ID will always be greater than 3, so if the return value is greater than 3, the app was found or opened successfully.

If the application could not be found or was not successfully loaded, returns an error code:

- APPERR_NOFILE: the file does not exist.
- APPERR_UNKNOWN: the file is not executable or of a known type.
- APPERR_LOAD: file system error (sets `_fileerr` with error code).
- APPERR_NOMEM: out of memory.

SymbOS name: Program_SharedService_Command (MSC_SYS_PRGSRV).

App_Release()

```
void App_Release(char appID);
```

Decrements the service counter incremented by `App_Service()`, so SymbOS knows when the shared service is no longer being used and can be closed. The parameter `appID` specifies the application ID of the service app to release.

SymbOS name: Program_SharedService_Command (MSC_SYS_PRGSRV).

Proc_Add()

```
signed char Proc_Add(unsigned char bank, void* header, unsigned char priority);
```

Launches a new process based on the information given in `bank`, address `header`. The process will be started with the priority `priority`, from 1 (highest) to 7 (lowest). The standard priority for application is 4.

Usually if we just want to run an executable file from disk, we should use `App_Run()`, not `Proc_Add()`. This function is for a lower-level operation, starting a new process running code we have already defined in memory. This may be code we have loaded from a file, or even part of our application's own main code; see `thread_start()` for a wrapper function that uses this to implement multithreading.

`header` must point to a data structure in the **transfer** segment with the struct type `ProcHeader`:

```
typedef struct {
    unsigned short ix; // initial value of IX register pair
    unsigned short iy; // initial value of IY register pair
    unsigned short hl; // initial value of HL register pair
    unsigned short de; // initial value of DE register pair
    unsigned short bc; // initial value of BC register pair
    unsigned short af; // initial value of AF register pair
    void* startAddr;    // address of routine to run
    unsigned char pid;  // process ID, set by kernel
} ProcHeader;
```

In addition, when the process is launched, its internal stack will begin at the address immediately before this header and grow downwards. So, we must define space for the stack immediately before the header in the **transfer** segment. `startAddr` can be an absolute address, or (as in the example below) the address of a `void` function in our own main code to run as a separate thread. Due to a quirk in SCC's linker, which currently treats initialized and uninitialized arrays differently, this buffer must

be given an initial value (such as {0}) to ensure that it is placed directly before the ProcHeader data structure in the **transfer** segment. (This may be improved in future releases.) For example:

```
char subprocID;

void proccode(void) {
    /* ... process code, do something here ... */
    Proc_Delete(subprocID); // end the subprocess (rather than returning)
}

_transfer char procstack[256] = {0};
_transfer ProcHeader prothead = {0, 0, 0, 0, 0, 0, // initial register values
                                proccode};      // address of routine to run

int main(int argc, char* argv[]) {
    subprocID = Proc_Add(_symbank, &prothead, 4);
    /* ... */
}
```

(See [Multithreading](#) for a simpler way to implement threads within our own code, as well as some important discussion about behaviors to avoid when doing this.)

Return value: On success, returns the process ID of the newly launched process. On failure, returns -1.

SymbOS name: Multitasking_Add_Process_Command (MSC_KRL_MTADDP).

Proc_Delete()

```
void Proc_Delete(unsigned char pid);
```

Forcibly stops execution of the process with the process ID `pid`. (This is most useful for processes we have launched ourselves with `Proc_Add()`; for stopping an entire application and freeing its resources, see [App_End\(\)](#).)

SymbOS name: Multitasking_Delete_Process_Command (MSC_KRL_MTDELP).

Proc_Sleep()

```
void Proc_Sleep(unsigned char pid);
```

Forcibly puts the process with the process ID `pid` into “sleep” mode. It will not continue execution until it receives a message or is woken up by another system function (such as `Proc_Wake()`).

SymbOS name: Multitasking_Sleep_Process_Command (MSC_KRL_MTSLEPP).

Proc_Wake()

```
void Proc_Wake(unsigned char pid);
```

Wakes up the process with the process ID `pid` from “sleep” mode.

SymbOS name: `Multitasking_Sleep_Process_Command (MSC_KRL_MTSLPP)`.

Proc_Priority()

```
void Proc_Priority(unsigned char pid, unsigned char priority);
```

Sets the scheduler priority of the process with the process ID `pid` to `priority`, which may be from 1 (highest) to 7 (lowest). The default priority is usually 4. A process is allowed to change its own priority.

SymbOS name: `Multitasking_Sleep_Process_Command (MSC_KRL_MTSLPP)`.

Timers

SCC implements three types of clock events that fire at regular intervals: **counters**, **timers**, and **wake timers**. These can be used in cases where we need something to occur at regular intervals of real time (such as movement in a game), rather than at regular intervals of CPU time.

A **counter** is the simplest method. A byte in memory can be registered as a counter, after which it will be incremented automatically from 1 to 50 times per second. We can then check the value of this byte regularly, and if it has changed, perform whatever action needs to be done.

A **timer** is more complicated. It is essentially a new process that is only given CPU time for brief, regular intervals every 1/50th or 1/60th of a second (depending on the screen vsync frequency of the platform). See `Timer_Add()` and `Proc_Add()` for details on how to define this process.

A **wake timer** is a simplified wrapper that uses a **timer** to perform a single task: at a regular interval, it sends a [message](#) to the specified process ID. The idea is that, instead of continuously running at 100% CPU checking a **counter**, the target process can sit idling on `Msg_Sleep()` and only “wake up” every so often when the wake timer fires.

Counter_Add()

```
unsigned char Counter_Add(unsigned char bank, char* addr, unsigned char pid,
                          unsigned char speed);
```

Registers a new **counter** byte at bank `bank`, address `addr`, incrementing every `speed`/50 seconds. (For example, to increment the counter twice per second, set `speed` = 25, because $25/50 = 0.5$ seconds) The process ID `pid` is the process to register this counter with (usually our own, `_sympid`).

Return value: On success, returns 0. On failure, returns 1.

SymbOS name: `Timer_Add_Counter_Command (MSC_KRL_TMADDT)`.

Counter_Delete()

```
void Counter_Delete(unsigned char bank, char* addr);
```

Unregisters the **counter** byte at bank bank, address addr so it no longer increments automatically.

SymbOS name: Timer_Delete_Counter_Command (MSC_KRL_TMDELT).

Counter_Clear()

```
void Counter_Clear(unsigned char pid);
```

Unregisters all **counter** bytes associated with the process ID *pid*.

SymbOS name: Timer_Delete_AllProcessCounters_Command (MSC_KRL_TMDELP).

Timer_Add()

```
signed char Timer_Add(unsigned char bank, void* header);
```

Behaves identically to Proc_Add(), but launches the new process as a **timer**. (See Proc_Add() for details on how a new process is implemented.) The timer code should ideally be short to ensure that it can fully complete in the allotted time, even under higher CPU load. Typically, we want to implement the timer code as a short loop that ends by calling Idle(); the timer process will then execute the loop contents, finish, go to sleep, and be woken up 1/50th of a second later for another pass through the loop:

```
void timer_loop(void) {
    while (1) {
        /* ... do something here ... */
        Idle();
    }
}
```

Return value: On success, returns the timer ID needed to stop the timer later with Timer_Delete(). On failure, returns -1.

SymbOS name: Multitasking_Add_Timer_Command (MSC_KRL_MTADDT).

Timer_Wake()

Currently only available in development builds of SCC.

```
signed char Timer_Wake(unsigned char pid, unsigned char msgid, unsigned short cycles);
```

A simplified wrapper around Timer_Add() that sets up a “wake timer”. Every cycles iterations (in 1/50ths of a second), the timer process will send a [message](#) with the first byte msg[0] = msgpid to the process ID pid. For example, to wake our main process up every 5 seconds with msg[0] = 200:

```
signed char wake_pid;
wake_pid = Timer_Wake(_sympid, 200, 5*50);
```

This function only supports running one wake timer at a time; for more complex behavior, see Timer_Add().

Return value: On success, returns the timer ID needed to stop the timer later with `Timer_Delete()`. On failure, returns -1.

Timer_Delete()

```
void Timer_Delete(unsigned char id);
```

Stops execution of a timer previously launched by `Timer_Add()` or `Timer_Wake()`. The parameter `id` is the timer ID returned by `Timer_Add()` or `Timer_Wake()`.

SymbOS name: `Multitasking_Delete_Timer_Command (MSC_KRL_MTDELT)`.

Multithreading

Yes, **SCC supports multithreading** (!), thanks to SymbOS's elegant system for spawning subprocesses. Internally, threads are implemented as subprocesses of the main application (see `Proc_Add()` and `Proc_Delete()`), but the convenience functions `thread_start()` and `thread_quit()` are provided to reduce the amount of setup required.

In addition to `symbolos.h`, these functions can be found in `symbolos/threads.h`.

thread_start()

```
signed char thread_start(void* routine, char* env, unsigned short envlen);
```

Spawns a new thread by launching `routine`, which should be a `void` function in our code that takes no parameters. Stack and other information for the new thread will be stored in the environment buffer `env`, which must be in the **transfer** segment. The length of this buffer is passed as `envlen` and should generally be at least 256 bytes (more if the thread will make heavy use of the stack, e.g., with deep recursion or large local buffers).

Example:

```
_transfer char env1[256];

void threadmain(void) {
    /* ...thread code goes here, which may call other functions... */
    /* ... */
    thread_quit(env1); // quit thread (see below)
}

int main(int argc, char* argv[]) {
    thread_start(threadmain, env1, sizeof(env1));
    /* ...main thread continues here... */
}
```

Return value: On success, returns the process ID of the new thread, which will also be stored as the last byte in `env`. On failure, returns -1.

thread_quit()

```
void thread_quit(char* env);
```

Quits the running thread associated with the environment buffer `env`. (This function will not return, so it should only be used inside the thread in question, to quit itself. To forcibly end a running thread from inside a different thread, use `Proc_Delete()`; however, this can leave resources hanging if the thread is interrupted in the middle of a system call, so it is usually better to have the thread quit itself with `thread_quit()`.)

Thread safety

Warning: When multithreading, we have to worry about all the problems that come with multithreading—race conditions, deadlocks, concurrent access, reentrancy, etc. The current implementation of libc is also not generally designed with thread-safety in mind, so while most small utility functions (`memcpy()`, `strcat()`, etc.) are thread-safe, others are not—in particular, much of `stdio.h`. Any function that relies on temporarily storing data in a static buffer (rather than local variables) is not thread-safe and may misbehave if two threads call it at the same time. When in doubt, check the library source code to verify that a function does not rely on any static/global variables, or write your own reentrant substitute (e.g., using only local variables, or with a semaphore system that sets a global variable when the shared resource is being used and, if it is already set, loops until it is unset by whatever other thread is using the resource). Any use of 32-bit data types (**long**, **float**, **double**) is also currently not thread-safe, as these internally rely on static “extended” registers, meaning that only one thread at a time can safely use 32-bit data types.

Standard SymbOS system calls that do not use 32-bit data types (`File_Open()`, etc.) should all be thread-safe, as these use a semaphore system to ensure that only one message is passed in `_symmsg` at the same time.

Thread-safe messaging

When using inter-process messaging functions (`Msg_Send()`, `Msg_Sleep()`, etc.) within a secondary thread, it is strongly recommended to use the utility function `_msgpid()` for the return process ID instead of `_sympid`. E.g.:

```
Msg_Sleep(_msgpid(), -1, _symmsg);
```

If we used `_sympid`, all messages would be addressed to the application’s main process. This queue can technically be read by any process, but using one queue for all messages makes it easy to end up in situations where one thread accidentally receives and discards a message intended for another thread, causing all sorts of strange bugs. It’s safer to keep each thread’s messages completely separate using `_msgpid()`.

(Note that the value of `_msgpid()` is *not* simply the current process ID; its behavior will depend on SymbOS version.)

Device routines

In addition to `symbol.h`, these functions can be found in `symbol/device.h`.

Contents

- [Screen status](#)
- [Mouse status](#)
- [Keyboard status](#)
- [Time functions](#)
- [System configuration](#)

Other device-related subsections:

- [Sound routines](#)
- [Printer routines](#)

Screen status

Screen_Mode()

```
unsigned char Screen_Mode(void);
```

Returns the current screen mode, which depends on the current platform:

Mode	Platform	Resolution	Colors
0	PCW	720x255	2
1	CPC/EP	320x200	4
2	CPC/EP	640x200	2
5	MSX	256x212	16
6	MSX	512x212	4
7	MSX	512x212	16
8	G9K	384x240	16
9	G9K	512x212	16
10	G9K	768x240	16
11	G9K	1024x212	16

SymbOS name: `Device_ScreenMode (SCRGET)`.

Screen_Mode_Set()

```
void Screen_Mode_Set(char mode, char force, char vwidth);
```

Sets the current screen mode. `mode` is the screen mode (see `Screen_Mode()`). If `force = 1`, the mode will be forced to change. For G9K modes only, `vwidth` specifies the virtual desktop width, one of:

- 0 = no virtual desktop

- 1 = 512 pixels wide
- 2 = 1000 pixels wide

Screen_Colors()

`unsigned char Screen_Colors(void);`

Returns the number of displayed colors in the current screen mode (2, 4, or 16).

SymbOS name: Device_ScreenMode (SCRGET).

Screen_Width()

`unsigned short Screen_Width(void);`

Returns the horizontal width of the screen, in pixels.

SymbOS name: Device_ScreenMode (SCRGET).

Screen_Height()

`unsigned short Screen_Height(void);`

Returns the vertical height of the screen, in pixels.

SymbOS name: Device_ScreenMode (SCRGET).

Screen_Redraw()

`void Screen_Redraw(void);`

Reinitializes the desktop background and redraws the entire screen.

SymbOS name: DesktopService_RedrawComplete (DSK_SRV_DSKPLT).

Color_Get()

`unsigned short Color_Get(char color);`

Retrieves the true palette color of the indexed color `color`.

Return value: A 12-bit color, with the components:

- `(value >> 8) = red component (0 to 15)`
- `(value >> 4) & 0x0F = green component (0 to 15)`
- `value & 0x0F = blue component (0 to 15)`

SymbOS name: DesktopService_ColourGet (DSK_SRV_COLGET).

Color_Set()

```
void Color_Set(char color, unsigned short value);
```

Sets the true palette color of the indexed color `color` to the 12-bit color `value`, which has the format:

- $(value \gg 8)$ = red component (0 to 15)
- $(value \gg 4) \& 0x0F$ = green component (0 to 15)
- $value \& 0x0F$ = blue component (0 to 15)

SymbOS name: DesktopService_ColourSet (DSK_SRV_COLSET).

Text_Width()

```
unsigned short Text_Width(unsigned char bank, char* addr, int maxlen);
```

Returns the width (in pixels) of the text string at bank `bank`, address `address` if it were plotted with the current system font. `maxlen` contains the maximum number of characters to measure; if the text is terminated by character 0 or 13 (`\r`), use -1.

SymbOS name: Screen_TextLength (TXTLEN).

Text_Height()

```
unsigned short Text_Height(unsigned char bank, char* addr, int maxlen);
```

Returns the height (in pixels) of the text string at bank `bank`, address `address` if it were plotted with the current system font. `maxlen` contains the maximum number of characters to measure; if the text is terminated by character 0 or 13 (`\r`), use -1. (This function effectively just returns the pixel height of the system font, so we can just run it once with a dummy string like "A" and cache the result.)

SymbOS name: Screen_TextLength (TXTLEN).

Mouse status

Mouse_X()

```
unsigned short Mouse_X(void);
```

Returns the horizontal position of the mouse pointer, in pixels.

SymbOS name: Device_MousePosition (MOSGET).

Mouse_Y()

```
unsigned short Mouse_Y(void);
```

Returns the vertical position of the mouse pointer, in pixels.

SymbOS name: Device_MousePosition (MOSGET).

Mouse_Buttons()

```
unsigned char Mouse_Buttons(void);
```

Returns the current status of the mouse buttons as a bitmask. We can perform a binary AND of the return value with `BUTTON_LEFT`, `BUTTON_RIGHT`, and `BUTTON_MIDDLE` to determine whether the respective button is currently pressed:

```
lbut = Mouse_Buttons() & BUTTON_LEFT;
```

SymbOS name: Device_MouseKeyStatus (MOSKEY).

Mouse_Dragging()

Currently only available in development builds of SCC.

```
unsigned char Mouse_Dragging(unsigned char delay);
```

A utility function for determining whether a mouse click is the start of a drag-and-drop operation (or just a normal click). After detecting a click, call `Mouse_Dragging()` with the `delay` parameter indicating a short length of time to wait (in 1/50ths of a second, e.g., 25 = half a second). If the user releases the mouse within this time limit, this function immediately returns 0, indicating a single click. If time expires with the button still down, or the user moves the mouse without releasing the button, it immediately returns a nonzero bitmask (equivalent to `Mouse_Buttons()`) indicating which buttons are being held down for a drag-and-drop operation.

Return value: 0 for a click, nonzero bitmask for a drag.

Keyboard status

Key_Down()

```
unsigned char Key_Down(unsigned char scancode);
```

Returns 1 if the key specified by `scancode` is currently down, otherwise 0. **Note that keys are tested by *scancode*, not by their ASCII value!** A set of [scancode constants](#) are provided for convenience.

SymbOS name: Device_KeyTest (KEYTST).

Key_Status()

```
unsigned short Key_Status(void);
```

Returns the status of the modifier keys as a bitmask. We can perform a binary AND of the return value with `SHIFT_DOWN`, `CTRL_DOWN`, `ALT_DOWN`, and `CAPSLOCK_DOWN` to determine whether the respective modifier key is currently applied:

```
caps = Key_Status() & CAPSLOCK_DOWN;
```

SymbOS name: Device_KeyStatus (KEYSTA).

Key_Put()

```
void Key_Put(unsigned char keychar);
```

Pushes the ASCII code `keychar` into the keyboard buffer as if it had been pressed on the keyboard.

SymbOS name: `Device_KeyPut (KEYPUT)`.

Key_Multi()

```
unsigned char Key_Multi(unsigned char scancode1, unsigned char scancode2,
                        unsigned char scancode3, unsigned char scancode4,
                        unsigned char scancode5, unsigned char scancode6);
```

Like `Key_Test()`, but tests up to six keys simultaneously. This may save time when testing large numbers of keys for (e.g.) a game. The return value is a bitmask:

- Bit 0: set if key `scancode1` is pressed
- Bit 1: set if key `scancode2` is pressed
- Bit 2: set if key `scancode3` is pressed
- Bit 3: set if key `scancode4` is pressed
- Bit 4: set if key `scancode5` is pressed
- Bit 5: set if key `scancode6` is pressed

Note that keys are tested by *scancode*, not by their ASCII value! A set of [scancode constants](#) are provided for convenience.

SymbOS name: `Device_KeyMulti (KEYMUL)`.

Time functions

These functions may only behave as expected on systems that have a realtime clock (RTC).

In addition to `symbolos.h`, these functions can be found in `symbolos/time.h`.

Time_Get()

```
void Time_Get(SymTime* addr);
```

Loads the current time into the `SymTime` struct at the address `addr`, which has the format:

```
typedef struct {
    unsigned char second; // 0 to 59
    unsigned char minute; // 0 to 59
    unsigned char hour;    // 0 to 23
    unsigned char day;     // 1 to 31
    unsigned char month;   // 1 to 12
    unsigned short year;   // 1900 to 2100
    signed char timezone;  // -12 to +13 (UTC)
} SymTime;
```

SymbOS name: Device_TimeGet (TIMGET).

Time_Set()

```
void Time_Set(SymTime* addr);
```

Sets the current time to the values in the SymTime struct at the address addr. (See Time_Get() for the format.)

SymbOS name: Device_TimeSet (TIMSET).

Time2Obj()

```
void Time2Obj(unsigned long timestamp, SymTime* obj);
```

An SCC utility function that decodes the system timestamp timestamp (obtained from, e.g., Dir_GetTime()) into the SymTime struct at the address addr. (See Time_Get() for the format.)

Obj2Time()

```
unsigned long Obj2Time(SymTime* obj);
```

An SCC utility function that converts the SymTime struct at the address addr into a system timestamp. (See Time_Get() for the format.)

Return value: A 32-bit system timestamp.

System configuration

Sys_Path()

Currently only available in development builds of SCC.

```
char* Sys_Path(void);
```

Returns the system path (e.g., C:\SYMBOS\) as a string, which will always end in a backslash (\).

SymbOS name: System_Information (SYSINF).

Sys_Type()

Currently only available in development builds of SCC.

```
unsigned short Sys_Type(void);
```

Returns the type of machine on which SymbOS is running, one of: TYPE_CPC, TYPE_ENTERPRISE, TYPE_MSX, TYPE_PCW, TYPE_NC, TYPE_SVM, TYPE_CPC464, TYPE_CPC664, TYPE_CPC6128, TYPE_CPC464PLUS, TYPE_CPC6128PLUS, TYPE_MSX1, TYPE_MSX2, TYPE_MSX2PLUS, TYPE_MSXTURBOR, TYPE_PCW8, TYPE_PCW9, TYPE_PCW16, TYPE_NC100, TYPE_NC150, TYPE_NC200, TYPE_SVM (SymbOSVM), or TYPE_OTHER (anything undefined, included for forward-compatibility with potential future ports).

The general class of machine can be obtained by AND'ing this value with one of TYPE_CPC, TYPE_MSX, TYPE_ENTERPRISE, TYPE_PCW, TYPE_NC, or TYPE_SVM.

SymbOS name: System_Information (SYSINF).

Sys_GetDrives()

Currently only available in development builds of SCC.

```
void Sys_GetDrives(void* dest);
```

Loads available drives into dest, which must be an 8-member array of type Device_Info located in the **transfer** segment. Device_Info has the format:

```
typedef struct {
    unsigned char letter; // ASCII drive letter or 0 = undefined
    unsigned short config; // drive setup (see below)
    unsigned char unused;
    char name[12]; // device name, zero-terminated
} Device_Info;
```

The member config is a bitmask defining the system drive setup (partition number, etc.), which depends on the drive type; most of this is not relevant to user applications, so see the [SymbOS developer documentation](#) for details. Example:

```
_transfer Device_Info drives[8];
Sys_GetDrives(&drives);
```

SymbOS name: System_Information (SYSINF).

Sys_DriveInfo()

Currently only available in development builds of SCC.

```
void Sys_DriveInfo(char letter, Drive_Info* obj);
```

Loads information about the drive with ASCII drive letter letter into obj, a struct of type Drive_Info passed by reference:

```
typedef struct {
    unsigned char status; // status (see below)
    unsigned char type; // medium type (see below)
    unsigned char removeable; // 0 or 1
    unsigned char fs; // filesystem (see below)
    unsigned char sectors; // sectors per cluster
    unsigned long clusters; // total clusters
} Drive_Info;
```

status may be one of DRIVE_NONE, DRIVE_READY, DRIVE_NOTREADY, or DRIVE_CORRUPT. fs may be one of FS_AMSDOS_DATA, FS_AMSDOS_SYS, FS_PCW_180K, FS_FAT12, FS_FAT16, or FS_FAT32. type may be one of:

- TYPE_FLOPPY_SS - single-sided floppy (Amstrad, PCW)
- TYPE_FLOPPY_DS - double-sided floppy (FAT12)
- TYPE_RAMDRIVE - RAM drive
- TYPE_HARDDRIVE - IDE/SCSI/SD/MMC mass storage drive

SymbOS name: Directory_DriveInformation (DIRINF).

Sys_DriveFree()

Currently only available in development builds of SCC.

```
unsigned long Sys_DriveFree(char letter);
```

Returns the total number of free 512-byte sectors on the drive with the ASCII drive letter `letter`. (Note that this calculation may take some time on a large FAT16 device.)

SymbOS name: Directory_DriveInformation (DIRINF).

Sys_GetConfig()

Currently only available in development builds of SCC.

```
void Sys_GetConfig(char* dest, unsigned short offset, unsigned char len);
```

A low-level function that copies `len` bytes starting at `offset` within the SymbOS configuration information into the buffer at address `dest`, which must be in the **transfer** segment. `offset = 0` is byte 163 in SYMBOS.INI (i.e., the system path). For information on offsets and what information can be retrieved this way, see the SymbOS Developer Documentation.

SymbOS name: System_Information (SYSINF).

Sound

Sound capabilities are only available if an appropriate sound daemon is running (in SymbOS 4.0 or later).

SymbOS supports two incompatible sound chips, PSG and OPL4. PSG refers to the standard 3-voice “chiptune” Programmable Sound Generator (e.g., AY-3-8912) integrated into most classic platforms SymbOS can run on. The OPL4 is a more powerful “wavetable” chip that supports recorded audio playback and is usually only available in expansion cards.

The SymbOS sound daemon makes a distinction between “music” and “sound effects.” In general, only one music track can be playing at a time, but multiple shorter sound effects may be triggered and mixed together on top of it as it plays. Music and effects are loaded as part of “collections,” which contain multiple subsongs or effects that can be referred to by numerical ID.

Sound errors are recorded in the global variable `_sounderr`, documented [here](#). An example sound application (`snddemo.c`) is included in SCC’s `sample` folder.

In addition to `symbol.h`, these functions can be found in `symbol/sound.h`.

Contents

- [Creating/getting sounds](#)
- [Sound functions](#)

Creating/getting sounds

For PSG, music/effect collections consist of packaged and compressed Arkos Tracker II .AKG/.AKS files (for music) or .AKX files (for sound effects). For OPL4, music/effect collections consist of packaged and compressed Amiga MOD files (for music) or PCM WAV files (for sound effects). Tools for packaging raw files into collections are described [here](#).

An easy way to add sound effects without creating your own collection is to use the default system effects collection. This is always loaded and can be accessed using the resource handle 0. The effects in this collection have standard names and are intended for use by programs: `FX_CLICK1`, `FX_CLICK2`, `FX_BEEP1`, `FX_BEEP2`, `FX_RING1`, `FX_RING2`, `FX_ALERT1`, `FX_ALERT2`, `FX_SLIDE1`, `FX_SLIDE2`, `FX_RAISE`, `FX_LOWER`, `FX_POPUP`, `FX_SHRINK`, `FX_TIC1`, `FX_TIC2`, `FX_SHOOT`, `FX_EXPLODE`, `FX_STEP`, `FX_LOSE`, `FX_WIN`, `FX_CAR`, and `FX_PLANE`. Note that the user can theoretically change what collection is used for the system sounds, so there is no guarantee of exactly what a given effect will sound like on every computer. However, the system effects should be stable enough that (e.g.) `FX_BEEP1` can always be expected to be some kind of beep.

Sound functions

Sound_Init()

```
signed char Sound_Init(void);
```

Initializes the sound interface, if present. This should be called before using any other sound functions.

After initialization, the process ID of the sound daemon will be stored in `_soundpid`, and the available audio hardware will be stored in `_soundhw` as an OR'd bitmask:

- `_soundhw & SOUND_PSG` will be nonzero if the user has a PSG-compatible sound chip.
- `_soundhw & SOUND_OPL4` will be nonzero if the user has an OPL4-compatible sound chip.

The preferred audio hardware will be stored in `_soundpref` (one of `SOUND_NONE`, `SOUND_PSG`, or `SOUND_OPL4`).

Return value: On success, sets the variables above and returns 0. On failure, sets `_sounderr` and returns -1.

Music_Load()

```
signed char Music_Load(unsigned char fid, unsigned char hw);
```

Loads a music collection from the open file handle `fid` into memory, for device `hw` (one of `SOUND_PSG` or `SOUND_OPL4`). Only one music collection can be loaded at a time, so if the program has already loaded a collection, it will first be freed with `Music_Free()`.

Note that this function takes a file *handle*, not a file *path*. That is, the file must first be opened with `File_Open()`, then read with `Music_Load()`, and then closed with `File_Close()`. (The idea is that multiple sound assets might be stored in the same file, so it's up to the program to decide how to manage this file; `Music_Load()` will just start reading from the file's current seek position.)

Return value: On success, returns 0. On failure, sets `_sounderr` and returns -1.

Music_Load_Mem()

```
signed char Music_Load_Mem(unsigned char bank, char* addr, unsigned short len);
```

Loads a music collection from the memory location at bank `bank`, address `addr`, where the music data is `len` bytes long. (Only PSG music can be loaded in this way.) Only one music collection can be loaded at a time, so if the program has already loaded a collection, it will first be freed with `Music_Free()`.

Return value: On success, returns 0. On failure, sets `_sounderr` and returns -1.

Music_Free()

```
void Music_Free(void);
```

Unloads the currently loaded music collection, if any. (This is also done automatically on program exit.)

Music_Start()

```
void Music_Start(unsigned char track);
```

Starts playing (from the beginning) track number `track` from the currently loaded music collection.

Music_Stop()

```
void Music_Stop(void);
```

Pauses and mutes the currently playing track from the currently loaded music collection.

Music_Continue()

```
void Music_Continue(void);
```

Resumes playing the last-played music track from the currently loaded music collection.

Music_Volume()

```
void Music_Volume(unsigned char vol);
```

Sets the playback volume for music, from 0 (silent) to 255 (loud). Note that volume is reset automatically to 255 (loud) after loading a music collection.

Effect_Load()

```
signed char Effect_Load(unsigned char fid, unsigned char hw);
```

Loads an effect collection from the open file handle *fid* into memory, for device *hw* (one of SOUND_PSG or SOUND_OPL4).

Note that this function takes a file *handle*, not a file *path*. That is, the file must first be opened with `File_Open()`, then read with `Effect_Load()`, and then closed with `File_Close()`. (The idea is that multiple sound assets might be stored in the same file, so it's up to the program to decide how to manage this file; `Effect_Load()` will just start reading from the file's current seek position.)

Up to 16 effect collections may be loaded simultaneously, distinguished by their resource handles.

Return value: On success, returns the resource handle of the effect collection (1-16). On failure, sets `_sounderr` and returns -1.

Effect_Load_Mem()

```
signed char Effect_Load_Mem(unsigned char bank, char* addr, unsigned short len);
```

Loads an effect collection from the memory location at bank *bank*, address *addr*, where the effect data is *len* bytes long. (Only PSG effects can be loaded in this way.)

Up to 16 effect collections may be loaded simultaneously, distinguished by their resource handles.

Return value: On success, returns the resource handle of the effect collection (1-16). On failure, sets `_sounderr` and returns -1.

Effect_Free()

```
void Effect_Free(unsigned char handle);
```

Unloads the effect collection with the resource handle `handle`. (This is also done automatically on program exit.)

Effect_Play()

```
void Effect_Play(unsigned char handle, unsigned char id, unsigned char volume,  
                unsigned char priority, unsigned char pan, int pitch) ;
```

Starts playing effect number `id` from the loaded effect collection with the resource handle `handle`. `volume` goes from 0 (silent) to 255 (loud).

`priority` specifies how the effect should be played (use 0 for the defaults specified below). For PSG effect collections, the available priorities are:

- `FX_ANY` - force playing on any channel (default)
- `FX_FORCE` - force playing on the specified channel
- `FX_OPTIONAL` - play only if a channel is free
- `FX_OPTCH` - play only if the specified channel is free
- `FX_ONLY` - play only if no other effect is active

For OPL4 effect collections, the available priorities are:

- `FX_PLAY` - play (default)
- `FX_SINGLE` - first stop any other instance of the same effect
- `FX_SOLO` - first stop any other effects from the same collection

`pan` can be one of `PAN_LEFT`, `PAN_MIDDLE`, or `PAN_RIGHT`. For OPL4 effect collections, `pan` can also be specified as a number from 0 (left) to 255 (right).

`pitch` is only meaningful for OPL4 effect collections and can be used to change the effect pitch. 0 = standard pitch.

Effect_Stop()

```
void Effect_Stop(unsigned char handle, unsigned char id);
```

Stops playing all instances of effect number `id` from the loaded effect collection with the resource handle `handle`.

Printer routines

Cross-platform printer support is provided (in SymbOS 4.0 and up) by the system print daemon. There are two standard ways of sending a print job to the printer: directly via the print daemon, or via the PrintIt helper program. On CPC and MSX machines only, it is also possible to directly send data to the printer port, using functions found in `symbos/device.h`.

Contents

- [Printing via the daemon](#)
- [Printing via PrintIt](#)
- [Direct printer functions](#)

Printing via the daemon

When running, the printer daemon (`PRINTD.EXE`) periodically scans the subfolder `PRINTD` within the system directory for files with the extension `.JOB`. Once a `.JOB` file is found, it is sent to the printer, with the daemon handling all the necessary hardware management. Printing a file therefore just requires creating a file in the `PRINTD` subfolder:

```
char fid;
char filepath[256];

// create output filename
strcpy(filepath, Sys_Path());
strcat(filepath, "PRINTD\\PR742A.TMP");

// open file
fid = File_New(_symbank, filepath);
if (fid <= 7) {
    // ... opened correctly, write to file ...
    // ... write code goes here ...
    // ... then close and rename to .JOB
    File_Close(fid);
    Dir_Rename(_symbank, filepath, "PR742A.JOB");
}
```

As this example demonstrates, it is good practice to initially write the file with a different extension and only rename it to `.JOB` when finished; otherwise, there is a possibility that the printer daemon may start trying to print the file before it is finished being written. Files sent to the printer in this way should be in plain ASCII, without extended characters (ASCII value >127), and we are responsible for word-wrapping any lines over 80 characters. Standardized formatting codes for bold, italic, etc. can be included, which the printer daemon will automatically convert to the correct formatting for the user's printer; see `README.TXT` in the printer daemon folder for the most up-to-date information on supported formatting codes.

Printing via PrintIt

Directly sending a job to the print daemon is flexible, but requires us to perform all necessary formatting, including word-wrapping and pagination (splitting the file into pages with the correct spacing for the user's printer). This can be very fiddly, so SymbOS provides a standardized formatting program, PrintIt (PRINTIT.EXE), which can always be found in the printer daemon folder. To use PrintIt, we simply pass the full path of an ASCII text file to PRINTIT.EXE as a command-line option. PrintIt will bring up a settings window with various formatting options; margins, word-wrap, line numbering, etc. When the user is satisfied with their settings, they just click "Print" and PrintIt automatically formats the file and sends it to the printer daemon.

For normal ASCII printing, using PrintIt is highly recommended because it lets the user set and remember the correct page-formatting settings for their printer across multiple apps. (We can still include formatting codes, which will be passed through unmodified to the printer daemon; see above.)

```
char fid;
char printcmd[256];
char filepath[256];

// create temporary file PRINT.TMP in the same folder as our app
Dir_PathAdd(0, "PRINT.TMP", filepath);
fid = File_New(_symbank, filepath);
if (fid <= 7) {
    // ... opened correctly, write to file ...
    // ... write code goes here ...
    File_Close(fid);

    // create PrintIt command (e.g., %PRINTD\PRINTIT.EXE C:\APPPDIR\PRINT.TMP);
    // note that App_Run() understands "%" as the system folder.
    strcpy(printcmd, "%PRINTD\PRINTIT.EXE ");
    strcat(printcmd, filepath);

    // run it
    App_Run(_symbank, printcmd, 0);
}
```

Direct printer functions

In addition to `symbos.h`, these functions can be found in `symbos/device.h`.

Print_Busy()

Currently only available in development builds of SCC.

```
signed char Print_Busy(void);
```

On CPC and MSX machines only, checks the status of the connected parallel printer. (This function interfaces directly with the printer hardware rather than going through the daemon, so it should only be used when direct hardware control is required. For normal printing, use `PrintIt` or the printer daemon.)

Return value: 0 = printer is ready to receive data; -1 = printer is busy; 1 = platform unsupported.

Print_Char()

Currently only available in development builds of SCC.

```
signed char Print_Char(unsigned char ch);
```

On CPC and MSX machines only, sends the single character `ch` to the connected parallel printer. (This function interfaces directly with the printer hardware rather than going through the daemon, so it should only be used when direct hardware control is required. For normal printing, use `PrintIt` or the printer daemon.)

Note that, on the Amstrad CPC, the high bit of `ch` will be ignored because the CPC printer port only connects 7 of the 8 data lines. Normal ASCII (<128) should work, but extended ASCII (>127) or 8-bit binary data may require workarounds.

Return value: 0 = printed successfully; 1 = platform unsupported; 2 = timeout.

Print_String()

Currently only available in development builds of SCC.

```
signed char Print_String(char* str);
```

On CPC and MSX machines only, sends the ASCII string `str` to the connected parallel printer. (This function interfaces directly with the printer hardware rather than going through the daemon, so it should only be used when direct hardware control is required. For normal printing, use `PrintIt` or the printer daemon.)

Return value: 0 = printed successfully; 1 = platform unsupported; 2 = timeout.

Reference tables

Contents

- [Keyboard scancodes](#)
- [Keyboard ASCII codes](#)
- [Colors](#)
- [Error codes](#)

Keyboard scancodes

In addition to `symbols.h`, these definitions can be found in `symbols/keys.h`.

Code	Code	Code	Code	Code
SCAN_0	SCAN_G	SCAN_W	SCAN_UP	SCAN_FIRE_1
SCAN_1	SCAN_H	SCAN_X	SCAN_DOWN	SCAN_FIRE_2
SCAN_2	SCAN_I	SCAN_Y	SCAN_LEFT	SCAN_JOY_DOWN
SCAN_3	SCAN_J	SCAN_Z	SCAN_RIGHT	SCAN_JOY_LEFT
SCAN_4	SCAN_K	SCAN_F0	SCAN_ALT	SCAN_JOY_RIGHT
SCAN_5	SCAN_L	SCAN_F1	SCAN_AT	SCAN_JOY_UP
SCAN_6	SCAN_M	SCAN_F2	SCAN_BSLASH	SCAN_LBRACKET
SCAN_7	SCAN_N	SCAN_F3	SCAN_CAPSLOCK	SCAN_MINUS
SCAN_8	SCAN_O	SCAN_F4	SCAN_CARET	SCAN_PERIOD
SCAN_9	SCAN_P	SCAN_F5	SCAN_CLR	SCAN_RBRACKET
SCAN_A	SCAN_Q	SCAN_F6	SCAN_COLON	SCAN_RETURN
SCAN_B	SCAN_R	SCAN_F7	SCAN_COMMA	SCAN_SEMICOLON
SCAN_C	SCAN_S	SCAN_F8	SCAN_CTRL	SCAN_SHIFT
SCAN_D	SCAN_T	SCAN_F9	SCAN_DEL	SCAN_SLASH
SCAN_E	SCAN_U	SCAN_FDOT	SCAN_ENTER	SCAN_SPACE
SCAN_F	SCAN_V	SCAN_ESC	SCAN_TAB	

Keyboard ASCII codes

In addition to `symbols.h`, these definitions can be found in `symbols/keys.h`.

Code	Code	Code	Code
KEY_CTRL_A	KEY_ALT_A	KEY_ALT_0	KEY_TAB
KEY_CTRL_B	KEY_ALT_B	KEY_ALT_1	KEY_BACK
KEY_CTRL_C	KEY_ALT_C	KEY_ALT_2	KEY_DEL
KEY_CTRL_D	KEY_ALT_D	KEY_ALT_3	KEY_ENTER
KEY_CTRL_E	KEY_ALT_E	KEY_ALT_4	KEY_ESC
KEY_CTRL_F	KEY_ALT_F	KEY_ALT_5	KEY_INS
KEY_CTRL_G	KEY_ALT_G	KEY_ALT_6	KEY_PRINT

Code	Code	Code	Code
KEY_CTRL_H	KEY_ALT_H	KEY_ALT_7	KEY_HOME
KEY_CTRL_I	KEY_ALT_I	KEY_ALT_8	KEY_END
KEY_CTRL_J	KEY_ALT_J	KEY_ALT_9	KEY_CTRL_1
KEY_CTRL_K	KEY_ALT_K	KEY_ALT_AT	KEY_CTRL_2
KEY_CTRL_L	KEY_ALT_L	KEY_F0	KEY_CTRL_3
KEY_CTRL_M	KEY_ALT_M	KEY_F1	KEY_CTRL_4
KEY_CTRL_N	KEY_ALT_N	KEY_F2	KEY_CTRL_5
KEY_CTRL_O	KEY_ALT_O	KEY_F3	KEY_CTRL_6
KEY_CTRL_P	KEY_ALT_P	KEY_F4	KEY_CTRL_7
KEY_CTRL_Q	KEY_ALT_Q	KEY_F5	KEY_CTRL_8
KEY_CTRL_R	KEY_ALT_R	KEY_F6	
KEY_CTRL_S	KEY_ALT_S	KEY_F7	
KEY_CTRL_T	KEY_ALT_T	KEY_F8	
KEY_CTRL_U	KEY_ALT_U	KEY_F9	
KEY_CTRL_V	KEY_ALT_V	KEY_FDOT	
KEY_CTRL_W	KEY_ALT_W	KEY_UP	
KEY_CTRL_X	KEY_ALT_X	KEY_DOWN	
KEY_CTRL_Y	KEY_ALT_Y	KEY_LEFT	
KEY_CTRL_Z	KEY_ALT_Z	KEY_RIGHT	

Colors

In addition to `symbos.h`, these definitions can be found in `symbos/windows.h`. International English synonyms (`COLOUR`, `GREY`) are also available.

Value	4-color	16-color
0	COLOR_YELLOW	COLOR_YELLOW
1	COLOR_BLACK	COLOR_BLACK
2	COLOR_ORANGE	COLOR_ORANGE
3	COLOR_RED	COLOR_RED
4		COLOR_CYAN
5		COLOR_DBLUE
6		COLOR_LBLUE
7		COLOR_BLUE
8		COLOR_WHITE
9		COLOR_GREEN
10		COLOR_LGREEN
11		COLOR_MAGENTA
12		COLOR_LYELLOW
13		COLOR_GRAY
14		COLOR_PINK

Value	4-color	16-color
15		COLOR_LRED

Error codes

The following errors are primarily issued by file commands (stored in `_fileerr`):

- `ERR_DEVINIT`: Device not initialized
- `ERR_DAMAGED`: Media is damaged
- `ERR_NOPART`: Partition does not exist
- `ERR_UNPART`: Unsupported media or partition
- `ERR_READ`: Error during sector read/write
- `ERR_SEEK`: Error during seek
- `ERR_ABORT`: Abort during volume access
- `ERR_NOVOL`: Unknown volume
- `ERR_TOOMANY`: No free filehandle
- `ERR_NODEV`: Device does not exist
- `ERR_NOPATH`: Path does not exist
- `ERR_NOFILE`: File does not exist
- `ERR_FORBIDDEN`: Access is forbidden
- `ERR_BADNAME`: Invalid path or filename
- `ERR_NOHANDLE`: Filehandle does not exist
- `ERR_DEVSLOT`: Device slot already occupied
- `ERR_FILEORG`: Error in file organization
- `ERR_BADDEST`: Invalid destination name
- `ERR_EXISTS`: File/path already exists
- `ERR_BADCODE`: Invalid subcommand code
- `ERR_BADATTRIB`: Invalid attribute
- `ERR_DIRFULL`: Directory is full
- `ERR_DISKFULL`: Media is full
- `ERR_PROTECT`: Media is write-protected
- `ERR_NOTREADY`: Device not ready
- `ERR_NOTEMPTY`: Directory is not empty
- `ERR_BADDEV`: Invalid destination device
- `ERR_FILESYS`: Not supported by file system
- `ERR_UNDEV`: Unsupported device
- `ERR_RDONLY`: File is read-only
- `ERR_NOCHANNEL`: Device channel unavailable
- `ERR_NOTDIR`: Destination is not a directory
- `ERR_NOTFILE`: Destination is not a file
- `ERR_UNDEFINED`: Undefined error

The following errors are primarily issued by SymShell commands (stored in `_shellerr`):

- ERR_NOPROC: Process ID is not registered with SymShell
- ERR_DEVFULL: Destination device is full
- ERR_RINGFULL: Internal ring buffer is full
- ERR_MOREPROC: Too many processes registered with SymShell
- ERR_NOSHELL: No SymShell session available (`_shellpid = 0`)

The following errors are primarily issued by the sound interface (stored in `_sounderr`):

- ERR_NOSOUND: No sound daemon
- ERR_TOOMANY: Too many effects collections already loaded

For network error codes, see [Network Library](#).

Graphics Library

While SymbOS provides features (primarily the `C_IMAGE` and `C_IMAGE_EXT` controls) for directly plotting images, the graphics format is internally complicated and depends on the exact platform and screen mode in which the application is running. The SCC graphics library simplifies this process by implementing a set of standard functions for loading and plotting images and sprites, as well as directly manipulating pixels on a canvas.

Note: All graphics functions are thread-safe, except that only one canvas may be [active](#) at a time. However, multiple threads can draw to the same canvas at the same time.

Contents

- [Using the library](#)
- [Using canvases](#)
 - [Creating canvases](#)
 - [Initializing canvases](#)
 - [Refreshing the display](#)
- [Drawing functions](#)
 - [Function reference](#)
- [Sprite functions](#)
 - [Converting images](#)
 - [Image sets and masks](#)
 - [Function reference](#)
- [Raw images](#)
 - [Raw image controls](#)
 - [Function reference](#)
- [Advanced topics](#)
 - [Moving sprites](#)
 - [Memory problems](#)
- [Reference](#)
 - [Color palette](#)

Using the library

To use the library, include the `graphics.h` header:

```
#include <graphics.h>
```

Additionally, use the `-lgfx` option when compiling to specify that the executable should be linked with the graphics library `libgfx.a`:

```
cc source.c -lgfx
```

Using canvases

The SCC graphics library provides a set of functions for working with “canvases”, bitmapped image buffers that can be displayed in SymbOS windows using the `C_IMAGE_EXT` control. We can plot sprites, draw lines, and directly edit the displayed image in other ways.

Creating canvases

To create a canvas, first define a `char` buffer with the size $(\text{width} * \text{height} / 2) + 24$:

```
_data char canvas[(128*64/2) + 24];
```

Canvas size is limited to 504x255 pixels, and **the width of a canvas in pixels must be a multiple of 8**. To work correctly on all platforms, a single canvas must also stay within a single 16KB memory segment; the easiest way to guarantee this is to place it in either the **data** or **transfer** segment. (This 16KB limit unfortunately means that the maximum *practical* size of a single canvas is much smaller than 504x255: e.g., 180x180, 240x136, 504x64, etc.)

To show this canvas in a window, we can use the `C_IMAGE_EXT` control. Set the `param` argument to point to the address of the canvas, and the `w` and `h` arguments to match the desired pixel width and height of the canvas:

```
_transfer Ctrl c_canvas =
{1,                // control ID
 C_IMAGE_EXT,      // control type
 -1,               // canvas bank (-1 = same as this record)
 (unsigned short)canvas, // address of canvas
 10, 10,           // x, y
 128, 64};         // w, h
```

If a canvas is greater than 252 pixels in width, we must instead create two `C_IMAGE_EXT` controls to display two halves (left and right) of the same canvas. These controls should be defined as follows:

- The width of the left control should be 252 pixels.
- The width of the right control should be (total width - 252) pixels.
- The `x` position of the right control should be 252 pixels greater than the `x` position of the left control.
- The `param` value of the right control should point to `(canvas + 10)` rather than `(canvas)`.

For example, for a canvas 304x16 pixels in size:

```
_transfer Ctrl c_canvas1 = {1, C_IMAGE_EXT, -1, (unsigned short)canvas, 10, 10, 252, 16};
_transfer Ctrl c_canvas2 = {2, C_IMAGE_EXT, -1, (unsigned short)canvas + 10, 10, 10, 16};
```

Initializing canvases

Before opening the window in which the canvas is displayed, we must initialize it with the `Gfx_Init()` function:

```
void Gfx_Init(char* canvas, unsigned short w, unsigned char h);
```

For example, to initialize the 128x64 pixel canvas defined at the start of this section:

```
Gfx_Init(canvas, 128, 64);
```

Before drawing to a canvas, we must select it as the active canvas using `Gfx_Select()`. If desired, we can maintain multiple canvases and switch which is being drawn to using this function.

```
void Gfx_Select(char* canvas);
```

For example, to select the canvas defined at the start of this section:

```
Gfx_Select(canvas);
```

Refreshing the display

Updating the canvas with a graphics function does not automatically update the screen to display the changes. This means that graphics are automatically double-buffered, reducing flicker, but also that we must manually tell SymbOS to redraw the canvas (which is often the slowest part of the whole process). The simplest method is to redraw the entire canvas by using `Win_Redraw()` system call to redraw its control:

```
Win_Redraw(winID, 1, 0); // redraw control ID 1 (the canvas's C_IMAGE_EXT control above)
```

However, redrawing a full canvas can visibly take a fraction of a second on a 4 MHz processor, which is nonideal for games with fluid animation. An alternative is to use the `Win_Redraw_Area()` system call to redraw only the part of the canvas that has changed. This is much faster, but requires us to keep track of the coordinates that have changed and issue the correct call:

```
Win_Redraw_Area(winID, 1, 0, 10, 10, 16, 16); // redraw only the 16x16 pixel region
                                                // starting at x=10, y=10 in control ID 1
```

Important: The coordinates for `Win_Redraw_Area` are relative to the *window content*, not the canvas! This means that, if the canvas is not at position $x = 0$, $y = 0$ in the window, the coordinates passed to `Win_Redraw_Area` must have the canvas position added in.

It is important to understand that refreshes occur asynchronously. We are telling the desktop manager to refresh the control at its earliest convenience, but our code will keep running in the meantime, and there is no way to determine when the refresh has actually occurred (except that, once the screen is refreshed, there will be a delay of at least 20 milliseconds before it can be refreshed again). This is helpful for realtime games, where waiting for a refresh would slow everything down, but can also create subtle traps. In particular, if we are continuously updating the canvas, what is actually shown on the screen will not be the contents of the canvas *at the time the refresh was requested*, but the contents of the canvas some fraction of a second later *when the refresh is actually performed*. Care should therefore be taken not to “get ahead of” the screen refresh by, e.g., plotting a sprite, requesting a refresh, and then immediately moving it somewhere else.

Drawing functions

Gfx_Pixel()

```
void Gfx_Pixel(unsigned short x, unsigned char y, unsigned char color);
```

Plots a single pixel with the color `color` to the pixel coordinates `x`, `y` on the currently active canvas. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_Safe_Pixel()

```
void Gfx_Safe_Pixel(unsigned short x, unsigned char y, unsigned char color);
```

Equivalent to `Gfx_Pixel()`, but performs a simple bounds check and skips plotting the pixel if the coordinates are invalid for the current canvas. This function should be used instead of `Gfx_Pixel()` in cases where it is not known ahead of time that the pixel coordinates are valid.

Gfx_Value()

```
unsigned char Gfx_Value(unsigned short x, unsigned char y);
```

Returns the color of the pixel on the active canvas at pixel coordinates `x`, `y`.

Gfx_Line()

```
void Gfx_Line(unsigned short x0, unsigned char y0,  
              unsigned short x1, unsigned char y1,  
              unsigned char color);
```

Draws a 1-pixel-thick line of color `color` from point `x0`, `y0` to point `x1`, `y1` on the current canvas. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_LineB()

```
void Gfx_LineB(unsigned short x0, unsigned char y0,  
               unsigned short x1, unsigned char y1,  
               unsigned char color, unsigned char bits);
```

Equivalent to `Gfx_Line()`, but only pixels corresponding to bits set in the bitmask `bits` will be drawn (starting from the most significant bit and repeating after 8 bits/pixels). For example, to draw a dashed line (two pixels on + two pixels off, repeating), we could use the bitmask `0xCC`.

Gfx_HLine()

```
void Gfx_HLine(unsigned short x, unsigned char y, unsigned short w, unsigned char color);
```


Draws a 1-pixel-thick line horizontal line of color `color` from point `x`, `y` to the point `w` pixels to the right. This is faster than using `Gfx_Line()` for cases where the line is known to be horizontal. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_VLine()

```
void Gfx_VLine(unsigned short x, unsigned char y, unsigned short h, unsigned char color);
```

Draws a 1-pixel-thick line vertical line of color `color` from point `x`, `y` to the point `h` pixels below. This is faster than using `Gfx_Line()` for cases where the line is known to be vertical. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_Box()

```
void Gfx_Box(unsigned short x0, unsigned char y0,  
             unsigned short x1, unsigned short y1,  
             unsigned char color);
```

Draws an unfilled rectangle of color `color` whose upper left corner is at the pixel coordinates `x0`, `y0` and whose lower right corner is at the pixel coordinates `x1`, `y1`. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_BoxF()

```
void Gfx_BoxF(unsigned short x0, unsigned char y0,  
              unsigned short x1, unsigned short y1,  
              unsigned char color);
```

Draws a filled rectangle of color `color` whose upper left corner is at the pixel coordinates `x0`, `y0` and whose lower right corner is at the pixel coordinates `x1`, `y1`. No bounds-checking is performed, so be sure that the coordinates are actually valid. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_Circle()

```
void Gfx_Circle(signed short x0, signed short y0, signed short radius, unsigned char color);
```

Draws an unfilled circle with the color `color`, a radius of `radius` pixels, and its centerpoint at `x0`, `y0`. The circle may overlap the edges of the canvas. In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Gfx_Text()

```
void Gfx_Text(unsigned short x, unsigned char y, char* text, unsigned char color, char* font);
```

Draws the text string `text` at the pixel coordinates `x`, `y` on the currently active canvas, using color `color` and the font at the address `font`. If `font = 0`, the system font will be used. No clipping is performed, so be sure that the coordinates are actually valid (including that the text will not overflow the right margin). In 4-color modes, higher colors will be automatically rendered down to 4 colors.

Drawing text in 16-color mode is currently slower than in 4-color mode. Drawing text using the system font is also a bit slower than using a custom font because of the need for banked memory access. (This is the opposite of how system text controls behave, where using the system font is faster.) Unlike system controls, it does not matter which segment custom font data is stored in.

A description of the font format can be found in the SymbOS Developer Documentation; fonts can be created using the SymbOS Font Editor application.

Gfx_ScrollX()

Currently only available in development builds of SCC.

```
void Gfx_ScrollX(int pixels);
```

Scrolls the entirety of the currently active canvas `pixels` pixels right (when `pixels > 0`) or left (when `pixels < 0`). `pixels` will be rounded down to the nearest 4 pixels in 4-color mode or to the nearest 2 pixels in 16-color mode; this limitation enables scrolling to be a very fast byte-copy operation (even faster than a window refresh).

The margin of the canvas that was just “scrolled onscreen” (e.g., the rightmost 4 columns of pixels when scrolling by -4) will be initially filled with garbage; we are responsible for overdrawing it with any content that should become visible as a result of the scroll.

Gfx_ScrollY()

Currently only available in development builds of SCC.

```
void Gfx_ScrollY(int pixels);
```

Scrolls the entirety of the currently active canvas `pixels` pixels down (when `pixels > 0`) or up (when `pixels < 0`). Unlike `Gfx_ScrollX()`, scrolling will be pixel-perfect, without rounding down. This is a very fast byte-copy operation (even faster than a window refresh).

The margin of the canvas that was just “scrolled onscreen” (e.g., the bottom 4 rows of pixels when scrolling by -4) will be initially filled with garbage; we are responsible for overdrawing it with any content that should become visible as a result of the scroll.

Gfx_Clear()

```
void Gfx_Clear(char* canvas, unsigned char color);
```

Clears the entirety of canvas `canvas` to color `color`. In 4-color modes, higher colors will be automatically rendered down to 4 colors. (This is faster than filling the entire canvas using `Gfx_BoxF()`.)

Sprite functions

We can load images in SGX format from disk and plot them directly on the canvas, or copy portions of the canvas as images for saving to disk or subsequent plotting. Images handled in this way **must have a width that is a multiple of 4 pixels** and cannot exceed 252x255 pixels in width/height. (This means that we cannot load, e.g., complete desktop backgrounds as sprites.)

Converting images

Images can be converted to SGX format with software such as [MSX Viewer 5](#) (classic version). The graphics library is able to correctly handle the erroneous header information MSX Viewer 5 creates for smaller 16-color images, so most images created by MSX Viewer 5 should be usable without modification.

Alternatively, the command-line tool `gfx2sgx` provided with the desktop version of SCC can be used to convert graphics in a variety of formats (`.bmp`, `.jpg`, `.png`, `.gif`, `.tga`, and more) to SGX format, e.g.:

```
gfx2sgx image.png
```

This will convert `image.png` into a 16-color SGX image in the same folder (`image.sgx`). Images are converted pixel-perfect by Euclidean similarity to the SymbOS palette colors, so any necessary scaling or recoloring must be done using other graphics software (such as GIMP or Photoshop) prior to conversion. A typical workflow might be to create a custom color palette matching the [default SymbOS palette](#), create pixel art using this palette, export it as an RGB `.png` file, and run it through `gfx2sgx`. For more complicated conversions (e.g., converting a true-color JPEG photo to a 16-color SGX image), automatic palette-conversion tools like GIMP's indexed color quantization may be useful (Image > Mode > Indexed, then convert back to RGB with Image > Mode > RGB before exporting to `.png`).

Image sets and masks

`gfx2sgx` allows the creation of multi-image sets ("tilesets" or "sprite sheets") from a single image. To create an image set, specify the width and height of each tile on the command-line. For example, to split the image `tiles.png` evenly into 24x16 pixel tiles:

```
gfx2sgx tiles.png 24 16
```

The resulting `.gfx` file can be loaded using `Gfx_Load_Set()` and individual tiles plotted using `Gfx_Put_Set()`, described below. Creating tilesets in this way prevents us from having to load numerous small files from disk. Tiles are extracted from the original image left-to-right, top-to-bottom, with the first tile (from the upper left of the original image) having the index 0.

A standard technique for creating transparent sprites is to use bitwise plotting modes (see `Gfx_Put()`, below). First, we plot a "mask" image with `PUT_AND` (bitwise AND); this image should consist of color 0 for all pixels that are to be overwritten, and color 15 for all pixels that are to be transparent. Then, we plot a "foreground" image with `PUT_OR` (bitwise OR); this image should consist of color 0 for all pixels that are to be transparent, and normal colors for all pixels that are to be overwritten. `gfx2sgx` can automatically split a transparent `.png` image into the necessary mask and foreground layers using the `-m` option. For example, to split the transparent 48x32-pixel image `tiles.png` evenly into six 16x16-pixel tiles, each with a mask layer:

```
gfx2sgx tiles.png -m 16 16
```

The mask for each tile will be placed immediately before it, i.e., the image:



Figure 1: Tile image

...becomes:



Figure 2: Extracted tiles with mask

The resulting tiles can be drawn transparently by first plotting the mask tile with mode `PUT_AND` and then plotting the foreground tile with `PUT_OR`:

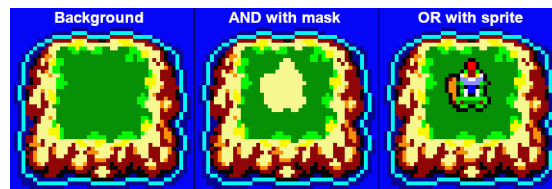


Figure 3: Technique demonstration

Gfx_Load()

```
unsigned char Gfx_Load(char* filename, char* buffer);
```

Loads a graphics asset file (in 4-color or 16-color SGX format) from the path `filename` into the buffer `buffer`. If the path `filename` is relative rather than absolute, it will be treated as relative to the current executable's path. If the current screen mode does not match the format of the file, it will be converted automatically to match using `Gfx_Prep()` (listed below).

`buffer` should be a char array the size of the file, in bytes (rounding up to the nearest 128 bytes), or else $(\text{width} * \text{height} / 2) + 4$ bytes long, whichever is greater.

Return value: On success, returns 0. On failure, sets `_fileerr` and returns 1.

Gfx_Load_Set()

```
unsigned char Gfx_Load_Set(char* filename, char* buffer);
```

Loads an image set file created with `gfx2sgx` (see above) from the path `filename` into the buffer `buffer`. If the path `filename` is relative rather than absolute, it will be treated as relative to the current executable's path. If the current screen mode does not match the format of the tiles, they will be converted automatically to match using `Gfx_Prep()` (listed below).

buffer should be a char array the size of the file, in bytes (rounding up to the nearest 128 bytes).

Return value: On success, returns 0. On failure, sets `_fileerr` and returns 1.

Gfx_Put()

```
void Gfx_Put(char* image, unsigned short x, unsigned char y, unsigned char mode);
```

Plots the image located in the buffer `image` to the pixel coordinates `x`, `y`. The plotting mode `mode` is one of:

- `PUT_SET` - overwrites the pixels on the canvas with the pixels in the image.
- `PUT_AND` - performs a bitwise AND between the pixels on the canvas and the pixels in the image.
- `PUT_OR` - performs a bitwise OR between the pixels on the canvas and the pixels in the image.
- `PUT_XOR` - performs a bitwise XOR between the pixels on the canvas and the pixels in the image.

No bounds-checking or clipping is performed, so be sure that the coordinates are actually valid.

Plotting is fast when the image can be copied directly without bit-shifting or modification (i.e., when `mode` is `PUT_SET` and `x` is a multiple of 4, or, for 16-color displays, a multiple of 2), but somewhat slower in all other cases. For performance-critical applications, a common trick to speed up sprite-plotting is to maintain four different versions of a sprite, each shifted horizontally by 1 pixel, so that sprites can always be plotted at `x` coordinates that are multiples of 4. However, the performance gains of this method when using a `mode` other than `PUT_SET` may be marginal.

Gfx_Put_Set()

```
void Gfx_Put_Set(char* image, unsigned short x, unsigned char y,  
                unsigned char mode, unsigned char tile);
```

Equivalent to `Gfx_Put()`, above, but plots tile number `tile` (where the first tile is `tile = 0`) from an image set loaded with `Gfx_Load_Set()`.

Gfx_Get()

```
void Gfx_Get(char* image, unsigned short x, unsigned char y, unsigned char w, unsigned char h);
```

Creates an image from the portion of the active canvas starting at pixels coordinates `x`, `y`, with width `w` pixels and height `h` pixels, storing the resulting image in the char buffer `image`. `w` must be a multiple of 8, and this buffer should be at least $(w * h) / 2 + 4$ bytes long. The resulting image will be in the same format expected by `Gfx_Put()`. No bounds-checking is performed, so be sure that the coordinates are actually valid.

Gfx_Save()

```
unsigned char Gfx_Save(char* filename, char* buffer);
```

Saves the image from the buffer `buffer` to the path `filename`. If the path `filename` is relative rather than absolute, it will be treated as relative to the current executable's path. Images will be saved in 4-color or 16-color SGX format, depending on the current screen mode.

Note that `buffer` is a `char` buffer into which an image has been loaded using `Gfx_Load()` or `Gfx_Get()`, *not* a complete canvas.

Return value: On success, returns 0. On failure, sets `_fileerr` and returns 1.

Gfx_Prep()

```
void Gfx_Prep(char* buffer);
```

If the bit depth of the image located in the buffer `buffer` does not match the current screen mode (i.e., 4-color vs. 16-color), or the image has the wrong type of header, converts it to the correct format. Since 4-color images require more space when converted to 16-color, `buffer` should be at least $(\text{width} * \text{height} / 2) + 4$ bytes long regardless of the original size of the image data.

This function is normally called automatically by `Gfx_Load()`, but we can also invoke it manually to convert image data embedded in the source as a character array into the correct format. E.g.:

```
_transfer char icon[35] = {0x02, 0x08, 0x08, 0xFF, 0xFF, 0xF8, 0xF1, 0xF8, 0xF1,
                          0x8F, 0x1F, 0x8F, 0x1F, 0x8F, 0x1F, 0x8F, 0x1F, 0xFF, 0xFF};

int main(int argc, char* argv[]) {
    /* ... other graphics setup here ... */
    Gfx_Prep(icon);
    Gfx_Put(icon, 0, 0);
    /* ... other commands here ... */
}
```

Character arrays like this can be created by running `gfx2sgx` with the `-c` option, which will output the image data as a character array in a `.c` file instead of as binary data in a `.sgx` file:

```
gfx2sgx icon.png -c
```

Gfx_Prep_Set()

```
void Gfx_Prep_Set(char* buffer);
```

Equivalent to `Gfx_Prep()`, but preps image sets instead of single images.

Gfx_TileAddr()

Currently only available in development builds of SCC.

```
char* Gfx_TileAddr(char* image, unsigned char tile);
```

Returns the address of tile number `tile` within the image set at the address `image`.

Raw images

Currently only available in development builds of SCC.

Raw image controls

Most of the graphics library deals with canvases, which allow graphics to be edited and redraw on the fly. However, if we just want to display static images as part of a form, we can often achieve this more efficiently by simply creating a `C_IMAGE` or `C_IMAGE_EXT` control and setting the parameter to the address of the static image data.

Step 1: Generate a compatible image file

For 16-color images, use `gfx2sgx` to convert the desired image into an .SGX file. For 4-color images, use `MSX Viewer 5` (classic version) or use `gfx2sgx` with the `-4` command-line option to force 4-color mode:

```
gfx2sgx image.png -4
```

Step 2: Load the image into a buffer

Load the image data into a suitably sized buffer that does not cross a 16KB boundary, e.g., in the **data** or **transfer** segments. The utility function `Gfx_Load_Raw()` is the recommended way to do this, since this will perform the necessary updates to the `extended graphics header` automatically.

(Note that, since we will not be manipulating this image after it is loaded, we can technically store it in `banked memory` if needed. For this reason, all raw image functions allow specifying the memory bank. If banked memory is used, remember to set the `.bank` property of the image control correctly and free the reserved memory with `Mem_Release()` before exit.)

Step 3: Create an image control

For a 4-color image, use a `C_IMAGE` control and set the parameter to the address of the image buffer:

```
_data char imgbuf[256]; // load the image data into this buffer
_transfer Ctrl c_image1 = {1, C_IMAGE, -1, (unsigned short)imgbuf, 10, 10, 24, 24};
```

For a 16-color image, use a `C_IMAGE_EXT` control and set the parameter to the address of the image buffer:

```
_data char imgbuf[384]; // load the image data into this buffer
_transfer Ctrl c_image1 = {1, C_IMAGE_EXT, -1, (unsigned short)imgbuf, 10, 10, 24, 24};
```

Gfx_Load_Raw()

```
unsigned char Gfx_Load_Raw(char* filename, unsigned char bank, char* buffer);
```

Loads a raw image file (in 4-color or 16-color SGX format) from the path `filename` into memory bank `bank`, address `buffer`. If the path `filename` is relative rather than absolute, it will be treated as relative to the current executable's path. `buffer` should be a char array the size of the file, in bytes (rounding up to the nearest 128 bytes), which does not cross a 16KB segment boundary

The difference between this function and the regular `Gfx_Load()` is that this function prepares the image file for direct use by a `control`, not as a graphics asset that can be plotted on a `canvas` using `Gfx_Put()`. If the image is meant to be plotted on a canvas, use `Gfx_Load()` instead.

Return value: On success, returns 0. On failure, sets `_fileerr` and returns 1.

Gfx_Load_Set_Raw()

```
unsigned char Gfx_Load_Set_Raw(char* filename, unsigned char bank, char* buffer);
```

Loads an [image set file](#) created with `gfx2sgx` from the path `filename` into memory bank `bank`, address `buffer`. If the path `filename` is relative rather than absolute, it will be treated as relative to the current executable's path. `buffer` should be a char array the size of the file, in bytes (rounding up to the nearest 128 bytes).

The difference between this function and the regular `Gfx_Load_Set()` is that this function prepares the image tiles for direct use by [controls](#), not as graphics assets that can be plotted on a [canvas](#) using `Gfx_Put_Set()`. If the images are meant to be plotted on a canvas, use `Gfx_Load_Set()` instead.

This function is a convenient way to load a large number of raw images at once. For the purpose of setting image control parameters, the address of each tile in the tileset can be determined using `Gfx_TileAddr_Raw()`.

Return value: On success, returns 0. On failure, sets `_fileerr` and returns 1.

Gfx_Prep_Raw()

```
void Gfx_Prep_Raw(unsigned char bank, char* buffer);
```

If the memory at bank `bank`, address `buffer` contains a raw 16-color image, prepares the image's extended header so that it can be drawn correctly by a `C_IMAGE_EXT` control. (This function is normally called automatically by `Gfx_Load_Raw()`, but we can also invoke it manually to convert image data [embedded in the source as a character array](#).)

Gfx_Prep_Set_Raw()

```
void Gfx_Prep_Set_Raw(unsigned char bank, char* buffer);
```

Equivalent to `Gfx_Prep_Raw()`, but preps image sets instead of single images.

Gfx_TileAddr_Raw()

```
char* Gfx_TileAddr_Raw(unsigned char bank, char* image, unsigned char tile);
```

Returns the address of tile number `tile` within the image set located in memory bank `bank`, address `image`. (The only difference between this and `Gfx_TileAddr()` is that the bank may be specified.)

Advanced topics

Moving sprites

Because `graphics.h` implement raw canvases and images, it does not have any built-in features for redrawing the background behind a sprite after it is moved or deleted. Anything drawn to a canvas

will simply stay there until overdrawn by something else. However, several simple techniques can be employed to create moveable sprites, depending on the specific needs of the application:

- Before plotting a sprite, use `Gfx_Get()` to copy the background behind it into a temporary buffer. Then, when the sprite needs to be moved, erase it by plotting the old background on top of it with `Gfx_Put()`.
- For graphics based on multiple layers of regularly-spaced tiles (like an RPG), simply replot any affected tiles from the bottom up with `Gfx_Put()`, redrawing the background over the sprite.
- A flexible (but memory-hungry) solution is to maintain two canvases: a “background” canvas containing the background, and a “visible” canvas actually shown in the window. To move a sprite, use `Gfx_Select()`, `Gfx_Get()`, and `Gfx_Put()` to copy the relevant parts of the “background” canvas over the sprite’s location on the “visible” canvas.
- If only a few sprites are needed, consider drawing them using separate `C_IMAGE_TRANS` controls pointing to [raw images](#).

Memory problems

In practice, the biggest challenge when developing graphics-heavy programs (like games) is running out of memory. For speed, the graphics library requires that all canvases and graphics assets be located in the application’s main 64KB address space (the **code**, **data**, and **transfer** segments). This is fine if we are only using the graphics library for small accents like icons or toolbars, but imagine an RPG that wants to keep a 230x140 canvas, a set of 100 16x16 tiles, and a set of 32 16x16 masked sprites in memory at the same time. That leaves only 19KB for the game code itself, assuming a byte-perfect memory map—not much to work with!

Further problems arise from the need to keep canvases within a single 16KB memory segment. (This is necessary to prevent graphics corruption on some platforms, like the MSX.) A common (if confusing) way to get into trouble is to have a large **code** segment, a large canvas in the **data** segment, and a comparatively empty **transfer** segment. SymbOS may then have trouble arranging the segments in a legal order even if the total file size is <64KB.

To understand what is going on, imagine a game with a 40KB **code** segment, a 16KB **data** segment, and a 3KB **transfer** segment. The **code** segment is loaded first, while the **transfer** segment must be in the uppermost 16KB of memory:

0x0000	0x4000	0x8000	0xC000
CCCCC	CCCCC	CCC	TT

Where does the **data** segment (DDDDDD) go? In theory it could go between the **code** and **transfer** segments, but this would result in it crossing a 16KB segment boundary, which is not allowed:

0x0000	0x4000	0x8000	0xC000
CCCCC	CCCCC	CCDDDD	DDDTT

So, SymbOS throws an “out of memory” error even though the total file size is <64KB. The easiest solution to this problem is usually to move some data from the **code** segment to the **transfer** segment:

0x0000	0x4000	0x8000	0xC000
CCCCC	CCCCC	DDDDDD	TTTTT







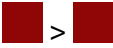









But what if we’re just hitting the 64KB limit? Some suggestions, from least to most complicated:

















- Eliminate all unnecessary canvas and buffer space. For example, in games, it is often unnecessary to actually draw the entire play field using a single large canvas. If there is a significant amount of whitespace, consider breaking the canvas up into several smaller canvases and drawing the whitespace using a C_AREA control instead (which takes almost no memory).
- Store graphics assets in [banked memory](#) and only load them into the application’s main 64KB address space when needed. (This works best when draw-time is not critical, or when we only need a portion of all available graphics assets at one time.)
- For repeating elements, instead of using one large C_IMAGE_EXT control pointing to one large canvas, use several small C_IMAGE_EXT controls pointing to the same small canvas. Overlapping C_IMAGE_TRANS controls could also be used to create transparent sprites without requiring a separate mask image.
- Most complicated, but theoretically useful for game engines: perform all graphics rendering in a dedicated [process](#) with its own 64KB address space, telling the rendering process what to do using [inter-process messaging](#) or a shared memory space in banked memory.

Reference

Color palette

The default SymbOS color palette is shown below, along with approximate HTML hex codes and equivalent C constants (from `symbos/windows.h`). In 4-color modes, colors greater than 3 will be automatically rendered down to colors 0-3 following the pattern shown on the right side of the color preview (i.e., color & 3). Constants in International English spellings (e.g., `COLOUR_WHITE`) are also available.

Color	ID	Keyword	Hex
 > 	0	COLOR_YELLOW	#F7F790
 > 	1	COLOR_BLACK	#060606
 > 	2	COLOR_ORANGE	#F79006
 > 	3	COLOR_RED	#F79006
 > 	4	COLOR_CYAN	#06F7F7
 > 	5	COLOR_DBLUE	#060690
 > 	6	COLOR_LBLUE	#9090F7
 > 	7	COLOR_BLUE	#0606F7

Color	ID	Keyword	Hex
 > 	8	COLOR_WHITE	#F7F7F7
 > 	9	COLOR_GREEN	#069006
 > 	10	COLOR_LGREEN	#06F706
 > 	11	COLOR_MAGENTA	#F706F7
 > 	12	COLOR_LYELLOW	#F7F706
 > 	13	COLOR_GRAY	#909090
 > 	14	COLOR_PINK	#F79090
 > 	15	COLOR_LRED	#F70606

Network Library

Note: The network library as described below is currently only available in development builds of SymbOS. Slightly different versions of `Net_Init()` and the raw TCP/UDP/DNS functions can be found in older releases in the header file `symbos/network.h`.

Note: All network functions are thread-safe, and multiple threads can have connections open simultaneously (although the network daemon usually only permits sockets to be accessed by the thread that originally opened them). If a thread calls a network function that is currently in use by another thread, it will automatically wait its turn.

Contents

- [Using the library](#)
- [Function reference](#)
 - [HTTP functions](#)
 - [DNS functions](#)
 - [TCP functions](#)
 - [UDP functions](#)
 - [FTP functions](#)
 - [Helper functions](#)
- [Reference](#)
 - [Error codes](#)

Using the library

To use the library, include the `network.h` header:

```
#include <network.h>
```

Additionally, use the `-lnet` option when compiling to specify that the executable should be linked with the network library `libnet.a`:

```
cc source.c -lnet
```

Network capabilities are only available if an appropriate network daemon is running. Use `Net_Init()` to initialize the network and connect to the daemon.

Network errors are recorded in the global variable `_neterr`, documented [below](#).

Net_Init()

```
signed char Net_Init(void);
```

Initializes the network interface, if present. This should be called before using any other network functions.

Return value: On success, sets `_netpid` to the process ID of the network daemon and returns 0. On failure, sets `_neterr` and returns -1.

HTTP functions

HTTP_GET()

```
int HTTP_GET(char* url, char* dest, unsigned short maxlen, char* headers,  
             unsigned char keep_headers);
```

Executes a complete HTTP GET request, downloading whatever content is returned from the URL `url` into the buffer at the address `dest`. A maximum of `maxlen` characters will be written, discarding any extra. If `maxlen = NET_FILE`, `dest` will instead be treated as the absolute path of a file to save the response to (with no length limit). If `keep_headers` is nonzero, the response will include the full HTTP headers as well as the body content.

`headers` can be optionally used to specify additional HTTP header requests in the outgoing request (use 0 for no custom headers). Each custom header line should be followed by the HTTP-standard line break `\r\n` (even the last one):

```
status = HTTP_GET("http://numbersapi.com", buffer, sizeof(buffer),  
                  "Accept: text/plain\r\nAccept-Language: en-US\r\n", 1);
```

Return value: On success, writes the response to the buffer `dest` (or the file indicated by `dest`) and returns the HTTP response status code (e.g., 200 “OK”, 404 “Not Found”). On failure, sets `_neterr` and returns -1. If a response is received but it does not contain a valid HTTP response status code, returns 0.

HTTP_POST()

```
int HTTP_POST(char* url, char* dest, unsigned short maxlen, char* headers,  
              char* body, unsigned short bodylen, unsigned char keep_headers);
```

Executes a complete HTTP POST request, downloading whatever content is returned from the URL `url` into the buffer at the address `dest`. A maximum of `maxlen` characters will be written, discarding any extra. If `maxlen = NET_FILE`, `dest` will instead be treated as the absolute path of a file to save the response to (with no length limit). If `keep_headers` is nonzero, the response will include the full HTTP headers as well as the body content.

POST data are passed in the buffer `body`, where `bodylen` is the length of the buffer. (It is necessary to specify `bodylen` manually because POST requests may include binary data.)

`headers` can be optionally used to specify additional HTTP header requests in the outgoing request (use 0 for no custom headers). Each custom header line should be followed by the HTTP-standard line break `\r\n` (even the last one).

```
status = HTTP_POST("http://example.com", "A:\\EXAMPLE.HTM", NET_FILE,  
                   0, "username=test&password=123", 26, 1);
```

Return value: On success, writes the response to the buffer `dest` (or the file indicated by `dest`) and returns the HTTP response status code (e.g., 200 “OK”, 404 “Not Found”). On failure, sets `_neterr` and returns -1. If a response is received but it does not contain a valid HTTP response status code, returns 0.

Proxy servers

By default, HTTP functions support only unencrypted connections (URLs beginning with `http://`). Native support for SSL-encrypted connections (URLs beginning with `https://`) is not practical on 8-bit hardware. However, it is possible to access HTTPS sites by using a modern go-between computer running a web proxy such as [WebOne](#). To route requests through a proxy, place the IP address of the proxy computer into the global buffer `_http_proxy_ip` (using, e.g., `DNS_Resolve()`) and the proxy's port into the global variable `_http_proxy_port`. For example:

```
DNS_Resolve(_sybank, "192.168.0.19", _http_proxy_ip);  
_http_proxy_port = 1234;
```

(In practice, of course, it is bad practice to hard-code proxy addresses like this—this should be a setting the user can edit, so the app can keep working even if the original proxy server goes down.) To stop routing requests through a proxy, clear `_http_proxy_ip` to all zeros.

Tracking progress

Download speeds on 8-bit hardware are not very fast, typically on the order of 56-112 kbps (that is, a bit faster than a dialup modem). At this speed, a multi-megabyte file can take several minutes to download, so it is helpful to have a way to track the progress of a download and interrupt it if needed.

This is most easily done by running `HTTP_GET()` or `HTTP_POST()` on a [separate thread](#). When running on a different thread, we can interrupt execution by writing a nonzero value to the global variable `_http_abort`. `HTTP_GET()` and `HTTP_POST()` will then stop downloading at their earliest convenience, writing/saving only what they have downloaded so far.

`HTTP_GET()` and `HTTP_POST()` also record their status in the global variable `_http_status`, with values matching the following constants:

- `HTTP_READY` - finished / not yet started
- `HTTP_LOOKUP` - performing DNS lookup and initial preparation
- `HTTP_CONNECTING` - connecting to remote server
- `HTTP_SENDING` - sending request
- `HTTP_WAITING` - waiting for first response packet
- `HTTP_DOWNLOADING` - downloading response

While downloading body content, the global **unsigned long** variable `_http_total` contains the total expected content length (in bytes) and the global **unsigned long** variable `_http_progress` contains the length received so far (in bytes).

DNS functions

DNS_Resolve()

```
signed char DNS_Resolve(unsigned char bank, char* addr, char* ip);
```

Performs a DNS lookup and attempts to resolve the host IP/URL stored in the string at `bank`, address `addr` into an IPv4 address, which will be stored in the 4-byte buffer at `ip`. (Note that the

URL should *not* be prefixed with a protocol like `http://`; the easiest way to guarantee this is to use `Net_SplitURL()` and then run `DNS_Resolve()` only on the extracted hostname.)

Return value: On success, stores the IP address in the 4-byte buffer pointed to by `ip` and returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `DNS_Resolve (DNSRSV)`.

DNS_Verify()

```
unsigned char DNS_Verify(unsigned char bank, char* addr);
```

Verifies whether the IP/URL stored in the string at `bank`, address `addr` is a valid IP or domain address. This function does not interact with the network hardware, so can be used to quickly determine whether an address is valid before initiating a full network request.

Return value: On success, returns `DNS_IP` for a valid IP address, `DNS_DOMAIN` for a valid domain address, or `DNS_INVALID` for an invalid address. On failure, sets `_neterr` and returns `DNS_INVALID`.

SymbOS name: `DNS_Verify (DNSVFY)`.

TCP functions

TCP is the primary low-level network protocol supported by SymbOS network hardware and underlies other protocols such as Telnet, HTTP, and FTP. In a TCP connection, a **client** requests to establish a two-way connection with a **server** (located at a specific IPv4 address and port number). Once established, both parties can send and receive streams of data, with the network hardware ensuring that the streams are delivered completely, correctly, and in the right order. The connection remains open until it is closed by one party (e.g., at the end of a Telnet session or HTTP request).

- To open a TCP connection, use `TCP_OpenClient()` or `TCP_OpenServer()`.
- Once open, data can be sent to the other party at any time with `TCP_Send()`.
- The network daemon will alert us to the arrival of new data (or a change in connection status) by sending message ID `NET_TCPEVT`. We can watch for this message in our main event loop and use `TCP_Event()` to understand its contents.
 - When `TCP_Event()` yields `NetStat.bytesrec > 0`, incoming data bytes are waiting; we can download them into a buffer using `TCP_Receive()`.
 - When `TCP_Event()` yields `NetStat.status = TCP_CLOSED`, the remote server has closed the connection (although there may still be data in the incoming buffer waiting to be downloaded). Once we've downloaded everything we want, we can free the socket with `TCP_Close()`.
- To disconnect from our end, use `TCP_Disconnect()`.

For a complete example of implementing a TCP connection, see `fast.c` in the **sample** folder.

TCP_OpenClient()

```
signed char TCP_OpenClient(char* ip, signed short lport, unsigned short rport);
```

Opens a client TCP connection to the IPv4 address `ip` (stored in a 4-byte buffer, one byte per octet) on local port `lport`, connecting to remote port `rport`. (For client connections, `lport` should usually be set to -1 to obtain a dynamic port number.) The function will then wait for the socket status to become `TCP_OPENED`, indicating a successful connection. (If the server refuses the connection, this function will fail with `_neterr = ERR_CONNECT`.)

Return value: On success, returns a socket handle to the new connection. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Open (TCPOPEN)`.

TCP_OpenServer()

```
signed char TCP_OpenServer(unsigned short lport);
```

Opens a TCP server listening on local port `lport`.

Return value: On success, returns a socket handle to the new server. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Open (TCPOPEN)`.

TCP_Event()

```
void TCP_Event(char* msg, NetStat* obj);
```

A utility function for parsing a `NET_TCPEVT` message into a readable status report. `msg` is the address of the 14-byte message buffer containing the message; `obj` is the address of a `NetStat` struct to store the results, with the format:

```
typedef struct {
    unsigned char socket;    // socket this message pertains to
    unsigned char status;    // status (see below)
    unsigned char ip[4];     // remote IP address
    unsigned short rport;    // remote port
    unsigned char datarec;   // 1 = data received, 0 = none
    unsigned short bytesrec; // received bytes waiting in buffer
} NetStat;
```

status may be one of `TCP_OPENING`, `TCP_OPENED`, `TCP_CLOSING`, or `TCP_CLOSED`.

TCP_Status()

```
signed char TCP_Status(unsigned char handle, NetStat* obj);
```

Returns the status of the TCP connection associated with the socket `handle` and stores the results in the `NetStat` struct `obj` (see `TCP_Event()`). This is equivalent to `TCP_Event()` but can be used to query the status of the connection at any time, not just after receiving a `NET_TCPEVT` message.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: TCP_Status (TCPSTA).

TCP_Send()

```
signed char TCP_Send(unsigned char handle, unsigned char bank, char* addr, unsigned short len);
```

Sends data from memory bank `bank`, address `addr`, to the host associated with the socket `handle`. A total of `len` bytes will be sent.

When running on a [separate thread](#), the global **unsigned long** variable `_tcp_progress` contains the number of bytes sent so far. Writing a nonzero value to the global variable `_tcp_abort` will cause `TCP_Send()` to abort the transfer at its earliest convenience, failing with `_neterr = ERR_TRUNCATED`.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: TCP_Send (TCPSND).

TCP_Receive()

```
signed char TCP_Receive(unsigned char handle, unsigned char bank, char* addr,
                        unsigned short len, TCP_Trans* obj);
```

Moves data which has been received from the remote host associated with socket `handle` to the memory at bank `bank`, address `addr`. Up to `len` bytes will be moved (or the actual amount in the buffer, whichever is less).

`obj` is an optional pointer to a `TCP_Trans` struct, into which additional information about the transfer will be loaded. This parameter may be set to `NULL` to omit this information. The structure of the struct is:

```
typedef struct {
    unsigned short transferred; // bytes transferred to destination
    unsigned short remaining;   // bytes remaining in the buffer
} TCP_Trans;
```

A small collection of subtleties:

- Note that setting `len` to the total number of available bytes and calling `TCP_Receive()` is not guaranteed to leave the buffer empty, because the network daemon can receive additional bytes at any time. The main situation where this matters is when the app wants to empty the buffer and wait for a `NET_TCPEVT` message from the network daemon alerting it when new data arrives. However, the network daemon will only send a `NET_TCPEVT` message when adding data *to an empty buffer*, so if the initial `TCP_Receive()` call does not actually empty the buffer completely, no message will arrive. One way to avoid this is to check the returned `TCP_Trans.remaining` value and loop, repeatedly calling `TCP_Receive()`, until this value is actually zero; alternatively, the app can not rely on receiving a message.
- Note that, because TCP data are sent as a continuous stream, fully clearing the TCP buffer does *not* necessarily mean that we have processed all the data the other party intends to send! It is the program's responsibility to understand (from the content of the data stream) when a pause

in receiving data means “request finished, please process it” versus “more data coming soon, please wait.” Likewise, we need to be careful about assuming that any given call of `TCP_Receive()` will yield a “complete” response, as incoming data may arrive discontinuously and split into chunks of any size (most often 1460 or 2048 bytes, but potentially as small as 1 byte each).

- Some older daemon versions contained a bug that implicitly required `addr` to be in the **data** or **transfer** segments. While this has been fixed, if the buffer is small, you may consider putting it in these segments for maximum compatibility.

Return value: On success, returns 0 and loads information into `obj`, if specified. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Receive (TCPRCV)`.

TCP_ReceiveToEnd()

```
signed char TCP_ReceiveToEnd(unsigned char handle, unsigned char bank,  
                             char* addr, unsigned short maxlen);
```

Listens on the TCP socket `handle`, saving all received data to the buffer `addr` until the remote host disconnects. No more than `maxlen` bytes will be saved, and the socket `handle` will be freed with `TCP_Close()` before returning. If `maxlen = NET_FILE`, `addr` will instead be treated as the absolute path of a file to save the response to (with no length limit). (This function is intended for cases where we know the remote host will disconnect when finished sending data.)

When running on a [separate thread](#), the global **unsigned long** variable `_tcp_progress` contains the number of bytes received so far. Writing a nonzero value to the global variable `_tcp_abort` will cause `TCP_ReceiveToEnd()` to abort the transfer at its earliest convenience, writing only what it has received so far and failing with `_neterr = ERR_TRUNCATED`.

Return value: On success, returns 0. On failure, disconnects immediately, sets `_neterr`, and returns -1.

TCP_Skip()

```
signed char TCP_Skip(unsigned char handle, unsigned short len);
```

Skips and throws away `len` bytes of data which have already been received from the host associated with the socket `handle`. `len` must be equal to or smaller than the total number of bytes in the buffer (see `TCP_Status()`).

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Skip (TCPSKP)`.

TCP_Flush()

```
signed char TCP_Flush(unsigned char handle);
```

Flushes the send buffer immediately. Some network hardware or software may hold data in the send buffer for a brief period of time before sending; this command requests to send it immediately.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Flush (TCPFLS)`.

TCP_Disconnect()

```
signed char TCP_Disconnect(unsigned char handle);
```

Sends a disconnect signal to the remote host associated with the socket `handle`, closes the TCP connection, and releases the socket. (This is intended for we want to initiate the disconnection with the remote host; see also `TCP_Close()`.)

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Disconnect (TCPDIS)`.

TCP_Close()

```
signed char TCP_Close(unsigned char handle);
```

Closes and releases the TCP connection associated with the socket `handle`, without first sending a disconnect signal. (This is intended for when the remote host has already closed the connection with us; see also `TCP_Disconnect()`.)

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `TCP_Close (TCPCLO)`.

UDP functions

UDP is a simple “connectionless” protocol that allows raw packets to be sent and received from arbitrary IP addresses. Unlike TCP, no connection is negotiated, and the network hardware does not verify that sent packets are actually delivered (or delivered in the right order). This makes UDP most suitable for use-cases like a multiplayer game, where we may want to receive status updates from an arbitrary number of other players without it mattering very much if an individual packet gets lost.

We can open a UDP session with `UDP_Open()`. When a UDP packet arrives on an open session, the network daemon will send message ID `NET_UDPEVT`, which we can process using `UDP_Event()`.

Warning: UDP functions are not available on all devices. In particular, the M4 Board (the most common network hardware for Amstrad CPC) does not provide UDP; `UDP_Open()` will always fail with an error when used on this device.

UDP_Open()

```
signed char UDP_Open(unsigned char type, unsigned short lport, unsigned char bank);
```

Opens a UDP session on local port `lport`. Data for this session will be stored in RAM bank `bank`

Return value: On success, returns a socket handle to the new session. On failure, sets `_neterr` and returns -1.

SymbOS name: UDP_Open (UDPOPN).

UDP_Event()

```
void UDP_Event(char* msg, NetStat* obj);
```

A utility function for parsing a NET_UDPEVT message into a readable status report. `msg` is the address of the 14-byte message buffer containing the message; `obj` is the address of a `NetStat` struct to store the results, which has the same format as for `TCP_Status()`:

```
typedef struct {
    unsigned char socket;    // socket this message pertains to
    unsigned char status;    // status
    unsigned char ip[4];     // remote IP address
    unsigned short rport;    // remote port
    unsigned char datarec;   // n/a
    unsigned short bytesrec; // received bytes waiting in buffer
} NetStat;
```

UDP_Status()

```
signed char UDP_Status(unsigned char handle, NetStat* obj);
```

Returns the status of the UDP connection associated with the socket `handle` and stores the results in the `NetStat` struct `obj` (see `UDP_Event()`). This is equivalent to `UDP_Event()` but can be used to query the status of the connection at any time, not just after receiving a NET_UDPEVT message.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: UDP_Status (UDPSTA).

UDP_Send()

```
signed char UDP_Send(unsigned char handle, char* addr, unsigned short len,
                    char* ip, unsigned short rport)
```

Sends a data packet from the memory address `addr` (in the bank specified in `UDP_Open()` for this session) to the IP address stored in the 4-byte buffer `ip`, using the UDP session with the socket `handle`. If sending fails because the buffer is full, the application should idle briefly and try again.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: UDP_Send (UDPSND).

UDP_Receive()

```
signed char UDP_Receive(unsigned char handle, char* addr);
```

Moves data which has been received on the UDP session with the socket `handle` to the memory at address `addr` (in the bank specified in `UDP_Open()` for this session). The entire packet will be transferred

at once, so be sure that there is enough space for an entire packet at the destination address. (UDP packets have a theoretical limit of 65507 bytes, but we can also check how much data is waiting with `UDP_Event()` or `UDP_Status()`.)

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `UDP_Receive (UDPRCV)`.

UDP_Skip()

```
signed char UDP_Skip(unsigned char handle);
```

Skips and throws away a complete packet which has already been received on the UDP session with the socket handle.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `UDP_Skip (UDPSKP)`.

UDP_Close()

```
signed char UDP_Close(unsigned char handle);
```

Closes and releases the UDP session associated with the socket handle.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

SymbOS name: `UDP_Close (UDPCLO)`.

FTP functions

FTP (File Transfer Protocol) is an older protocol that now runs on TCP, using simple command-response pairs over an open TCP “control” connection and opening a second “data” connection when necessary to transfer data. The network library provides a set of functions for basic FTP commands.

Because FTP does not usually send data unsolicited, it is not generally necessary to listen for `NET_TCPEVT` events as we would for a typical TCP connection. A typical workflow is just:

- Connect to an FTP server with `FTP_Open()`.
- Upload, download, or browse files with `FTP_Upload()`, `FTP_Download()`, or `FTP_Listing()`.
- Disconnect when done with `FTP_Disconnect()`.

For diagnosing connection problems, in addition to the usual behavior of `_neterr`, the last FTP response code (e.g., 421 = service not available) is stored in the global variable `_ftp_response`.

FTP_Open()

```
signed char FTP_Open(char* ip, int rport, char* username, char* password);
```

Opens a client FTP control connection to the IPv4 address `ip` (stored in a 4-byte buffer, one byte per octet), connecting to port `rport` (usually 21). If the server requests a username and password, the provided strings `username` and `password` will be used.

Return value: On success, returns a TCP socket handle to the new connection. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1. (A common example: invalid login credentials will result in `_neterr = ERR_CONNECT`, `_ftp_response = 430`.)

FTP_Upload()

```
signed char FTP_Upload(unsigned char handle, char* filename, unsigned char bank, char* addr, unsigned
```

Executes an FTP upload over the open control connection with the socket number `handle`. `filename` is the name of the *destination* file on the FTP server (not the local filename!) Data can be uploaded from two sources:

- When `maxlen = NET_FILE`, the entire file with the absolute path at bank `bank`, address `addr` will be uploaded.
- When `maxlen` is a positive number, `maxlen` bytes of memory will be uploaded from bank `bank`, address `addr`.

`mode` is the upload mode, one of `FTP_ASCII` or `FTP_BINARY`. (`FTP_BINARY` is usually the safest option unless we know the file is ASCII-only.)

Examples:

```
char buffer[1024]; // some data in memory to upload
FTP_Upload(socket, "remote1.txt", _symlink, buffer, sizeof(buffer), FTP_BINARY);
FTP_Upload(socket, "remote2.txt", _symlink, "A:\\LOCAL.TXT", NET_FILE, FTP_ASCII);
```

When running on a [separate thread](#), the number of bytes sent so far is stored in the global **unsigned long** variable `_tcp_progress` (for uploads from a memory buffer) or the global **unsigned long** variable `_ftp_progress` (for uploads from a file). Writing a nonzero value to the global variable `_tcp_abort` will cause `FTP_Upload()` to abort the transfer at its earliest convenience, writing an incomplete file and failing with `_neterr = ERR_TRUNCATED`.

Return value: On success, returns 0. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1.

FTP_Download()

```
signed char FTP_Download(unsigned char handle, char* filename, unsigned char bank, char* addr, unsig
```

Executes an FTP download over the open control connection with the socket number `handle`. `filename` is the name of the *source* file on the FTP server (not the local filename!) Data can be downloaded to two destinations:

- When `maxlen = NET_FILE`, data will be downloaded to a file; bank `bank`, address `addr` contains the absolute file path.

- When `maxlen` is a positive number, data will be downloaded to memory bank `bank`, address `addr`. A maximum of `maxlen` bytes will be downloaded.

`mode` is the download mode, one of `FTP_ASCII` or `FTP_BINARY`. (`FTP_BINARY` is usually the safest option unless we know the file is ASCII-only.)

Examples:

```
char buffer[1024]; // buffer to store downloaded data
FTP_Download(socket, "remote1.txt", _symlink, buffer, sizeof(buffer), FTP_BINARY);
FTP_Download(socket, "remote2.txt", _symlink, "A:\\LOCAL.TXT", NET_FILE, FTP_ASCII);
```

When running on a [separate thread](#), the global **unsigned long** variable `_tcp_progress` contains the number of bytes received so far. Writing a nonzero value to the global variable `_tcp_abort` will cause `FTP_Download()` to abort the transfer at its earliest convenience, writing only what it has received so far and failing with `_neterr = ERR_TRUNCATED`.

Return value: On success, returns 0. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1.

FTP_Listing()

```
signed char FTP_Listing(unsigned char handle, unsigned char bank, char* addr, unsigned short maxlen)
```

Equivalent to `FTP_Download()`, but downloads a listing of files in the server's current directory instead of a specific file. The format of the directory listing is not standardized, but in practice is almost always either Unix-style `ls` output or DOS-style `dir` output, one file per line.

Return value: On success, returns 0. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1.

FTP_ChDir()

```
signed char FTP_ChDir(unsigned char handle, char* path);
```

Changes the current working directory of the open FTP control connection with the socket number `handle` to `path`. Paths may be relative or absolute, behaving similarly to a terminal `cd` command:

- `FTP_ChDir(socket, "subdir");` - enter subdirectory `subdir`
- `FTP_ChDir(socket, "..");` - go up one directory
- `FTP_ChDir(socket, "/dir/subdir");` - go to the absolute path `/dir/subdir` (note that absolute paths are separated by slashes `/`, *not* backslashes `\`!)

Most FTP servers give users access to a specific subfolder of their native filesystem, treating the root of this folder as `/`. However, some servers use `/` to mean the root of their entire native filesystem (which the user may only be permitted to access part of). To determine the current working path, we can use the FTP `PWD` command:

```
char buffer[256];
FTP_Command(socket, "PWD\r\n", buffer, sizeof(buffer));
```



```
// this will leave buffer[] loaded with something like:  
// 257 "/current/path" is the current directory.
```

Return value: On success, returns 0. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1.

FTP_Command()

```
int FTP_Command(unsigned char handle, char* cmd, char* addr, unsigned short maxlen);
```

Executes a complete FTP command/response pair on the open control connection with the socket number `handle`. The FTP command `cmd` should be a single-line ASCII string ending in `\r\n` (e.g., "LIST\r\n"). The response text will be written to the buffer `addr`, up to a maximum of `maxlen` bytes.

Return value: On success, returns the three-digit FTP response code (e.g., 550 = file not found). On failure, sets `_neterr` and returns -1.

FTP_Response()

```
int FTP_Response(unsigned char handle, char* addr, unsigned short maxlen);
```

Waits for an FTP response on the open control connection with the socket number `handle`, without first sending a command. The response text will be written to the buffer `addr`, up to a maximum of `maxlen` bytes. (This is intended for catching responses that are not directly related to a command, e.g., 226 Transfer complete..)

Return value: On success, returns the three-digit FTP response code (e.g., 550 = file not found). On failure, sets `_neterr` and returns -1.

FTP_GetPassive()

```
signed char FTP_GetPassive(unsigned char handle, char* ip, unsigned short* port);
```

Executes an FTP PASV command on the open control connection with the socket number `handle`, saving the IPv4 address returned by the server to the 4-byte buffer `ip` and the port number to the variable `port` (passed by reference).

Explanation: FTP is unusual in that, for historical reasons, data transfer occurs over a second “data” TCP connection opened parallel to the main “control” connection. Historically, this was done by having the server connect back to a port on the client computer (“active” mode). However, modern firewalls typically block this type of unsolicited connection, so most data connections are now made in “passive” mode: the client sends a PASV command to the server, which then responds with an IPv4 address and port number for the client to open the data connection from its side. For simple uploads/downloads, the functions `FTP_Upload()` and `FTP_Download()` handle all of this automatically, but this function exists to facilitate more complicated tasks that require a passive connection.

Return value: On success, sets `ip` and `port` to the values indicated by the server and returns 0. On failure, sets `_neterr` (and potentially `_ftp_response`) and returns -1.

FTP_Disconnect()

```
signed char FTP_Disconnect(unsigned char handle);
```

Sends a QUIT command to the FTP server associated with the socket `handle`, closes the TCP connection, and releases the socket.

Return value: On success, returns 0. On failure, sets `_neterr` and returns -1.

FTP_Close()

(alias for `TCP_Close()`, since the FTP control connection is just a normal TCP connection.)

Helper functions

Net_ErrMsg()

```
void Net_ErrMsg(void* modalWin);
```

Displays a message box with the current error in `_neterr`, if any. `modalWin` specifies the address of a window data record that should be declared modal, if any; this window will not be able to be focused until the message box is closed. If `modalWin = 0`, no window will be declared modal.

Net_SplitURL()

```
signed char Net_SplitURL(char* url, char* host, char** path, unsigned short* port);
```

A utility function that splits the string `url` (containing a URL) into its constituent components, writing the hostname to the buffer `host`, the address of the path/query string to the variable passed by reference as `path`, and the port number to the variable passed by reference as `port`. The buffer `host` should be at least 65 bytes long.

For example, the URL `http://example.com:8080/path?id=1` would be split into:

- `host`: `example.com`
- `path`: address of `path?id=1` in `url`
- `port`: `8080`

If not explicitly specified in the URL, `Net_SplitURL()` will attempt to determine a default port number from the detected protocol (e.g., port 80 for HTTP).

Return value: On success, returns the identified protocol (one of `PROTO_OTHER`, `PROTO_HTTPS`, `PROTO_HTTP`, `PROTO_FTP`, `PROTO_IRC`, `PROTO_SFTP`, `PROTO_FILE`, `PROTO_IMAP`, `PROTO_POP`, or `PROTO_NNTP`). On failure, sets `_neterr = ERR_BADDOMAIN` and returns -1.

Net_PublicIP()

```
signed char Net_PublicIP(char* ip);
```

A utility function that attempts to determine the computer's public-facing IP address (by querying the free AWS service `checkip.amazonaws.com`). This is useful for server programs that need to tell the user what IP address remote clients should try to connect to.

Note: This function is larger than one might expect because it pulls in the full HTTP-handling portion of the library.

Return value: On success, stores the IP address in the 4-byte buffer pointed to by `ip` and returns 0. On failure, sets `_neterr` and returns -1.

Net_SkipMsg()

```
void Net_SkipMsg(signed char handle);
```

A utility function that removes all remaining `NET_TCPEVT` or `NET_UDPEVT` messages pertaining to the socket `handle` from the message queue. This is mainly used internally, but can be useful to keep the ring buffer from overflowing in apps that only selectively ingest certain event messages, putting the rest back on the queue. (If you don't know what that means, you probably don't need to worry about this.)

iptoa()

```
char* iptoa(char* ip, char* dest);
```

Converts the numeric IPv4 address stored in the 4-byte buffer `ip` into a readable string representation in the buffer `dest` (e.g., `192.168.0.1`). `dest` must be at least 16 bytes long.

Return value: `dest`.

Reference

Error codes

The following errors are primarily issued by the network interface (stored in `_neterr`):

- `ERR_OFFLINE`: Offline/not connected/no network daemon
- `ERR_NOHW`: No hardware setup
- `ERR_NOIP`: No IP configuration
- `ERR_NOFUNC`: Function not supported
- `ERR_HARDWARE`: Unknown hardware error
- `ERR_WIFI`: WiFi error (SymbiFace 3)
- `ERR_NOSOCKET`: No more free sockets
- `ERR_BADSOCKET`: Socket does not exist
- `ERR_SOCKETTYPE`: Wrong socket type
- `ERR_SOCKETUSED`: Socket is already in use by another process
- `ERR_BADDOMAIN`: Invalid domain string
- `ERR_TIMEOUT`: Connection timeout
- `ERR_RECURSION`: Recursion not supported

- `ERR_TRUNCATED`: Truncated response
- `ERR_TOOLARGE`: Packet too large
- `ERR_CONNECT`: TCP connection not established (or not yet established)
- `ERR_NETFILE`: File error while performing network operation; check `_fileerr`
- `ERR_RESPONSE`: Unexpected response

Special considerations

Contents

- [The `malloc\(\)` heap](#)
- [Native preprocessor](#)
- [Quirks of `stdio`](#)
- [Using `as` as a standalone assembler](#)
- [Building SCC](#)
- [SCC versus SDCC](#)

The `malloc()` heap

The `libc` function `malloc()` dynamically allocates a new block of memory for storing data. On a modern system, this is the usual way of requesting additional memory, and a lot of existing C code assumes that it can `malloc()` more memory indefinitely—often without even checking for out-of-memory errors. This is a reasonable assumption on modern platforms, but definitely not true on the Z80 (which has a limited 64KB address space), and even less true on SymbOS (where multiple multitasking applications may be packed into the same 64KB bank). In practice, a SymbOS executable must either 1) use [banked memory](#) system calls to access memory in other banks, or 2) declare upfront how much memory it needs in its main 64KB bank.

SCC currently implements an imperfect compromise, allocating a static 4KB heap for `malloc()` to expand into. This is sufficient for everyday usage (such as allocating `FILE*` records and other small data structures), but not for allocating large data buffers. If larger buffers are needed, we have three options:

- Rewrite the code to use static buffers.
- Rewrite the code to use [banked memory](#) system calls.
- Increase the static heap size with the `cc` command-line option `-h`:

```
cc -h 16384 source.c
```

Native preprocessor

For desktop cross-compilation, SCC uses MCPP as its preprocessor, which should support all standard C preprocessor syntax. For improved speed when running natively on SymbOS, SCC uses a stripped-down preprocessor. The native processor supports the most common directives used in 99% of cases (`#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#else`, `#endif`), but not `#if (condition)` (which is much more complicated to implement, and only rarely used). Use combinations of `#ifdef` or `#ifndef` instead of `#if (condition)`. Function-style definitions (e.g., `#define add(x,y) ((x)+(y))`) are supported, but only with single-character argument names.

A more advanced (but slower) native preprocessor is available in the `bin/symbos/alternate` folder of the SCC source repository. This preprocessor is adapted from the Fuzix Compiler Kit, and can be substituted directly for the default native preprocessor if desired.

Quirks of `stdio`

File sizes

Due to a limitation of the filesystem, files stored on AMSDOS filesystems (e.g., CPC floppy disks) will often be terminated with an EOF character 0x1A and then some garbage padding (see [File Access](#)). To improve compatibility, most `stdio.h` functions treat character 0x1A as EOF. If we need to read a binary file that includes legitimate 0x1A characters, the file should be opened in binary (b) mode, e.g.:

```
f = fopen("data.dat", "rb");
```

...with the tradeoff being that we now need to pay attention to the fact that there may be garbage data at the end of the file. (This problem does not apply to the FAT filesystems used by most mass storage devices.)

`fseek()`

`fseek()` and `lseek()` past the existing end of a file will generally follow the POSIX behavior of filling in the intervening space with zeros (to create a “sparse file”). However, for multiple internal reasons, this will occur at the time of the seek, rather than if/when the file is written to.

`printf()`

Because of how SCC handles variable argument lists, SCC’s implementation of `printf()` and its relatives (`vsprintf()`, etc.) are pickier than some others about the data types of passed arguments matching the data types indicated in the format string. In particular, 32-bit values should be cast to `(int)` before being printed with `%i`, and 8-bit and 16-bit values should be cast to `(long)` before being printed with `%l`. (Since 8-bit values are passed internally as 16-bit values on the stack, it is not necessary to cast 16-bit values to 8-bit or vice versa.)

Using `as` as a standalone assembler

`as` can be used in conjunction with `ld` as a standalone assembler (including natively on SymbOS). First, use `as` to assemble an assembly (`.s`) file to an object (`.o`) file:

```
as asmfile.s
```

Then, use `ld` to link the `.o` file into a binary. To output a raw binary file without `ld` trying to patch the start of the file (see below), we should run `ld` with the `-b` option:

```
ld asmfile.o -b -o asmfile.out
```

The `-o` option allows us to specify the output file name, in this case `asmfile.out`. Multiple `.o` files can be linked into a single binary by listing them in the command line (e.g., `ld asmfile1.o asmfile2.o...`), so we can split up large projects into multiple `.s` files, assemble them separately into object files, and link all the object files together. Symbols can be shared between source files using the `.export` directive, as described above.

The behavior without the `-b` option is slightly different. `ld` assumes that we want to arrange any defined segments (`.code`, `.symdata`, `.symtrans`) into a SymbOS executable and that the code at the start

of the `.code` segment defines a valid SymbOS executable header. After linking, the appropriate bytes in the header will be updated with the actual segment lengths. If our assembly files do in fact define a valid SymbOS executable header, this allows us to conveniently define segments using the `.symdata`, etc. directives rather than manually tracking their locations with symbols.

Important quirk when doing this natively on SymbOS: For complicated internal reasons, the native SymbOS executables `as.exe` and `ld.exe` are shipped as `.exe` files, not `.com` files. If we run these executables directly from SymShell, SymbOS will interpret them as windowed applications and will not show their output in SymShell. In order to see their output in SymShell, we must rename them to `as.com` and `ld.com` (or create new copies with these names).

Building SCC

The current primary build target for SCC is Windows. Install MinGW and Python 3, ensure that their `bin` folders are in the system path, and then run the `make.bat` batch files found throughout the SCC source tree to compile the relevant parts of the codebase. (This really ought to transition to proper Makefiles, but whatever.)

SCC versus SDCC

SCC is being developed as a replacement for Nerlaska's SymbosMake SDK for SDCC. The original version of this SDK is now hard to find, but a patched version for SDCC 4.1.12+ can be found in the [CPvM source repository](#).

While an incomplete early effort (SymbosMake reports the version number 0.0.1), this SDK has served admirably, being used to develop major applications including CPvM, Zym, and Star Chart. The key advantage of SymbosMake is more efficient code generation. Since it is based on the SDCC cross-compiler, it is able to take advantage of SDCC's modern memory- and CPU-hungry optimization techniques and more efficient calling convention. SDCC also supports several helpful C features missing in SCC, most notably inline assembly and named struct initializers. Thus, for performance-critical applications, SymbosMake remains a viable tool.

However, SCC aims to improve on SymbosMake in several important ways:

- It's open-source, so bugs can actually be fixed
- A full build chain with object files and a linker, rather than single-file compilation
- A native stdio port, so `printf()` etc. can be used in SymShell
- More comprehensive headers for system calls
- More flexible segment management; it is much easier to place globals in the correct segment, large buffers are not forced into the **data** segment unnecessarily, and static initializers do not waste space by being forcibly duplicated in the **transfer** segment
- Static initializers can contain direct pointers to other data structures, rather than having to initialize this at runtime (very useful when defining windows and controls)
- Much faster compilation times
- Much more comprehensive documentation
- Faster floating-point library

Porting tips

Contents

- [Common problems](#)

Common problems

“Unknown symbol” errors on compilation

This occurs when the code references a function that is not actually defined. (Because this error originates at the linking stage, an underscore will be added to the beginning of the listed name—i.e., `_funcname` for `funcname()`—and no line number will be given in the error.) If the function is a standard library function that should exist, there are two possibilities:

1. SCC may not (yet) provide the function. This is especially likely for platform-specific functions like POSIX system calls (`fork()`, `chmod()`, etc.)
2. The function may be in an external library that must be manually linked with `-l` command-line options. Notably, following C convention, most `math.h` functions (`sqrt()`, etc.) are in the external library `libm.a`. To use these functions, `cc` must be run with the option `-lm`.

“Out of memory” error when starting the compiled app

This error is often confusing because it can occur even when there is apparently still memory available. It is important to remember that SymbOS does not work with a single large address space like modern systems, but must arrange memory in a more constrained way (see the [section on memory segments](#)). So, what this error really means is that SymbOS cannot allocate *the exact arrangement of memory segments requested by the app*, not necessarily that it is out of memory entirely.

The two common causes of this error are:

1. The app does not obey SymbOS’s [segment size restrictions](#). (Usually the compiler will warn about this.) The code might simply be too large to fit in 64KB, but it’s also common for code written for modern systems to declare buffers that are much larger than actually needed, because memory is considered cheap. Look through the code to see if anything can be slimmed down. (Tip: To see how much space different parts of the code are taking up, run `cc` with the option `-M mapfile.txt`. This will create a “map file” called `mapfile.txt` showing the relative addresses of different symbols in the executable. By running this file through the included command-line utility `sortmap.exe`, it is possible to see how much space each object occupies.)
2. The executable file was incompletely written or corrupted somehow. Because SymbOS reads memory-allocation information from the executable’s header, if this header is corrupted, it will often trigger a misleading “out of memory” error instead of something more specific. Verify that all steps of the compilation are finishing correctly and without errors, and that the file is not being modified in the process of transferring it to the computer or emulator where it is being run.

(For out-of-memory errors raised by the app itself during normal operation, see [the section on `malloc\(\)` under Special Considerations](#).)

Problems with `malloc()`

See [the section on `malloc\(\)` under Special Considerations](#). A lot of modern code implicitly assumes that it can `malloc()` as much memory as it needs; indeed, many smaller (or lazier) programs won't even bother to check for out-of-memory on `malloc()`, since who ever heard of getting an out-of-memory error for allocating a measely few megabytes? SymbOS, that's who!

`printf()` formatting looks wrong

Note that, for reasons of space and performance, SCC's `printf()` implementation does not properly handle every advanced feature available on other platforms. More complex formatting may need to be simplified or rewritten. (True ANSI C `printf()` is a notoriously huge function, often requiring 20KB or more of space to implement!)

Can't `fopen()` a file that should be openable

A few possibilities:

1. The file path may be wrong in a nonobvious way. Note that, to save memory, SymbOS does not keep track of an app's "current path" in the same way most modern systems do, so SCC's `libc` implementation generally assumes that the "current path" is the location of the executable (for GUI apps) or the current SymShell path (for console apps). If this is not what the app itself expects, problems may arise.
2. The file name may be wrong in a nonobvious way. SymbOS only supports files with MS-DOS 8.3-style names, whereas a lot of existing C code assumes files can have arbitrarily long names; these may need to be modified to fit the 8.3-style format.
3. Too many files may be open at once. SymbOS has a hard limit of 8 concurrent filehandles, shared across all open applications.
4. If the app uses `malloc()`, it may be running out of heap space to store the file handle (see [the section on `malloc\(\)` under Special Considerations](#)).

`fread()` etc. won't read the end of a file (or reads extra junk at the end of a file)

This usually relates to a known limitation of AMSDOS and CP/M-format filesystems. See [the section on file sizes under Special Considerations](#) for details and workarounds.