## GETTING STARTED

Welcome to the world of C programming for the Spectrum, Timex/Sinclair and their clones. My C compiler of choice is Z88DK, a cross compiler that implements a nearly ANSI-C subset of the language. There are a couple of other non-commercial C compilers available for Z80 machines (SDCC, Hi-Tech C) but Z88DK has a number of advantages:

- It boasts significant C library support including a comprehensive independent floating point library
- It has support for more than 20 different Z80 machines
- Code generated is reliable
- It implements a nearly ANSI-C subset of the language
- It is actively being developed by a responsive team

SDCC is also being actively developed but its original target was the 8051. The compiler's support for the Z80 is still preliminary but it may be a top-choice in the future if its development goals see fruition.

Hi-Tech C was a commercial C compiler targetted at CP/M that has been released as freeware by Hi-Tech. It boasts a global code optimizer but as its intended target is CP/M, it requires a significant effort invested to provide library support for other Z80 machines not equipped with CP/M.

Z88DK is a cross compiler, meaning you develop software on a separate development machine and not necessarily the target machine. The Z88DK zip file comes with complete C source so you can compile it for whatever target machine you would like to develop on. There are also binaries available for 16-bit DOS and 32-bit Win32 platforms (thanks Dennis) as well as the Amiga so that you may not have to do this compilation yourself.

I mentioned that Z88DK is a nearly ANSI-C compiler. I am aware of three shortcomings:

- Pointers to structs cannot be typedef'd. If porting existing code, try using macros but be careful as a macro can't always substitute for a typedef!
- Function pointers cannot be prototyped. This makes a declaration like "void *(*my_malloc)(uint bytes);" illegal but this is easily circumvented by declaring all functions pointers as "void*" type. You will see that I frequently use function pointers in the examples in the tutorial section.
- Multi-dimensional arrays are not supported.

## INSTALLING Z88DK

Step one is to download and install Z88DK. Visit http://z88dk.sourceforge.net/, scroll down to "download" and click on the "downloads" link. This will take you to sourceforge's Z88DK downloads page. Grab the latest version (now 1.5) for your platform. Once you've downloaded that unzip the whole thing into a sensible directory (like "C:\z88dk" for a WinTel machine).

The next step is to compile Z88DK. The best place to get help with this is to ask on the Z88DK mailing list. However if you are using a Windows machine, Dennis Gronning will save you some grief. He has made available precompiled binaries for Win32 and 16-bit DOS on his webpage. You will need the frontend (ZCC), the preprocessor (CPP), the compiler (SCCZ80), the optimizer (COPT), the assembler/linker (Z80ASM) and APPMAKE. Download each of these and unzip them into the z88dk binary directory ("C:\z88dk\bin" if you've been following my suggestions).

Next, if you have a Windows or DOS machine, download init.bat and place this in "C:\z88dk". This is a batch file that will set up some environment variables for z88dk when you are ready to compile your C programs. Non-windows users will want to set up the same environment variables in their ".cshrc" file or equivalent.

Finally, update some z88dk files to correct a few bugs that have been found since release 1.5. Unzip the new stdlib.h file into "C:\z88dk\include" and the three other files in z88dk1.5fix.zip into "C:\z88dk\lib". The new z80rules files fix a few code optimizer bugs and the new spec-crt file adds code to allow setting the initial location of the Z80 stack using a pragma directive (see the examples in the tutorial).

I would suggest creating a work directory in "C:\z88dk\work" where you can save your C programs for compilation.

Congratulations, your are finished with the z88dk installation! You may find it worthwhile to peruse the "C:\z88dk\include" directory to see exactly what is available from z88dk's built-in libraries.

If you are new to C, there are plenty of introductions scattered around the web. The definitive C reference is "The C Programming Language," by Kernighan and Ritchie. Make sure it's at least the second edition as earlier editions cover C before it was standardized by an ANSI committee. A good followup is "Expert C Programming Deep C Secrets," by Peter Van Der Linden. These books are not introductions to programming - they are presentations of the C language itself. If you want to become an expert you should also read a good book on data structures.

## INSTALLING THE SPRITE PACK LIBRARY

Grab the latest version of sprite pack and unzip it into a suitable location (perhaps "C:\z88dk\work\splib2"). The complete source code of the sprite pack library is made available along with a DOS makefile batch program to compile the library. Apologies to non-DOS users; I will get a real makefile utility in the future. In the meantime you will have to make the necessary modifications to the batch file(s) in order to be able to compile the library yourself.

Sprite Pack is customizable by editing the "SPconfig.def" file prior to compiling the library. It supports four video modes: the Spectrum's standard display, the Timex hi-colour display (256x192 pixel, 32x192 colour), the Timex hi-res display (512x192 pixel) and the Timex double buffered display. You can also alter things like the row range of the display that will be used and the locations of various sprite pack tables. The config file is fairly straightforward to understand but if anything is a little unclear, please read the programmer's tutorial below to understand the options better.

To compile the library on a WinTel machine, edit the "SPconfig.def" file to your liking, open up a DOS box in Windows, change to the "C:\z88dk" directory, run "init.bat" to set up environment variables, change to "C:\z88dk\work\splib2" and run "makefile.bat". Following compilation you will have a new "splib2.lib" library file.

Copy this "splib2.lib" file to "C:\z88dk\lib\clibs" so that the z88dk linker will be able to find it. Also copy the header file "spritepack.h" to "C:\z88dk\include".

I also included a number of precompiled libraries for your convenience. "splib2a.lib" targets the Spectrum video display, "splib2b.lib" targets the Timex double-buffer display mode, "splib2c.lib" targets the Timex hi-colour display and "splib2d.lib" targets the Timex hi-res display. You can avoid compiling the library yourself by simply renaming one of these files to "splib2.lib" and then copying it to the "C:\z88dk\lib\clibs" directory. The settings used to compile these library files are in this "SPconfig.def" file.

## USING Z88DK

This discussion is Wintel-specific. Extrapolate to your own platform.

Z88DK is a breeze to use. Edit your C programs using your favourite text editor and save your work into "C:\z88dk\work". When you are ready to compile, open up a DOS box in Windows, change to the "C:\z88dk" directory and run "init.bat" to set up z88dk's environment variables. (NOTE: if you get an insufficient environment space error you can increase the amount of environment space available by right-clicking on the DOS window's title bar, click properties then the memory tab and modifying the initial environment space box. Or you can do what I do: "set path=c:\windows;c:\windows\command" and then re-run "init.bat"). Then change to your work directory "C:\z88dk\work" and compile your program:

```
zcc +zx -vn myprog.c -o myprog.bin -lm -lsplib2 -lmalloc
-lndos

+zx       Tells z88dk that we want to target the Spectrum
-vn       Keeps z88dk in quiet mode so we don't see a whole
list of messages
-o        The following name should be used as the output file
-lm       Link to the floating point math library (optional)
-lsplib2  Link to the Sprite Pack library (optional)
-lmalloc  Link to the C malloc functions (optional)
-lndos    Link to some i/o stubs (necessary)
```

The optional link parameters are only necessary if your program uses code from those libraries. The output of a successful compilation is the file "myprog.bin" which is a machine code program that starts at address 32768. Z88DK generates code that will start at 32768 by default; you can change this with the appropriate compile option.

Most emulators will allow you to load this binary directly into memory at 32768. If you do this, you can start the program with a "RANDOMIZE USR 32768" command.

Z88DK also comes with a utility that will convert this binary into a more standard tap file. Here's how:

```
bin2tap myprog.bin myprog.tap [32768]
```

The tap file "myprog.tap" will be generated from the supplied binary. The optional numerical parameter specifies the start address of the binary and is 32768 by default.

The complete "zcc" compile command compiles your C program in several steps (get rid of the '-vn' parameter to see them). Among these steps is a translation of your C code into assembler and then a compilation of your assembler program with any necessary linking with library subroutines. If you've made a C error you will hear about it many times.

As an alternative, you can initially compile your program with:

```
zcc -a myprog.c
```

This will only translate your C code into assembler. You will only hear about C errors and you will only be told about them once. This can be a great relief :-).

## ADVANCED Z88DK

I don't have the time at the moment but in the future I'd like to fill in some details about the other tools that come with z88dk, how to understand the assembly code generated by z88dk and the best ways to write C code.

I'd also like to discuss some interfacing tricks between assembler and C so that you can easily mix assembly into your C programs. This will include how to use the Sprite Pack library directly from assembler. A small example of this can be found in the pacmen demo.

How to make your own z88dk libraries so that you can share your code with the rest of the T/S and Z80 community!

## SPRITE PACK LIBRARY PROGRAMMER'S TUTORIAL

Sprite Pack is an evolving collection of machine code subroutines for the Spectrum, TS2068, TC2048 and their clones. The collection is made available as a Z88DK library, meaning you can call the subroutines from machine code, C or from a mixture of the two. One of the primary reasons I made it available from C is so that novice programmers would be able to write good programs without in-depth knowledge of assembly language or the Spectrum's hardware. The Spectrum is a bit of a paradox in that it was designed to be easily programmed but because the hardware is so simple you have to do all the work. This makes it much more of a challenge to write *good* software for it than for many of its contemporaries which had hardware support for sprites, music synthesis, etc. C is a small language and, in my opinion, is easier to write programs with than BASIC. The results are certainly better in terms of speed and what can be accomplished. I hope you will agree with me after following this tutorial.

For advanced users, the library offers the option of quickly getting programs up and running. Although compiled C is not as fast as assembly language, since the SP library (and much of the Z88DK library) is written in assembly language, C is used mainly to hold together calls to already optimized machine code subroutines. There is no way to write faster programs for the Sinclair machines than with Z88DK and C without resorting to assembly language. Since most programs spend 90% of the time executing 10% of the code, if performance still needs improvement, that 10% of C code can be rewritten seamlessly in assembler to improve performance further. This can save a lot of development time!

Sprite Pack was originally billed as a games development environment. The SP library has become a catch-all library for most of the assembly language subroutines I have written over the last dozen years or so. This means that it is evolvng into more than just a gaming tool. Future development will add support for sound and music, a GUI toolkit, a windowing system and a simple multitasking library.

And now a reality check. I realize that most readers will want to write games. SP's sprite engine cannot do it all -- the reality is that the Spectrum (and its derivatives and clones) are primarily limited by CPU speed. There simply isn't enough surplus CPU cycles available to write a do-it-all general purpose sprite and graphics engine. You will always get optimal results by writing a tailored graphics engine for each piece of software you write. However, I have tried to write a general purpose sprite engine that can do a lot. What it was not designed for: scrolling games where most of the screen is updated on every frame. What is was designed for: games and applications where only portions of the screen need updating on every frame (in this category most non-game applications will fall as will platformers, puzzlers, non-scrolling arcade type games). The display algorithm actually keeps track of what portions of the screen change between updates at a character-cell granularity and then only redraws those portions when the screen is updated. This is a lot of overhead if much of the screen needs redrawing, but can make programs faster if most of the screen is static. Another graphics engine designed specifically for smooth scrollers is planned.

Sprite Pack currently consists of nine major independent modules:

- SPRITES. Supports the creation, movement and deletion of any number of sprites on screen. Sprites can be placed with pixel precision thanks to a built-in software rotater and can be one of four types: mask, or, xor and load. Background tiles cover the screen background when sprites are used.
- COMPLETE IM2 INTERRUPT API: Interrupt service routines can be installed and uninstalled on any vector. A specially supplied Generic Interrupt Service Routine can be installed on any vector and can have any number of hooks attached to it. On interrupt each of these hooks is called in sequence.
- INPUT: Read input from the keyboard, joysticks and mice.
- SCREEN ADDRESS HELPERS: A set of functions that can compute screen addresses from pixel coordinates or character coordinates. Functions can also modify the screen address to move up, down, left or right by pixel or character amounts.
- RECTANGLE INTERSECTIONS: Subroutines that can detect collisions among rectangles, intervals and points. Perfect for collision detection of sprites or monitoring a "hot spot" on a graphical pointer.
- DYNAMIC BLOCK MEMORY ALLOCATOR: A fast, compact block memory allocator. Complements C's built-in malloc functions.
- ABSTRACT DATA TYPES: Submodules include linked list, hash table and heap data types.
- DATA COMPRESSION: Currently consists of a static Huffman compressor / decompressor.
- MISCELLANEOUS: Includes a random number generator, multiplications, endian swap, a pattern flood fill, etc.

This tutorial describes the C API and assumes the Spectrum video mode was selected prior to library compilation. It will be helpful to have a copy of "spritepack.h" open while reading this. This file contains all of SP's function prototypes -- search for the function of interest and you will see what parameters are expected when the function is called. You will also notice that all data structure names and function names that exist in the SP library are prefaced by 'sp_'. This should be especially helpful to new C programmers so that they can easily see what is and is not part of the C language itself.

## SPRITES & BACKGROUND TILES

The sprite module is centered around a differential screen updater that divides the display into the familiar 32x24 array of character cells. Each character cell contains exactly one background tile and can contain any number of sprite characters. The updater is 'differential' because it does not actually redraw the whole screen when an update occurs. It will only redraw those character cells that have changed since the last update. Information on which character cells have been modified is stored in an internal data structure. Marking specific character cells as needing a redraw in the next update is called 'invalidation' and marking them as not needing a redraw is called 'validation'. Most SP functions that alter the background tiles or sprites will automatically invalidate the character cells they affect so that the screen is redrawn properly in the next update. You can take more control over which areas on screen are redrawn or not through calls to "sp_Invalidate" and "sp_Validate".

The two graphical entities that the sprite module understands are the background tile and the sprite. "Background tile" is really a fancy name for a coloured UDG. They are printed to the screen in almost the same way letters and real UDGs are printed to the screen from BASIC. Here is an example that will print the letter 'A' in black on white colour at position (10,12) on the screen:

```
    sp_PrintAtInv(10,12,INK_BLACK | PAPER_WHITE,'A');
```

There are several variations of this function (see sp_PrintAt, sp_PrintAtInv, sp_PrintAtDiff) that only differ on whether they invalidate the character cell being printed to. This example uses "sp_PrintAtInv" which, besides printing the letter 'A', will also invalidate the cell so that the letter is drawn in the next screen update. REMEMBER: None of SP's commands immediately cause changes to the display! Display changes occur all at once when the screen is updated in a call to "sp_UpdateNow".

Time for an example. Since we haven't gotten very far yet you will be briefly introduced to a handful of SP functions that haven't been discussed. They will be more fully discussed later. A compiled tap file is also available.

```
#include <stdlib.h>
#include <spritepack.h>

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

#pragma output STACKPTR=61440

main()
{
   uint keyI, keyC;

   #asm
   di
   #endasm
   sp_InitIM2(0xf1f1);
   sp_CreateGenericISR(0xf1f1);
   #asm
   ei
   #endasm

   sp_Initialize(INK_BLACK | PAPER_WHITE, ' ');
   keyI = sp_LookupKey('i');
   keyC = sp_LookupKey('c');
   while(1) {
      sp_UpdateNow();
      sp_PrintAt(rand()%24, rand()%32, INK_YELLOW |
PAPER_BLUE, 'A');
      sp_PrintAtInv(rand()%24, rand()%32, INK_RED |
PAPER_CYAN, 'B');
      sp_Pause(20);
      if (sp_KeyPressed(keyC)) {
         sp_ClearRect(sp_ClipStruct, INK_BLACK | PAPER_WHITE,
' ', sp_CR_TILES);
         sp_Invalidate(sp_ClipStruct, sp_ClipStruct);
      } else if (sp_KeyPressed(keyI))
         sp_Invalidate(sp_ClipStruct, sp_ClipStruct);
   }
}
```

You may have noticed two non-standard items in this listing if you have some sort of background in C programming. The special "pragma" directive tells Z88DK that we want to start the program with the Z80's stack pointer (SP) set to 61440. Spectrum BASIC places the stack at an inconvenient location so this moves it just below some tables that are automatically initialized by SP when "sp_Initialize()" executes. Include this directive in all your SP programs just before C's main() function. SP's memory map is fully discussed in the DYNAMIC BLOCK MEMORY ALLOCATOR section.

The other odd item is the embedded assembler "di" and "ei" instructions that disable interrupts while the enclosed C code runs. For this program, BASIC's interrupt service routine needs to be disabled because one of the functions called uses the IY register (sp_Invalidate; see a list of all functions in SP using IY). The interrupt service routine provided by Spectrum BASIC expects the IY register never to be touched. I could have simply disabled interrupts with "#asm; di; #endasm" but then the "sp_Pause" function would lock up as it counts interrupts to measure time. The alternative I've used here is to create my own interrupt service routine (which does nothing) to replace Basic's. It is not necessary for you to understand this now as all of this will be explained fully in the INTERRUPTS section.

"sp_ClipStruct" is one of Sprite Pack's variables. Its odd-looking declaration has been copied out of SP's header file "spritepack.h". There are about a dozen of these variables defined in the header file that can be used in your program by declaring them in your main file in this way. "sp_ClipStruct" is a pointer to a rectangle that defines the outline of the full screen in characters. More information on why this declaration is the way it is can be found in the Sprite Pack Variables section.

"sp_Initialize" initializes the sprite module and must be called before any sprites or background tiles are used. Its parameters, in this case, cause the initializer to clear the screen to black on white space characters.

"sp_LookupKey" returns a scan code for an ascii character that represents the key presses necessary to generate the character. A later call to "sp_KeyPressed" accepts this scan code and will return true if the key is currently being pressed.

"sp_Pause" functions exactly like the Basic PAUSE command.

The "sp_Invalidate" call accepts two rectangles in character units. The intersection of these two rectangles defines the area of the screen that will be marked for update. Typically the second rectangle covers the full screen (ie, "sp_ClipStruct") and the first rectangle defines the area that should be invalidated, which could extend past the screen's boundaries. In this example, "sp_ClipStruct" is used for both rectangles causing the full screen to be invalidated.

"sp_ClearRect" does what its name suggests - it clears a rectangular area on screen. In this case, the full screen is cleared ("sp_ClipStruct") with black on white space characters. The flag "sp_CR_TILES" indicates that only background tiles are cleared. Other flag settings are "sp_CR_SPRITES" - to clear sprites only - and "sp_CR_AALL" to clear sprites and tiles. This function does *not* invalidate the area it clears meaning the area will not necessarily be redrawn in the next screen update. The following call to "sp_Invalidate()" makes sure the area is redrawn.

On each pass through the main loop, two letters are printed at random locations on the screen. 'A's are printed without invalidation and 'B's are printed with invalidation. Each time through the loop a call to "sp_UpdateNow" is made to redraw portions of the screen that have been invalidated. This causes all the 'B's to be seen, but no 'A's! The 'A's *are* being printed, but are not being invalidated so they do not appear when the screen is being updated. To verify this, hold down the 'i' key on the keyboard. This will force the entire screen to be invalidated (and therefore redrawn) and the 'A's will suddenly appear. Pressing 'c' will clear the screen.

Another function you may find useful is "sp_ScreenStr()" which is analagous to BASIC's SCREEN$ and ATTR commands. It returns the character code and colour of the *background tile* occupying a specific location on the screen. Note that sprites floating over the background tile may change how the character cell actually appears on screen. If you want to deal directly with the display file and what is actually viewable on screen, you should investigate the SCREEN ADDRESS functions.

```
    uint i;
    i = sp_ScreenStr(10, 12);
```

In this case, the code of the background tile at position (10,12) will be in the bottom 8 bits of 'i' and the tile colour at that position will be in the top 8 bits of 'i'.

"sp_GetTiles()" stores the background tiles in a rectangular area on screen into an array. "sp_PutTiles()" prints those background tiles back onto the screen. This can be helpful, for example, if you wanted to display a temporary menu. First you would save what is on screen underneath the menu using "sp_GetTiles()", then you would print your menu (with "sp_PrintString()", likely), process menu commands, and finally you would erase the menu by restoring the screen underneath it with "sp_PutTiles()". The array used to store the background information needs two bytes per character cell in the rectangular area. You may also want to use "sp_GetTiles()" to record a rectangular area and then use "sp_PutTiles()" to print the area on screen in several locations, using the storage array as a macro. An even better way to do this is to use "sp_PrintString()".

"sp_PrintString()" prints a string of characters to the screen as background tiles. What makes it special is that it maintains a current print position and the string can contain embedded commands. A listing of byte commands understood by this function:

| | | | |
|---|---|---|---|
| 0 | 0x00 | Terminate string | |
| 5 | 0x05 | NOP | Ignored character |
| 6 | 0x06 | Goto N | Goto X coordinate on the same line |
| 7 | 0x07 | Print String W | Print string at address W (like a subroutine call) |
| 8 | 0x08 | Move Left | |
| 9 | 0x09 | Move Right | |
| 10 | 0x0a | Move Up | |
| 11 | 0x0b | Move Down | |
| 12 | 0x0c | Home | Move to top left corner of bounds rectangle |
| 13 | 0x0d | Carriage Return | X coordinate to 0, Y coordinate increased |
| 14 | 0x0e | Repeat N | Begins a block within string |
| 15 | 0x0f | End Repeat | Ends the block, repeating it N times |
| 16 | 0x10 | Ink N (0-7) | |
| 17 | 0x11 | Paper N (0-7) | |
| 18 | 0x12 | Flash N (0 / 1) | |

| | | | |
|---|---|---|---|
| 19 | 0x13 | Bright N (0 / 1) | |
| 20 | 0x14 | Attribute N | Set colour to an 8-bit attribute value |
| 21 | 0x15 | Invalidate N (0 / 1) | Set Invalidation mode |
| 22 | 0x16 | At N (y), N (x) | |
| 23 | 0x17 | At N (dy), N (dx) | Relative AT.  The print position will be moved a distance (dy,dx) characters |
| 24 | 0x18 | X Wrap N (0 / 1) | Enable or disable wrapping in X direction |
| 25 | 0x19 | Y Inc N (0 / 1) | Enable or disable increasing Y coordinate when X Wrap occurs |
| 26 | 0x1a | Push State | Save current print state |
| 27 | 0x1b | Escape | Next byte in string will be interpretted as a printable character instead of a command code |
| 28 | 0x1c | Pop State | Pop current print state |
| 29 | 0x1d | Transparent Character | If in invalidate mode, prints a "nothing" but marks the character cell for update.  Will change the colour if attribute != 0x80 |

The 'N's in the table are placeholders for byte-size parameters and the 'W's are placeholders for word-size parameters.  For example, if code 22 is read ('AT'), two bytes follow holding the new y coordinate and then the new x coordinate.

The print string function prints text into a bounding rectangle.  No characters will be printed to the screen outside the rectangle.  If xwrapping is enabled, text will wrap when the horizontal border of the rectangle is hit.  If yinc is enabled, this horizontal wrap will also increment the y coordinate.  Without xwrapping the print position will continue to be updated outside the bounding rectangle, but text will not be printed to screen until the print position returns to the rectangle area.  All coordinates are relative to the top left corner of the bounding rectangle.

The bounding rectangle, current print position, colour and various mode bits are saved in a print string structure (struct sp_PSS):

```
struct sp_PSS {
   struct sp_Rect *bounds;  /* bounding rectangle for printed
text */
   uchar flags;    /* bit 0=invalidate?, 1=xwrap?, 2=yinc?,
3=onscreen? (res.) */
   uchar x;                 /* current x coordinate relative
to bounding rect  */
   uchar y;                 /* current y coordinate relative
to bounding rect  */
   uchar colour;            /* current attribute */
   void *dlist;             /* reserved */
   void *dirtychars;        /* reserved */
   uchar dirtybit;          /* reserved */
};
```

As the string is printed, this structure is updated and will therefore retain state for the next time it is used in a print string call. It is easiest to make changes to this structure through byte command codes in a printed string, but you can make changes directly to the structure if desired. The reserved fields in the structure need to be updated if you make changes to the bounds rectangle or the x/y coordinates. You can do this with an explicit call to "sp_ComputePos" or with an embedded command like 'AT' or 'Home' in the next string printed.

Let's see an example in action. See if you can guess what the following program does and then check the results by running the tap file in your favourite emulator.

```
#include <spritepack.h>

/* embedded relative AT commands would be more compact */

uchar *special =
"\x16\x00\x00\x0e\x04\x1a\x0e\x05\x14\x16AB\x14\x38 \x0b" \

"\x08\x08\x08\x14\x16CD\x14\x38 \x0a\x0f\x1c\x0b\x0b\x0b\x0f";

/* UDG Graphics */

uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};
uchar A[] = {0x55,0xaa,0x54,0xa9,0x01,0x7f,0x3e,0x8f};
uchar B[] = {0x95,0xaa,0x95,0xca,0xc0,0xfe,0xbc,0xf8};
uchar C[] = {0x45,0xa6,0x57,0xa7,0x4e,0xac,0x49,0x92};
uchar D[] = {0xa1,0x6a,0xe5,0xf2,0x75,0x62,0x19,0x0a};

/* Initial State of Print String Struct */

struct sp_Rect r = {9,0,15,32};         /* Bound Rect @ (9,0)
height=15, width=32 */
struct sp_PSS ps = {
   0,                                   /* Illegal C to put
'&r' directly here */
   sp_PSS_INVALIDATE | sp_PSS_XWRAP,    /* Invalidate Mode, X
Wrap */
   0,                                   /* X = 0 */
   0,                                   /* Y = 0 */
   INK_YELLOW | PAPER_RED,              /* Attribute = yellow
on red */
   0,                                   /* reserved */
   0,                                   /* reserved */
   0                                    /* reserved */
};

#pragma output STACKPTR=61440
```

```
main()
{
   uchar x;

   #asm
   di
   #endasm
   sp_InitIM2(0xf1f1);
   sp_CreateGenericISR(0xf1f1);
   #asm
   ei
   #endasm

   sp_TileArray(' ', hash);
   sp_TileArray('A', A);
   sp_TileArray('B', B);
   sp_TileArray('C', C);
   sp_TileArray('D', D);
   ps.bounds = &r;            /* since we couldn't do it in
the static initializer */

   sp_Border(YELLOW);
   sp_Initialize(INK_BLACK | PAPER_CYAN, ' ');

   for (x=15; 1; x=(x-1)&0x1f) {
      sp_UpdateNow();
      special[2] = x;
      sp_PrintString(&ps, special);
      sp_Pause(10);           /* hold down a key to see it go
faster */
   }
}
```

The "special" string contains command codes that print a single star (defined by printing the letters 'AB' and then 'CD' directly underneath) repeated 20 times in a 5x4 array. Under this declaration are declarations for five UDG graphics. These are defined in exactly the same way as they are defined in Basic, with each byte representing a bit pattern for one row in a 8x8 UDG graphic.

The Print String Struct is declared next with several of the fields initialized as indicated in the comments. The reserved fields will be initialized as the string is printed when the 'AT' command is encountered as the first item in the string (the alternative would be to call "sp_ComputePos" explicitly). The initial coordinates are relative to the bounds rectangle. Since the bounds rectangle is located at (Y,X)=(9,0) on the screen, the initial coordinate of (0,0) will actually be mapped to (9+0,0+0) on the screen. The XWRAP flag is set but not YINC. This causes characters printed past the right border of the bounds rect to wrap back to the first column of the bounds rect. Because YINC is not set, the Y coordinate will not be increased when this wrap occurs.

Next we encounter something new. Up to now you will have noticed that we have been printing background tiles by specifying a character code (eg the letter 'A'). I have not mentioned how those character codes are associated with an actual UDG graphic representation. This is done through the tile array, which is an array of pointers to UDG graphic definitions, one for each character code. Sprite Pack automatically initializes this array to point into the ROM character set and that is the reason we could merrily print to the screen without any thought. The "sp_TileArray" calls change the graphic definitions for certain character codes to something new. The first associates the "hash" graphic with the space character.  The next few calls change the graphics associated with the letters A through D to a 2x2 star.

The "sp_Border" call sets the border colour.  The "sp_Initialize" call clears the screen with a black on cyan space character.  The space character has been associated with a hash graphic, hence the new background.

The 'x' variable of the FOR loop is used to track the location of the star array.  Its value is directly poked into the special string's first AT command x coordinate.  That is how the movement is achieved.

SP defines labels for colours that can be ORed together to generate a numerical attribute value.  The list: INK_BLACK, INK_BLUE, INK_RED, INK_MAGENTA, INK_GREEN, INK_CYAN, INK_YELLOW, INK_WHITE, PAPER_BLACK, PAPER_BLUE, PAPER_RED, PAPER_MAGENTA, PAPER_GREEN, PAPER_CYAN, PAPER_YELLOW, PAPER_WHITE, BRIGHT, FLASH, TRANSPARENT.  To form an attribute with green ink, yellow paper and bright you could use (INK_GREEN | PAPER_YELLOW | BRIGHT) rather than the numerical 116.  The special "TRANSPARENT" value (128 - flashing black on black) is reservedd to mean the transparent colour: any existing colour is not altered.  This special colour is only supported when drawing sprites and printing the special 'transparent char' using "sp_PrintString()".  The plain colours BLACK, BLUE, RED, MAGENTA, GREEN, CYAN, YELLOW and WHITE are also defined for use with the "sp_Border" function.  They are, in fact, identical numerically to the INK_* macros.

The last background tile function in the current version of SP is "sp_Pallette()".  This function applies only in the Timex hi-colour mode where the colour resolution is 32x192 rather than the 32x24 colour resolution of the standard Spectrum display.  The pallette array functions very similarly to the tile array.  A colour tile (pallette entry) from 0-255 is associated with an 8-byte 'attribute' UDG.  The attribute value in function calls is this single byte pallette entry which is used to look up the 8 colours used in each character cell.  By default, all pallette entries point at a solid black on white colour when in hi-colour mode.

The other graphical object understood by the sprite module is the sprite.  Sprites are graphical objects that float over the background and amongst themselves.  SP can support any number of any size sprites, with performance obviously suffering as the number and size of *moving* sprites on screen increase (recall that the updater is differential, meaning only areas on screen that change -- perhaps caused by moving sprites -- contribute to draw time.  Static sprites are not redrawn).  Sprites can exist on one of 64 planes, with a lower plane being closer to the viewer.  A sprite on plane 63 would lie just above the background while a sprite on plane 0 would lie above all other sprites and the background.  The draw order (which sprite is on top of which) is indeterminate if two overlapping sprites lie in the same plane.

SP supports four kinds of sprites: mask, or, xor and load.  The kind of sprite determines how the sprite is drawn and how long it takes to draw.  Mask sprites are the slowest and use a mask to erase portions of the screen background that the sprite is drawn into (the mask acts like a cookie-cutter that cuts out the background before the sprite is drawn into it).  With mask sprites it is possible for a sprite to have blank areas that overwrite the screen.  Or and Xor sprites are faster and do not use masks.  The sprite graphic of an or/xor sprite is logically ORed or XORed into the screen.  Without the use of a mask, a sprite's blank areas will not blank out the underlying background.  Load sprites are by far the quickest; they are written directly to the screen, completely obliterating any background or other sprites that may lie underneath them.

The type of sprite you choose will depend on the performance and the effect you are looking for.  Advanced programmers can 'customize' a sprite so that portions of it are different types.  It may be beneficial to performance, for example, to make the interiors of large sprites load-type and the outlines mask-type.

A sprite is a rectangular array of graphical characters having a row height and a column width measured in character units (ie 8x8 pixel squares). You can think of them as big UDGs. Sprite graphics are defined in columns: the graphic definition for all characters in a column must be contiguous. Different columns can be separated arbitrarily in memory. To begin, let's have a look at an example definition:

```
extern uchar col1[];

#asm

._col1
defb @00111100, @00000000
defb @01000010, @00000000
defb @01001010, @00000000
defb @01000010, @00000000
defb @00111100, @00000000
defb @00010000, @10000001
defb @00010000, @11000111
defb @00011000, @11000001

defb @00110100, @00000001
defb @01010000, @00000001
defb @00010000, @00000011
defb @00101000, @00000000
defb @01000101, @00010000
defb @10000010, @00011000
defb @01000000, @00011111
defb @00000000, @11111111

#endasm
```

First off, this is not standard C. I prefer to define sprites using embedded assembler so that I can use binary to better visualize the graphic. You can use any standard C alternative (like an array) to define these graphic bytes in your own program. What I have done here is declare an external array "col1" that points at a single-column graphic. Since this variable is extern, Z88DK will expect to find it defined elsewhere as "_col1" (the name with a preceeding underscore). In a bit of trickery I arrange to declare this name in the embedded assembler fragment above so that the C variable "col1" points at the location of the embedded assembler label "_col1".

As you can see, "_col1" is the location of a walking stickman graphic. "_col1", as with all columns, is one character wide and in this case is two characters tall. Unlike UDGs which use a single byte to define 8 horizontal pixels, sprites use two bytes. The first byte defines the 8 pixels just like with UDGs and the second byte defines the mask. The mask has 1s where the underlying screen will show through the sprite and 0s where the underlying screen will be covered. The 0s in the mask byte inside stickman's head ensures that stickman's head will be drawn as a blank oval no matter what background lies behind the sprite. Only mask-type sprites pay attention to these mask bytes. Or, Xor and Load types ignore the mask byte. In hi-colour mode, a third attribute byte must be defined for each vertical pixel.

To create a sprite, we make a call to "sp_CreateSpr":

```
struct sp_SS *man;
man = sp_CreateSpr(sp_MASK_SPRITE, 2, col1, 1, TRANSPARENT);
```

A successful call to "sp_CreateSpr" places the sprite off screen and returns a pointer to a "struct sp_SS" which holds information like the size and location of the sprite. You can have a look at its definition in the "spritepack.h" header file for more information. The first parameter indicates the sprite type to be created; here it's a mask sprite that's being created. Other options are sp_OR_SPRITE, sp_XOR_SPRITE and sp_LOAD_SPRITE. The next parameter indicates how tall the sprite is in characters. Our stickman is two characters tall. The third parameter points to the graphical definition of the first column in the sprite. Sprites are built up by adding one column at a time; our stickman will have just one column for the time being. The fourth parameter is the plane the sprite will occupy. This can be any number from 0-63 with 0 meaning nearest the viewer and therefore on top of all other sprites and 63 meaning just above the background and below all other sprites. The final parameter depends on what video mode you are in. In Spectrum mode this final parameter is the colour of the entire sprite column. The TRANSPARENT colour, if you remember, means this sprite column will not alter the colour on screen. It is only possible to specify a single colour for the entire column in this function call, but we will investigate later how to individually colour a sprite's character squares using other means. In the hi-res mode this final parameter is ignored and in the hi-colour mode this final parameter is the colour threshold.

Very good, we have a sprite. We can move the sprite anywhere on screen (or off!) using two main functions: "sp_MoveSprAbs" -- move sprite absolute -- and "sp_MoveSprRel" -- move sprite relative. You can probably guess from the names that the former will move a sprite to a specific pixel position while the latter will move it a relative pixel distance from its current position. If you look these functions up in "spritepack.h" you will also see a few variations. These variations will be discussed in a forthcoming section on achieving better performance and not in this overview tutorial.

As always, moving a sprite does not immediately cause the display to change. Display changes only happen when "sp_UpdateNow" is called. Let's see an example with precompiled tap file:

```
#include <spritepack.h>
#pragma output STACKPTR=61440

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar col1[];
uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};
struct sp_UDK keys;

/* Create memory allocator for sprite routines */

void *my_malloc(uint bytes)
{
   return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;


main()
{
   uint reset;
   char dx, dy, i;
   struct sp_SS *man;

   #asm
   di
   #endasm

   sp_TileArray(' ', hash);
   sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
   sp_Border(MAGENTA);
   sp_AddMemory(0, 255, 14, 0xb000);

   keys.up    = sp_LookupKey('q');
   keys.down  = sp_LookupKey('a');
   keys.left  = sp_LookupKey('o');
   keys.right = sp_LookupKey('p');
   keys.fire  = sp_LookupKey(' ');
   reset      = sp_LookupKey('r');

   man = sp_CreateSpr(sp_MASK_SPRITE, 2, col1, 1,
TRANSPARENT);
   sp_MoveSprAbs(man, sp_ClipStruct, 0, 10, 15, 0, 0);

   while(1) {

      sp_UpdateNow();

      i = sp_JoyKeyboard(&keys);
      if ((i & sp_FIRE) == 0) {
         dx = dy = 1;
      } else {
         dx = dy = 8;
      }
      if ((i & sp_LEFT) == 0)
         dx = -dx;
      else if ((i & sp_RIGHT) != 0)
         dx = 0;
      if ((i & sp_UP) == 0)
         dy = -dy;
      else if ((i & sp_DOWN) != 0)
         dy = 0;

      if (sp_KeyPressed(reset))
         sp_MoveSprAbs(man, sp_ClipStruct, 0, 10, 15, 0, 0);
      else
         sp_MoveSprRel(man, sp_ClipStruct, 0, 0, 0, dx, dy);
   }
}


/* Sprite Graphics defined in some inline assembler */

#asm

._col1
defb @00111100, @00000000
defb @01000010, @00000000
defb @01001010, @00000000
defb @01000010, @00000000
defb @00111100, @00000000
defb @00010000, @10000001
defb @00010000, @11000011
defb @00011000, @10000001

defb @00110100, @00000001
defb @01010000, @00000001
defb @00010000, @00000011
defb @00101000, @00000000
defb @01000101, @00000000
defb @10000010, @00010000
defb @01000000, @00011000
defb @00000000, @00011111

#endasm
```

You should be able to understand most of this program but there are a couple of new things in there. The easiest one to explain is the user-defined keyboard joystick. The "struct sp_UDK keys;" declaration creates a structure that stores scan codes for a keyboard joystick, covering left, right, up, down and a single fire button. The scan codes are stored in the structure with calls to "sp_LookupKey" as was done before when we were interested in checking whether single keys were pressed. A later call to "sp_JoyKeyboard" reads this keyboard joystick and returns a single byte that identifies which directions are actively selected on the keyboard. The code shows how the byte is interpretted.

The more complicated item to explain is the memory allocator. As you may recall, the updater is character cell oriented. As sprites are created, they are broken up into character size pieces and each such piece is described by a "struct sp_CS" (see the "spritepack.h" header file) that in turn contains information on how to draw the sprite in each character cell it occupies. So a call to "sp_CreateSpr" not only returns a 14-byte "struct sp_SS" describing the sprite as a whole, but behind the scenes, SP is creating one 14-byte "struct sp_CS" to describe each character cell that makes up a sprite. The question is where does the SP library get this extra memory for these structs from?

The answer is that you must provide a subroutine that will supply available memory on demand. This subroutine is pointed at by "u_malloc" (user malloc) whose full ANSI prototype is "void *(*u_malloc)(uint bytes);". In English, the "u_malloc" function takes a single unsigned int parameter indicating how many bytes are requested and must return the address of that free memory or NULL if no memory is available. If you are familiar with C, you will see that this is exactly what the standard C "malloc" call does. You can, in fact, use the C malloc function by declaring "void *u_malloc = malloc;" (Note: The Z88DK malloc implementation requires you to declare an array from which memory is malloced -- remember we are dealing with a 64K machine here not a multi-megabyte machine with a virtual address space! Here's an example). This extra level of indirection expands options available to programmers.

In this program I did not use the standard C "malloc" as memory allocator. Instead I've used Sprite Pack's dynamic block memory allocator. You can read about the various advantages and disadvantages of this versus using the standard C functions in the DYNAMIC BLOCK MEMORY ALLOCATION section. The block memory allocator manages queues of available fixed-size memory blocks. Queue #0 might contain 5 byte blocks, Queue #1 might contain 10 byte blocks, etc. -- the choice is up to you since you are responsible for adding this memory to the queues in the first place. By default, SP is compiled to manage 5 memory queues, a figure that can be customized in "SPconfig.def" prior to recompiling the SP library.

The only memory requester in this program is the sprite routines which will make requests for 14 byte blocks. The memory allocation function "my_malloc" knows that the byte size request is for a 14 byte block and simply returns an available memory block from queue #0 of the dynamic block memory allocator. In the main() program, a call to "sp_AddMemory" adds 255 fourteen-byte blocks to queue #0 with free memory taken from address 0xb000 (45056) which I know is available.  255 blocks is overkill because the stickman sprite only needs 3 of these blocks when it is created.

The "u_free" function points at a subroutine that can free memory blocks as they are discarded.  The nuances of memory allocation and memory maps are better explained in the DYNAMIC BLOCK MEMORY ALLOCATION section.

Alright then, that was quite the diversion.  Back to sprite movement.  The stickman is initially placed at the centre of the screen by executing "sp_MoveSprAbs(man, sp_ClipStruct, 0, 10, 15, 0, 0);".  The first parameter identifies which sprite will be moved.  The second parameter is the sprite's clipping rectangle.  It is set to "sp_ClipStruct" which defines the full screen but you can make it any rectangle ON SCREEN that you like.  The next parameter is for animation and will be discussed later; it is set to zero to indicate the sprite is not animated.  The last four parameters have to do with the sprite's new location.  A sprite's location is divided into two parts: a character coordinate and a pixel offset within the character.  This statement places the sprite at character coordinate row = 10, column = 15 with 0 rightward horizontal pixel displacement (0..7 is valid) and 0 downward vertical pixel displacement (0..7 is valid).  The full screen is 32 columns wide and 24 rows high (unless you are in the hi-res video mode when there are 64 columns).  Sprites can be moved through 255 horizontal columns and 255 vertical columns.  Clearly, most of this space is off screen.  This off-screen area wraps, meaning a two-column wide sprite drawn at column 255 will have its first column invisible at column 255 and its second column appear in column 0 of the screen.

Stickman's movement is done through a call to "sp_MoveSprRel(man, sp_ClipStruct, 0, 0, 0, dx, dy);".  The parameter list is the same as for "sp_MoveSprAbs" but the location is not absolute this time - it is relative.  The character coordinate -- (0,0) in this call -- is the relative character movement in the Y and X directions with -127 to +128 being valid displacements.  The pixel offset -- (dx,dy) in this call -- indicates relative pixel movement with -127 to +128 being valid.

Run the program to have a go.  The QAOP keys move the sprite one character at a time (8 pixels).  At this step size the sprite gets around rather quickly.  If the sprite gets lost off screen, press 'r' to move it back to the middle of the screen.

This program also allows you to move the sprite a single pixel at a time -- simply hold down the space key while pressing the QAOP direction keys.  If you do this you'll soon notice that something doesn't quite look right!

What went wrong?  The sprite graphic definition is not quite right.  We are fine as long as the sprite is drawn at an exact character coordinate.  Then the sprite's character cells can be drawn exactly into character cells on screen.  However if we try to move the sprite horizontally a fraction of a character, say 4 pixels, things go wrong.  The stickman is one character wide.  How many characters does it occupy if it is moved to the right 4 pixels?  Two!  That is mistake number one: we need to specify the stickman's width as 2 characters.

The second mistake has to do with how SP gets pixel precision in the vertical direction.  When a sprite is moved down a fraction of a character (say N pixels where N<=7), the actual graphic pointers used in the sprite definition are moved back 2*N bytes by the sprite routines (3*N bytes in hi-colour mode).  This has a couple of consequences: 1) Each sprite column's graphics have to be defined contiguously in memory and 2) There needs to be 7 blank pixel rows above each sprite column.  The same situation as with horizontal placement occurs with vertical placement, requiring an extra blank row below the sprite definition.

Repairing the stickman sprite, we have this definition:

```
#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._col1
defb @00111100, @00000000
defb @01000010, @00000000
defb @01001010, @00000000
defb @01000010, @00000000
defb @00111100, @00000000
defb @00010000, @10000001
defb @00010000, @11000011
defb @00011000, @10000001

defb @00110100, @00000001
defb @01010000, @00000001
defb @00010000, @00000011
defb @00101000, @00000000
defb @01000101, @00000000
defb @10000010, @00010000
defb @01000000, @00011000
defb @00000000, @00011111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._col2
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

We started with a sprite 2 rows high and one column wide. We add an extra blank column on the right for pixel precise placement of the sprite in the horizontal direction (note this wouldn't be required if the sprite were only drawn at an exact horizontal character coordinate). We add an extra blank row below each column for pixel precise placement of the sprite in the vertical direction (again not required if the sprite were only drawn at an exact vertical character coordinate). We make sure that there are 7 blank vertical pixels above each sprite column. The result is as shown above. Notice that the blank row at the end of "_col1" doubles as the 7 blank pixels above "_col2". We now have a sprite 3 rows high and 2 columns wide. Modifying the original program somewhat to accommodate this larger sprite, we get a new tap file and source file. Run the tap file to see the difference. Notice how the (now) multi-column sprite is built up by tacking on the extra column using the "sp_AddColSpr" call. In general, sprite columns are added to an existing sprite with repeated calls to "sp_AddColSpr".

Sprite definitions do seem to take up a lot of memory, but there is a way to reduce this, which we will investigate shortly. If you recall, the sprite routines break up the sprite graphic into character size cells and create a "struct sp_CS" to describe each of those sprite characters. The "sp_IterateSprChar" function allows you to access each of these "struct sp_CS" structures one at a time in sequence. In the next example, we will use it to individually colour the squares in the 3x2 stickman sprite. Here is the code fragment:

```
uchar n;
void addColour(struct sp_CS *cs)
{
   if (n == 0)
      cs->colour = INK_BLACK | PAPER_WHITE;
   else if (n == 1)
      cs->colour = INK_BLUE | PAPER_BLACK;
   else
      cs->colour = TRANSPARENT;
   n++;
   return;
}

main()
{
   struct sp_SS *man;

   /* Create the Man Sprite as before, then: */

   n = 0;
   sp_IterateSprChar(man, addColour);

   /* ... */
}
```

"sp_IterateSprChar" iterates over all the characters in a sprite in column-major order and passes a pointer to the corresponding "struct sp_CS" to a user-specified subroutine. In the code fragment above, the man sprite is iterated over and the "addColour" function is passed a "struct sp_CS *" for each character in the man sprite. I have used a global variable "n" to keep track of which sprite character is being considered. N begins at 0 and then is incremented inside the "addColour" subroutine for each sprite character investigated. Using this arrangement and the fact that "sp_IterateSprChar" operates in column-major order, I know that inside "addColour", N=0 for the first character in the first column of the man sprite (the head), N=1 for the second character of the first column of the man sprite (the legs), N=3 for the third character in the first column (blank), N=4 for the first character in the second column of the man sprite (blank), N=5 for the second character in the second column of the man sprite, etc. Inside "addColour" I test N and set the desired colour for the sprite character.

What other properties of sprite characters can we change using this technique? Here's a look at the definition of a "struct sp_CS":

```
struct sp_CS {
   uchar *next_in_spr;  /* big endian!! */
   uchar *prev;         /* big endian!! */
   uchar spr_attr;      /* sprite type (bits 6..7) | sprite
plane (bits 0..5) */
   uchar *left_graphic;
   uchar *graphic;
   uchar hor_rot;
   uchar colour;        /* attribute in spectrum mode,
threshold in hi-colour mode */
   uchar *next;         /* big endian!! */
   uchar unused;
};
```

The "spr_attr" member can be modified to change which sprite plane the character occupies and its sprite type (mask, or, xor and load). Modifying this member would allow interior characters of a sprite to be load type and boundary characters mask type, perhaps improving sprite draw performance. Making the individual characters of a sprite lie in different sprite planes could lead to unique sprite interleaving effects.

The "graphic" member points at the graphical definition of the sprite character (one character = 8 vertical pixels remember). If the sprite is placed vertically some number of pixels off an exact row coordinate, it is this pointer that is modified upward as mentioned earlier. The "left_graphic" pointer points at the graphical definition of the sprite character directly to the left in the same sprite. When the sprite graphic is rotated right to get pixel precise placement in the horizontal direction, this is where the leftmost pixels come from. The first column of a sprite has this member point at a special null sprite, supplied by SP and called "sp_NullSprPtr" (this is a Sprite Pack Variable whose declaration needs to be copied out of the header file into your main program if you intend to use it). The null sprite is simply a blank sprite character. We'll be seeing it used explicitly shortly.

The "struct sp_SS" is used to describe the sprite as a whole, providing details such as its current location. Some of its members also specify sprite plane and type, but these are only used when a new sprite column is added to the sprite with "sp_AddColSpr". It is the information in a "struct sp_CS" that controls how an individual sprite character is drawn.

The extra blank column added to sprite definitions to allow pixel precise placement in the horizontal direction takes up a lot of extra memory. There is a way to avoid this: make the graphics in the final column point at the null sprite character rather than actually defining a blank column in memory. This can be done simply in this example by adding:

```
    if (n > 2)
       cs->graphic = sp_NullSprPtr;
```

to the "addColour" subroutine. This will change the graphic pointed at by sprite characters in column 2 to the null sprite (a blank character). Instead of defining this second column and adding it to the sprite definition using "sp_AddColSpr" as in the last example, we can add a "dummy" second column having the same graphics as the first column. Then we blank out the graphics in the second column using the new "addColour" subroutine. Here is the result with precompiled tap file:

```
#include <spritepack.h>
#pragma output STACKPTR=61440

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar *sp_NullSprPtr;
#asm
LIB SPNullSprPtr
._sp_NullSprPtr         defw SPNullSprPtr
#endasm

extern uchar col1[];
uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};
struct sp_UDK keys;

/* Create memory allocator for sprite routines */

void *my_malloc(uint bytes)
{
   return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;

/* Iterate over sprite subroutine */

uchar n;
void addColour(struct sp_CS *cs)
{
   if (n == 0)
      cs->colour = INK_BLACK | PAPER_WHITE;
   else if (n == 1)
      cs->colour = INK_BLUE | PAPER_BLACK;
   else
      cs->colour = TRANSPARENT;

   if (n > 2)
      cs->graphic = sp_NullSprPtr;
   n++;
   return;
}

main()
{
   uint reset;
   char dx, dy, i;
   struct sp_SS *man;

   #asm
   di
   #endasm

   sp_TileArray(' ', hash);
   sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
   sp_Border(MAGENTA);
   sp_AddMemory(0, 255, 14, 0xb000);

   keys.up    = sp_LookupKey('q');
   keys.down  = sp_LookupKey('a');
   keys.left  = sp_LookupKey('o');
   keys.right = sp_LookupKey('p');
   keys.fire  = sp_LookupKey(' ');
   reset      = sp_LookupKey('r');

   man = sp_CreateSpr(sp_MASK_SPRITE, 3, col1, 1,
TRANSPARENT);
   sp_AddColSpr(man, col1, TRANSPARENT);
   n = 0;
   sp_IterateSprChar(man, addColour);
   sp_MoveSprAbs(man, sp_ClipStruct, 0, 10, 15, 0, 0);

   while(1) {

      sp_UpdateNow();

      i = sp_JoyKeyboard(&keys);
      if ((i & sp_FIRE) == 0) {
         dx = dy = 1;
      } else {
         dx = dy = 8;
      }
      if ((i & sp_LEFT) == 0)
         dx = -dx;
      else if ((i & sp_RIGHT) != 0)
         dx = 0;
      if ((i & sp_UP) == 0)
         dy = -dy;
      else if ((i & sp_DOWN) != 0)
         dy = 0;

      if (sp_KeyPressed(reset))
```

```
          sp_MoveSprAbs(man, sp_ClipStruct, 0, 10, 15, 0, 0);
       else
          sp_MoveSprRel(man, sp_ClipStruct, 0, 0, 0, dx, dy);
    }
}


/* Sprite Graphics defined in some inline assembler */

#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._col1
defb @00111100, @00000000
defb @01000010, @00000000
defb @01001010, @00000000
defb @01000010, @00000000
defb @00111100, @00000000
defb @00010000, @10000001
defb @00010000, @11000011
defb @00011000, @10000001

defb @00110100, @00000001
defb @01010000, @00000001
defb @00010000, @00000011
defb @00101000, @00000000
defb @01000101, @00000000
defb @10000010, @00010000
defb @01000000, @00011000
defb @00000000, @00011111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

I deliberately chose the colours so that the colour-clash problem would be obvious. Colours do not follow the actual graphic around very well. It is up to you to solve this problem. The Timex hi-colour mode has no vertical colour clash but still has the same horizontal colour clash problem. The "colour threshold" parameter in each sprite character in this mode determines how much the sprite has to be moved horizontally within a character square before the colour is also rotated horizontally. Using sensible values can help in relieving horizontal colour clash.

Once you are finished with a sprite, you can delete it by first moving it off screen (important!) and then calling "sp_DeleteSpr". "sp_DeleteSpr" will call "u_free" to reclaim all the memory that was invisibly allocated to create the "struct sp_CS" and "struct sp_SS" structs when the sprite was created.

The last topic in sprites to cover is animation. If you've ever experimented with flick books as a child, you will already know that a sprite can be animated by replacing its graphic image by successively modified ones in quick sequence. Each of these images is called a specific "frame" of the sprite. The animate parameter in the "sp_MoveSprAbs" and "sp_MoveSprRel" subroutines allow you to specify the relative location in memory of the next frame for the sprite. By specifying this animation parameter, the sprite routines will automatically draw the next frame. An interesting example in action is the "PacMen" demo appearing later in the tutorial. Grab the tap file now to give it a go. Be sure to try clicking and dragging with one of the mice. We haven't covered enough material to discuss how this works yet.

## IM 2 INTERRUPTS

Interrupts? What are they? It's really quite simple. If you've ever been eating dinner when the phone rang, you've been interrupted. Typically you would answer the phone and when the conversation was over, you'd hang it up and resume eating your dinner. CPUs also get interrupted. In the Spectrum, the machine code program known as the "BASIC Interpretter" goes through your program one line at a time, figuring out what you want it to do and then it does it. The ULA interrupts the Z80 while it's going about interpretting your BASIC program every time the TV display is drawn, 50 times a second if you live in Europe and 60 times a second if you live in North America. The Z80 stops what it is doing, jumps to a special program -- BASIC's Interrupt Service Routine (ISR) -- to do the things it is supposed to do when the ULA interrupts and when it's done, it returns to interpretting your BASIC.

The Z80 supports two kinds of interrupts, namely maskable and non-maskable. The non-maskable interrupt (NMI) is not commonly used by attached hardware and will not be discussed here. The maskable interrupt (MI), on the other hand, is frequently used by external peripherals to asynchronously request service. The Z80 has a dedicated pin for this purpose (/INTRQ); whenever an external peripheral would like some attention from the CPU it can pull this pin low and cause a maskable interrupt to occur. The reason this kind of interrupt is called 'maskable' is because it can be ignored in software by executing a 'DI' instruction. With maskable interrupts enabled (the 'EI' instruction), exactly how the Z80 responds depends on which of three interrupt modes it is in: IM0, IM1 or IM2.

IM0 is activated by executing the 'IM 0' instruction. This mode is identical to the 8080's interrupt response. In this mode, when the Z80 acknowledges an interrupt, the interrupting peripheral is expected to place an instruction on the data bus that the Z80 then executes. Peripherals typically supply a 'RST x' instruction, which is a single byte call to one of eight fixed entry points in the first 256 bytes of memory. More sophisticated peripherals could supply a 'CALL' instruction with a specific subroutine address. This interrupt mode is rarely used in Z80 applications and presumably is there for 8080 compatibility.

IM1 is activated by executing the 'IM 1' instruction. This is the mode that Spectrum BASIC operates from. In this mode, the Z80 will respond to a maskable interrupt by executing a restart to address 0x0038 (a restart is a subroutine call). Thus code to deal with all interrupting peripherals needs to be located at 0x0038. On the unexpanded Spectrum, the sole interrupting device is the ULA which will generate a maskable interrupt just before it begins to draw the video display. This interrupt occurs every 1/50s. The Spectrum ROM code at 0x0038 uses this interrupt to update the system clock and to read the keyboard. In more general applications, this mode allows attached peripherals to be simple since all they need to do is assert the /INTRQ pin and no more (in IM0, for example, the peripheral also had to supply an instruction on the data bus at the right time). The software at 0x0038 is more complex because it must determine which peripheral caused the interrupt and then deal with it. This may be done by polling every single attached peripheral (ie ask them: Did you just interrupt me?) or external hardware may latch an identifier for the interrupting peripheral when it causes an interrupt. The software can read this register to immediately determine who caused the interrupt.

IM2 is the most powerful interrupt mode and is activated by executing the 'IM 2' instruction. This is a vectored interrupt mode: when the Z80 acknowledges the interrupt, the interrupting peripheral is expected to supply an 8-bit identifier. The contents of the 'I' register and the 8-bit identifier are concatenated (with 'I' being the most significant 8-bits) to form a 16-bit address that contains the address of the interrupt service routine to jump to. Effectively, I*256 points at a 256 byte table that contains interrupt service routine addresses. The peripheral supplies the index into the table to find the right interrupt service routine. By convention, the interrupting peripheral's identifier is even to make things work smoothly. This ensures that table entries do not collide. Eg: A peripheral with ID 0 uses addresses I*256+0 and I*256+1 to store its interrupt service routine address. Peripheral ID 2 uses addresses I*256+2 and I*256+3 to store its ISR address. A peripheral with ID 1 would have table addresses that conflicted with both.

That said, the Spectrum's sole interrupter (the ULA) was not designed for working in IM2. It does not supply an identifier when the Z80 expects one. However, since nothing controls the data bus at the appropriate time, passive resistors pull the data bus to a predictable value: 255. The Z80 will therefore see a peripheral ID of 255 whenever the ULA interrupts. This is an odd value that will cause an ISR fetch from table addresses I*256+255 and I*256+256. For the Spectrum, therefore, we'll need a 257 byte interrupt vector table. Everything is not gold in Spectrum land, however. Certain common Spectrum peripherals were improperly designed and could spew random data on the data bus when the Z80 is looking for a peripheral ID. Therefore, with one of these devices attached, the ULA's peripheral id could become a random bit pattern. A similar problem is faced by the TS2068 which does not have pull-ups on all its data bus lines. The TS2068's SCLD peripheral ID, like the ULA ID, could therefore be a random bit pattern. The solution used by many is to use a 257 byte table containing the same byte value repeated 257 times. Then the ISR address looked up will have the same value no matter what the peripheral ID is. With Sprite Pack, a good value to store is 0xf1 in all 257 bytes of the im2 vector table. The interrupt service routine called will always be at 0xf1f1 no matter what the peripheral ID is. This kind of defeats the purpose of vectored interrupts as it is no longer known which peripheral caused an interrupt. Effectively, we're left with an IM1 mode that jumps to a user-definable address rather than the fixed 0x0038 address. This is how most Spectrum programmers use IM2.

All is not lost however. The Spectrum can reap the full power of IM2 by simply not attaching offending peripherals. The AMX mouse is an example of an interrupting peripheral that functions in IM2. The TS2068 can fully use IM2 if attached IM2 peripherals pull up all the data bus lines.

Sprite Pack fully supports the IM2 interrupt mode. The first step is to initialize the IM2 mode. Do this by calling 'sp_InitIM2(void *default_isr);". This will construct the IM2 vector table for you, initialize all the vectors to jump to the same default interrupt service routine, set im2 mode and load the 'I' register with the appropriate value. The 257 byte IM2 vector table is located at the address specified in the "SPconfig.def" file prior to library compilation. You don't really need to worry about this as it's all been set to sensible default values for you.

The "default_isr" parameter is the address of the default interrupt service routine. If you like, you can use "sp_EmptyISR()", supplied by Sprite Pack, as your default. This empty ISR does exactly what the name suggests: nothing. It simply reenables interrupts and returns.

Once the IM2 mode has been set up, interrupt service routines can be registered on specific vectors using "sp_InstallISR(uchar vector, void *isr);". The 'vector' parameter is the 8-bit peripheral ID supplied by the interrupting peripheral. As mentioned earlier this should be an even number. On the Spectrum and clones, the obvious exception is the ULA ID of 255. The 'isr' parameter is the address of the interrupt service routine to associate with the vector. Because an interrupt can occur at any time, the subroutine that services an interrupt must make sure that the Z80's state was not altered from the moment it was called before it returns. Thus ISRs must ensure that registers are saved when called and restored prior to returning and that interrupts are re-enabled prior to returning. These are rather low-level details so this subroutine is really meant for registering assembly language ISRs.  You can get away with a C function registered in this way but you would have to embed assembler into the C function to handle these details.

The alternative is to register Sprite Pack's Generic ISR on a particular vector to handle the interrupt.  The Generic ISR can have multiple 'hooks' registered on it in an ordered list.  When an interrupt occurs on the Generic ISR's vector, each of these hooks are run one after the other.  If any hook sets its carry flag on return ('SCF' in assembler or 'return_c(0);' from Z88DK C) the rest of the hooks in the list will not be run.  This kind of behaviour is common when multiple peripherals are hard-wired to respond on the same vector (ie they supply the same peripheral ID).  In this case, ISRs for these peripherals are registered as hooks on the vector.  The ISR for each peripheral should first check if that peripheral actually generated the interrupt.  If so, it should handle it and set the carry flag on return so that the subsequent ISRs are not run.  If the ISR does not handle the interrupt it should return with the carry flag reset so that the next ISR hook *is* run.  In the context of the Spectrum, multiple hooks on the ULA interrupt (vector 255) can allow several independent subroutines that need to be run every 1/50s to be executed one after the other.  The first might scan the keyboard and a second might play music on the AY chip, for instance.  In this case you would want both hooks to run so both ISRs would return with carry flag reset.  NOTE: A hook must have 2 bytes free prior to its start address.  The Generic ISR takes care of saving and restoring all registers for you.

This is a lot of information to absorb, especially for beginners.  An example will demonstrate how easy it all is.

```
#include <stdio.h>
#include <malloc.h>
#include <spritepack.h>
#pragma output STACKPTR=61440

/* Sprite Pack Variables Copied from "spritepack.h */

/* a rectangle that defines the screen area */
extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

/* Global Variables */

struct {
   uchar frames;
   uchar seconds;
   uchar minutes;
   uchar hours;
} time;

extern uchar *lastk;           /* system variable LAST_K */
#asm
   ._lastk  defw 23560;
#endasm

uchar stop = 0;                /* stop clock? */
struct sp_PSS ps;
uchar timestring[] = "\x16\x0c\x0a00:00:00:00";
uchar brick[] = {0xff,0x22,0x22,0x22,0xff,0x88,0x88,0x88};
```

```
extern uchar arrowspr1[];
extern uchar arrowspr2[];

/* Keyboard Scanning Hook */

#asm
defw 0                        /* two free bytes before start
of hook */
#endasm
int keyboard_isr(void)
{
   uchar c;

   if ((c = sp_GetKey()) != 0)
      *lastk = c;

   if (stop)
      return_c(0);
   return_nc(0);
}

/* Clock Hook */

#asm
defw 0                        /* two free bytes before start
of hook */
#endasm
int clock_isr(void)
{
   if (++time.frames == 50) {
      time.frames = 0;
      if (++time.seconds == 60) {
         time.seconds = 0;
         if (++time.minutes == 60) {
            time.minutes = 0;
            if (++time.hours == 24)
               time.hours = 0;
         }
      }
   }
   return_nc(0);
}

/* user memory allocation policy uses C's malloc */

HEAPSIZE(256)            /* malloc's heap will be 256 bytes in
size */
void *u_malloc = malloc;
void *u_free = free;

main()
{
   uchar i;
   uint key_S, key_s, x, y;
   struct sp_SS *ss;
   uchar *temp1, *temp2;

/* Set Up IM2 Vectors -- comments follow listing */

   #asm
   di
   #endasm
   sp_InitIM2(0xf1f1);
   sp_CreateGenericISR(0xf1f1);
   sp_RegisterHook(255, keyboard_isr);
   sp_RegisterHookLast(255, clock_isr);
   sp_MouseAMXInit(0, 2);
   #asm
   ei
   #endasm

   heapinit(256);   /* initialize malloc heap */
   sp_TileArray(' ', brick);
   sp_Initialize(INK_BLACK | PAPER_WHITE, ' ');
   sp_Border(CYAN);

   ps.bounds = sp_ClipStruct;
   ps.flags = sp_PSS_INVALIDATE | sp_PSS_XWRAP | sp_PSS_YINC;
   ps.colour = INK_WHITE | PAPER_RED;

   /* Create a small colourless arrow pointer */

   ss = sp_CreateSpr(sp_MASK_SPRITE, 3, arrowspr1, 0,
TRANSPARENT);
   sp_AddColSpr(ss, arrowspr2, TRANSPARENT);

   key_S = sp_LookupKey('S');
   key_s = sp_LookupKey('s');

   *lastk = 0;
   while (1) {

      sp_UpdateNow();

      /* Move Arrow Sprite */

      sp_MouseAMX(&x, &y, &i);
      sp_MoveSprAbs(ss, sp_ClipStruct, 0, y/8, x/8, x&7, y&7);

      /* Print the Time */

      temp1 = &time.hours; temp2 = timestring+3;
      while (*temp2) {
         *temp2 = (*temp1)/10+'0'; temp2++;
         *temp2 = (*temp1)%10+'0'; temp2++;
         temp1--; temp2++;
      }
      sp_PrintString(&ps, timestring);

      /* Show Results of Key Read */

      i = getk();
      if ((i > 31) && (i < 129))
         sp_PrintAtInv(1, 1, INK_BLUE | PAPER_YELLOW, i);

      /* Stop or Start the Clock */

      if (sp_KeyPressed(key_S))
         stop = 1;
      else if (sp_KeyPressed(key_s))
         stop = 0;
   }
}

/* Arrow Sprite */

#asm
defb 0,255,0,255,0,255,0,255,0,255,0,255,0,255   ; 7 blank
pixels above sprite

._arrowspr1
defb @00000000, @00111111
defb @01000000, @00011111
defb @01100000, @00001111
defb @01110000, @00000111
defb @01111000, @00000011
defb @01111100, @00000001
defb @01111110, @00000000
defb @01101000, @00000000

defb @01001000, @00000011
defb @00000100, @00110001
defb @00000100, @11110001
defb @00000010, @11111000
defb @00000010, @11111000
defb @00000000, @11111100
defb @00000000, @11111111
defb @00000000, @11111111

defb 0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255   ; blank
row below sprite
                                                       ; also 7
pixels above arrowspr2
._arrowspr2
defb 0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255
defb 0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255

defb 0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255   ; blank
row below sprite
```

```
    #endasm
```

Once again there are a few functions in this program we haven't seen yet but their purpose is fairly obvious. Let's start the discussion in main().

The first block of code sets up the IM2 environment. Whenever changes are made to the IM2 environment, interrupts must be disabled. A little inline assembler takes care of this. The call to "sp_InitIM2" initializes the IM2 mode and sets all vectors to point at the interrupt service routine at address 0xf1f1 (this ISR subroutine does not exist yet, btw). Notice that the ISR address has the same LSB and MSB bytes (0xf1). This will cause a 257-byte IM2 vector table to be created containing 0xf1 bytes. This is suitable for Spectrums, TS2068s and their clones where the ULA may place a random ID on the bus during interrupt acknowledge.

The next call to "sp_CreateGenericISR" makes a copy of SP's Generic ISR subroutine to address 0xf1f1. At this point all interrupts will be handled by this specific generic ISR because all vectors point at it. The Generic ISR subroutine is 'GENERICISRSIZE' bytes long (currently 12 bytes; this number is unlikely to change). By default, the memory map set up by Sprite Pack conveniently leaves a small pocket of memory available at 0xf1f1 to hold this generic ISR subroutine.

Next two hooks are attached to the generic ISR on vector 255, the ULA interrupt vector. Since the ULA interrupts every 1/50s, these hooks will be run every 1/50s. The first hook is the keyboard_isr subroutine and the other is the clock_isr subroutine. Because these subroutines are hooks, they require two free bytes prior to their start. If you look at their function definitions you'll see that the two free bytes are supplied by a little inline assembler reserving a blank word. If you are curious, the two bytes are used as a next pointer in a linked list of hooks.

There are two ways to register a hook. One way is with "sp_RegisterHook" and "sp_RegisterHookLast" which are synonyms for registering a hook at the end of a Generic ISR's hook list. The other way is with "sp_RegisterHookFirst" which will register a hook at the beginning of a Generic ISR's hook list. Order is important with hooks because any hook setting a carry flag before returning will cause all subsequent hooks not to be run in the current interrupt. REMEMBER that hooks can only be registered with the Generic ISR as supplied by Sprite Pack. A plain ISR installed on a particular vector with "sp_InstallISR()" cannot have hooks registered for it. SP's Generic ISR is a special ISR supplied by SP that can handle hooks.

Z88DK's stdin functions expect to detect key presses by reading Spectrum Basic's LAST_K system variable at address 23560. Under normal operation the Spectrum is in IM1 and the ULA interrupt therefore causes code at address 0x0038 to be run. One of the things this code does is scan the keyboard. If a key is pressed, the ascii code of the key is stored in LAST_K at 23560. Spectrum Basic (or Z88DK's stdin functions if a C program is running) will notice that LAST_K contains this code and will read it to interpret a key press. After reading it, LAST_K is written with 0. This way a single keypress is interpretted exactly once. The ROM code at 0x0038 automatically implements key repeat so that if a particular key is held down, LAST_K is periodically written with the ascii code of the key pressed.

Now that we live in IM2 mode, the ROM interrupt service routine at 0x0038 will no longer run. Actually we *could* run it by deliberately calling it, but it makes no sense to suffer all the overhead of the Basic system when our program will not be using Basic. Since Z88DK wants to read keyboard input from LAST_K, we will have to scan the keyboard ourselves and put the key code into LAST_K. "keyboard_isr" does exactly this. The call to "sp_GetKey()" returns the ascii code of a key pressed or 0 if no key is pressed. It also features key repetition as explained more fully later in the tutorial. If the ascii code returned by "sp_GetKey()" is not 0, it is stored into the system variable LAST_K for consumption by Z88DK's stdin library. You may want to have a look at how LAST_K's address of 23560 is attached to the C pointer variable "lastk". Before keyboard_isr returns it checks if "stop" is true (non-zero). If it is, the carry flag is set making sure that the next hook is not run. The next hook is the clock_isr so this will effectively stop the clock!

The clock_isr keeps track of the time in the "time" structure.  The idea is that clock_isr is run every 1/50s (on a North American TS2068 this will be every 1/60s; adjust the clock accordingly!) and all time accumulation follows from that.

Returning to the initialization of the IM2 mode, the two hooks have been registered with calls to "sp_RegisterHook".  The following call to "sp_MouseAMXInit(0,2);" installs the AMX mouse on two interrupt vectors.  The AMX mouse is an IM2 peripheral that will generate two different interrupts.  One interrupt is generated when the mouse is moved a single unit in the X direction and the other is generated when the mouse moves a single unit in the Y direction.  As with most Z80 peripherals, the peripheral ID the AMX interface uses for each interrupt is programmable and must be even (it's MUST because the AMX hardware cannot store an odd peripheral ID).  Any two non-conflicting vectors will do and I have selected vectors 0 and 2.  Note that vector 254 would conflict with the ULA's 255 vector.  SP takes care of initializing the AMX interface with this call and installs appropriate interrupt service routines on vectors 0 and 2 to track the mouse position.

With this done, the IM2 setup is complete and interrupts are reenabled with some inline assembler.

If you are reading this tutorial in sequence the rest of this program should be understandable with only a few exceptions.  The "sp_MouseAMX" function reads the current pixel location and button state of the mouse.  Z88DK's "getk()" function reads system variable LAST_K for keypresses.  Other stdin functions (fgets, fgetc, etc) would also read LAST_K for keyboard input.  The order of testing for a capital-S keypress and a lower-case-s keypress using "sp_KeyPressed" is important (a capital-S keypress checks the state of both the CAPS and S keys; a lower-s keypress only checks the state of the S key).

One note: not all emulators support the AMX mouse but one that does is Spectaculator.  The purpose of this program is to illustrate the use of IM2 mode, which the AMX mouse requires.

And finally, a word on the Spectrum's built-in ISR at 0x0038.  If you do not switch to IM2 or disable interrupts, your C programs will periodically be interrupted by the ULA and will have this interrupt serviced by BASIC's ISR at 0x0038.  As mentioned above, this ISR reads the keyboard and updates a clock (the FRAMES system variable).  BASIC's ISR expects to have exlcusive use of the Z80's IY register to index into the system variables memory area.  For your program to function well with the BASIC ISR running, it cannot use the IY register, or at the least, disable interrups while using it and then restore it to its expected value prior to reenabling interrupts.

Several functions in SP use the IY register pair: **sp_MoveSprAbs*, sp_MoveSprRel*, sp_IterateSprChar, sp_IntIntervals, sp_IntLargeRect, sp_IntRect, sp_Invalidate, sp_Validate, sp_IterateDList, sp_PrintString**.  If any of these functions are called  by your program, it will likely crash unless you have disabled interrupts or created your own ISR to service the ULA interrupt.

## INPUT

Sprite Pack comes with a rich set of functions to read the keyboard, a variety of joysticks and a handful of different mice.  Let's talk about the keyboard first.

Four handy functions are "sp_WaitForKey()", "sp_WaitForNoKey()", "sp_Pause(ticks)" and "sp_Wait(ticks)".  The first two functions do what their names suggest: they wait for any key to be pressed and for no keys to be pressed respectively.  If at any point in a program you want to wait for a keypress, the following code fragment will do the trick:

```
    sp_WaitForNoKey();
    sp_WaitForKey();
```

This pair will ignore keypresses that occur prior to hitting this point in the program.

"sp_Pause(ticks)" works just like the Basic PAUSE command. The statement will terminate after a key is pressed or after 'ticks' interrupts have occurred, whichever is first. If the ULA is the sole interrupter then 'ticks' is measured in 1/50s. A warning: if interrupts are disabled, this fragment will lock up the machine (it executes 'HALT' to wait for an interrupt to occur).

```
sp_Pause(250);
```

Will pause for 5 seconds or until a key is pressed. "sp_Wait(ticks)" is identical except that it waits out the full time, ignoring any keypresses.

You can check the instantaneous state of the keyboard using scan codes. A scan code is a 16-bit number that describes a single key and CAPS/SYM shift combination. Scan codes are retrieved using "sp_LookupKey()". Using a scan code, "sp_KeyPressed()" can be called to determine if that key is currently pressed. Here are some examples:

```
uint key_A, key_b, key_quote, key_copyright, key_ESC;

key_A = sp_LookupKey('A');
key_b = sp_LookupKey('b');
key_quote = sp_LookupKey('\"');
key_copyright = sp_LookupKey(127);
key_ESC = sp_LookupKey(27);

if (sp_KeyPressed(key_copyright))
   printf("Copyright!\n");
...
```

"sp_LookupKey()" accepts an ascii code as its parameter and returns the corresponding 16-bit scan code. You'll notice that ascii codes that are not normally generated by the Spectrum keyboard can be generated using SP. This is because SP maps its own set of ascii codes to the Spectrum keyboard. Four unique ascii codes are assigned to each key: the unshifted code, the CAPS shifted code, the SYM shifted code and the CAPS+SYM shifted code. The connection between keypress and ascii code is stored in the key translation table (".\globalvariables\SPkeytranstbl.asm"). An effort has been made to simulate the PC keyboard, with the use of CAPS+SYM shift as a CTRL key. The location of the key translation table is exposed through the SP variable "sp_KeyTransTable". If desired you can modify ascii code and key associations using this table. In this example, the 'A' char is generated by 'CAPS+A', the 'b' char is generated by 'B' (note that the state of CAPS is *not* checked for lower case letters), the " char is generated by 'SYM+P', the copyright symbol is generated by 'CAPS+SYM+0' and the escape character is generated by 'CAPS+SYM+1'. Since there are only 160 unique ascii codes that can be generated on the 40 key Spectrum keyboard, sp_LookupKey may not be able to return a valid scan code for a corresponding ascii code. You can check for this using Z88DK's special "iferror (...)" test. "Iferror" functions like an "if" statement but checks the state of the Z80's carry flag. If the carry flag is set, the body of "iferror" is executed. "sp_LookupKey" sets the carry flag if the ascii code cannot be generated by the keyboard.

The previous functions will work when multiple keys are simultaneously pressed. Sometimes you are only interested in a single keypress, for example when a document is typed up in a word processor. "sp_Inkey()" samples the current state of the keyboard and returns the ascii code of a single keypress. It will return 0 if no keys or multiple keys are pressed.

"sp_GetKey()" is a more sophisticated function. It will return the ascii code of a keypress or 0 if no keys or multiple keys are pressed as with "sp_Inkey()". However, it also has a key repeat feature. Three Sprite Pack variables affect "sp_GetKey()"'s behaviour:

```
uchar *sp_KeyDebounce;
uchar *sp_KeyStartRepeat;
uchar *sp_KeyRepeatPeriod;
```

To use these variables you must copy their declarations from "spritepack.h" into your main file. "sp_KeyDebounce" determines how long a key has to be pressed before it is initially noted as pressed. At this point an "sp_GetKey()" will return the ascii code for the key. Once the key is read, "sp_KeyStartRepeat" determines how long the key needs to be pressed before it starts to repeat. After this time has passed, "sp_GetKey()" will again return the ascii code for the key. After the key is again read, and thereafter, "sp_KeyRepeatPeriod" determines the repeat rate of the keypress. If the key is ever released or multiple keys are pressed at the same time, "sp_GetKey" starts again from square one at key debounce. The measurement of time is actually in units of "how frequently sp_GetKey is called". For example, if "sp_GetKey" is called as part of an interrupt service routine, time is measured in 1/50s.

The following program lets you type on the screen and experiment with these settings. A precompiled tap file is also available for running on an emulator.

```
SORRY EXAMPLE NOT COMPLETE YET
```

SP also provides a variety of joystick functions:

```
uchar sp_JoySinclair1(void);
uchar sp_JoySinclair2(void);
uchar sp_JoyTimexLeft(void);
uchar sp_JoyTimexRight(void);
uchar sp_JoyTimexEither(void);
uchar sp_JoyFuller(void);
uchar sp_JoyKempston(void);
uchar sp_JoyKeyboard(struct sp_UDK *keys);
```

As you can see Sinclair, TS2068, Fuller Box, Kempston and User-Defined Keyboard joysticks are supported. All joystick functions return a byte in the same format. This byte can be tested for the four directions and a single fire button using the built in bit masks sp_RIGHT, sp_LEFT, sp_UP, sp_DOWN and sp_FIRE. The values returned by the joystick functions are active low, meaning the result of an AND with one of these masks should be zero to indicate the direction is asserted.

The keyboard joystick function accepts a parameter describing the scan codes associated with each direction and the fire button in the sp_UDK structure.

Because each joystick function returns the same format byte value, it is easy to use a user-selected joystick in your programs. Here is a code fragment that assigns a user-selected joystick function to a function pointer:

```
struct sp_UDK keys;
void *joyfunc;       /* void (*joyfunc)(void) or void
(*joyfunc)(struct sp_UDK *) */

main()
{
   /* some default keys for the user-defined keyboard joystick
*/

   keys.up    = sp_LookupKey('q');
   keys.down  = sp_LookupKey('a');
   keys.left  = sp_LookupKey('o');
   keys.right = sp_LookupKey('p');
   keys.fire  = sp_LookupKey('m');
   ...

   /* a menu sets 'selected' to the user's choice of joystick
*/
   /* then we use 'selected' to choose an appropriate joystick
function */

   if (selected == 0)
      joyfunc = sp_JoyKempston;
   else if (selected == 1)
      joyfunc = sp_JoySinclair1;
   else if (selected == 2)
      joyfunc = sp_JoyTimexLeft;
   else
      joyfunc = sp_JoyKeyboard;

   ...

   /* later in the program the joystick is read */

   result = (joyfunc)(&keys);
   if ((result & sp_UP) == 0)
       /* do up action */
   if ((result & sp_FIRE) == 0)
       /* do fire action */

   ...

}
```

Z88DK does not allow function pointers to be prototyped so I've declared the function pointer 'joyfunc' as void* type. In the program, 'joyfunc' is assigned a pointer to one of the joystick functions depending on the user's choice from a menu. Later on a call is made to the appropriate joystick function using this function pointer. The "sp_JoyKeyboard" function is different from the other joystick functions in that it requires a single parameter. The function pointed at by 'joyfunc' is *always* called with this extra parameter. Only the keyboard joystick function will make use of the parameter; all other joystick functions will ignore it. This is a standard C feature: functions in C are what is called "caller-save", meaning the caller is responsible for setting up the parameters on the stack, calling the destination routine and then for restoring the stack. As the caller, "main()" will place the "&keys" parameters on the stack before calling "joyfunc" in the code fragment above. Most of the joystick functions will not bother looking for this parameter on the stack because they don't expect to find one there. "sp_JoyKeyboard" will use this parameter, however. On return from the joystick function, "main()", being the caller, will remove this parameter from the stack, leaving things as they were before the joystick function was called. In short, everything works out okay and this 'trick' doesn't even count as a hack!

Finally, the AMX mouse, the Kempston Mouse and a simulated mouse can be read through SP:

```
void sp_MouseAMX(uint *xcoord, uchar *ycoord, uchar *buttons);
void sp_MouseKempston(uint *xcoord, uchar *ycoord, uchar
*buttons);
void sp_MouseSim(struct sp_UDM *m, uint *xcoord, uchar
*ycoord, uchar *buttons);
```

The same trick can be done as with the joystick functions to support a variety of mice using a single function call through a pointer to a mouse function (see the PacMen demo which does exactly this).

All mouse read functions return the current pixel coordinate of the pointer and the state of the mouse button(s). The Y coordinate returned is an 8-bit value confined to 0..191 (or less if SP was compiled to use a subset of the full screen's row range) but the X value returned is a full 16-bits. The reason is that these mouse functions are written to work in all of SP's supported video modes. In the hi-res mode, the X coordinate can be in the range 0..511. In the other video modes, however, the X coordinate will be confined to the expected 0..255 range. Masks in "spritepack.h" let you test the returned "buttons" parameter for the state of the left, right or middle mouse buttons. The Kempston mouse has two buttons, the AMX has three and the simulated mouse has just one.

The Kempston mouse is simplest to use because it requires no extra setup. Simply call the function as the prototype shows to get the mouse's current position and button state.

The AMX mouse on the other hand is built to use the IM2 mode. Because of this, both it and the IM2 mode must be initialized when the program starts. The full procedure is shown in an example in the INTERRUPTS section and will not be repeated here.

> Advanced Users Only: Adjusting AMX Mouse Sensitivity
>
> Because of the way the AMX mouse works (an interrupt is generated whenever the mouse moves one unit in the X or Y directions), it is possible to scale the movements of the mouse to make it more or less sensitive. This sensitivity can be changed by modifying the settings pointed at by Sprite Pack variables "uint *sp_MouseAMXdx" and "uint *sp_MouseAMXdy". The X and Y coordinates of the mouse are stored internally in a 16-bit binary fixed point format. The placement of the binary point depends on which video mode is active. For Spectrum and Hi-Colour mode, the binary point is placed between the two bytes with the most significant byte representing the integer part of the X or Y coordinate value. In Hi-Res mode, the binary point is placed between bits 6 and 7 for the X coordinate only, allowing the top 9 bits of the coordinate to represent a value in 0-511. The "sp_MouseAMXdx" / "dy" values represent the binary fixed point value added to the X and Y coordinates when the mouse is moved a distance of one unit in the positive direction. By default these values are set to 1 pixel using the correct fixed point representation for each video mode.

The simulated mouse simulates a mouse using any joystick including a keyboard joystick. A mouse is simulated in the sense that the mouse coordinates accelerate the longer a direction is asserted according to a user-supplied acceleration profile. The information on input device and acceleration profile are kept in a "struct sp_UDM". Here is a code fragment:

```
struct sp_UDK keys;
struct sp_MD accel[] = {
   {8, 0x0100, 0x0100},       /* 8 times, 1 pixel dx, 1 pixel
dy */
   {8, 0x0300, 0x0300},       /* 8 times, 3 pixel dx, 3 pixel
dy */
   {8, 0x0500, 0x0500},       /* 8 times, 5 pixel dx, 5 pixel
dy */
   {255, 0x0800, 0x0800}      /* Final delta must have a count
of 255 */
};
struct sp_UDM mouse;

main()
{
   /* define keys for simulated mouse */

   keys.up    = sp_LookupKey('q');
```

```
      keys.down  = sp_LookupKey('a');
      keys.left  = sp_LookupKey('o');
      keys.right = sp_LookupKey('p');
      keys.fire  = sp_LookupKey('m');

      /* set up simulated mouse to use keyboard */

      mouse.keys = &keys;
      mouse.joyfunc = sp_JoyKeyboard;
      mouse.delta = accel;

      /* read mouse location */

      sp_MouseSim(&mouse, &x, &y, &buttons);

      ...
   }
```

In this example, a mouse is being simulated using the keyboard joystick. Because the keyboard joystick is being used, a "struct sp_UDK" keys structure must be assigned into the "struct sp_UDM mouse.keys" member. If any other joystick was being used, this parameter is ignored. Next the "struct sp_UDM mouse.joyfunc" member is set to point at one of the joystick functions. The final initialization step is to assign a pointer to the acceleration profile to "struct sp_UDM mouse.delta". Have a look at the acceleration profile in the code snippet above to see how one is created. The profile can be any size as long as the final entry has a count of "255".

The mice also allow their positions to be set by the program. See the "sp_SetMousePos*" functions in "spritepack.h".

## SCREEN ADDRESS HELPERS

The Spectrum display file is organized in a non-obvious way. If you would like to draw directly to the screen, Sprite Pack provides several functions that can compute screen addresses corresponding to pixel and character coordinates and that can modify screen addresses to move in all four directions from a current position. The latter are a much quicker method to generate a screen address than directly recomputing from a new pixel or coordinate position.

First a reminder. Sprite Pack's sprite module (this includes background tiles and sprites) does not know about anything you might draw directly to the screen. If SP's sprite module decides it is going to redraw a particular character square it will overwrite anything you may have drawn to that character square. There are three ways to deal with this if you intend to make use of the sprite module and draw to the screen directly yourself. One is to configure SP before the SP library compiles to use a subset of the screen's full row range. The areas outside this range will not be touched by SP and you can safely draw directly into it without any unexpected trouble. A second way is to validate portions of the screen ("sp_Validate") containing your artwork prior to asking SP to redraw the screen ("sp_UpdateNow"). Validated areas on screen will not be touched by the updater. The final way is probably the most interesting and the most difficult. Rather than drawing directly on the screen, consider drawing onto background tiles instead. Through SP's Tile Array, the UDG definitions of up to 256 characters can be arranged in memory as a secondary display file. Your functions would then draw to the background tiles' definitions as if they were drawing to a display file and SP would update those background tiles on screen. Since there are only 256 background tile definitions, you could only uniquely cover 1/3 of the screen in this way. A future version of SP will introduce windowing, making it possible to do this for the full screen. Of course there is nothing that says you *have* to use SP's sprites or background tiles. Maybe you would prefer to draw everything directly to the display file yourself.

Outside the realm of Sprite Pack, Z88DK also provides several functions that directly draw on screen, including line draw and circle commands. SP adds a pattern flood filler as well (see MISCELLANEOUS). These screen address functions will help in developing more graphics functions that write directly to the screen.

```
    void *sp_GetCharAddr(uchar row, uchar col);
    void *sp_GetScrnAddr(uint xcoord, uchar ycoord, uchar *mask);
    void *sp_GetAttrAddr(void *scrnaddr);
```

"sp_GetCharAddr" returns the screen address corresponding to the top scan line of the character coordinate (row,col). Subsequent scan lines in the character cell are exactly 256 bytes apart. "sp_GetScrnAddr" returns the screen address corresponding to the given (x,y) pixel coordinate. Each screen address contains information for 8 pixels; the mask returned contains a byte with a single bit set representing the individual pixel. "sp_GetAttrAddr" returns the attribute address corresponding to a given screen address. The contents of the attribute address controls the colour of pixels on screen.

Several more functions modify a screen address in the fashion suggested by their names:

```
    void *sp_CharDown(void *scrnaddr);
    void *sp_CharLeft(void *scrnaddr);
    void *sp_CharRight(void *scrnaddr);
    void *sp_CharUp(void *scrnaddr);
    void *sp_PixelDown(void *scrnaddr);
    void *sp_PixelUp(void *scrnaddr);
    void *sp_PixelLeft(void *scrnaddr, uchar *mask);
    void *sp_PixelRight(void *scrnaddr, uchar *mask);
```

These functions will return with the carry flag set if the screen address moves out of the screen boundary. You can test whether the carry flag is set from C using Z88DK's "iferror(..)" statement.

Some kind of example will be inserted here later.

## RECTANGLE INTERSECTIONS

Sprite Pack supplies several functions for detecting intersections among rectangles, intervals and points. These are useful for detecting collisions between sprites or for determining where a pointer is pointing when the mouse button is pressed.

There are two kinds of rectangles in Sprite Pack: the "struct sp_Rect" and the "struct sp_LargeRect". The former uses 8-bit values to specify the rectangle while the latter uses 16-bit values. "sp_Rect" is used internally by the sprite module of SP for background tile areas and sprites. "sp_LargeRect" is used by the rectangle intersection functions here. The definition of "sp_LargeRect" looks like this:

```
    struct sp_LargeRect {
       uint top;            /* Interval #1 */
       uint bottom;
       uint left;           /* Interval #2 */
       uint right;
    };
```

The absolute coordinates of the rectangle defined by an "sp_LargeRect" are stored in the structure. The "top" and "bottom" members specify the Y coordinates of the top and bottom of the rectangle and the "left" and "right" members specify the X coordinates of the left and right sides of the rectangle. All points are inclusive. Rectangle coordinates wrap. For example if the top coordinate of a rectangle is at

65530 and the bottom coordinate of a rectangle is 10, the rectangle is understood to occupy vertical coordinates 65530 to 65535 and 0 to 10 inclusive, for a total height of 6+11=17 units.

Intervals are also understood by SP:

```
struct sp_Interval {     /* [x1,x2]    */
   uint x1;              /* left side  */
   uint x2;              /* right side */
};
```

An interval is a one-dimensional range of values denoted by a left side and a right side, inclusive. As with rectangles, intervals also wrap. A closer look at "sp_LargeRect" will reveal that a large rectangle is made up of two intervals, one for the X range and one for the Y range occupied by the rect.

Using these data structures, SP provides several functions for detecting intersections among rectangles, intervals and points:

```
int sp_IntRect(struct sp_Rect *r1, struct sp_Rect *r2, struct
sp_Rect *result);
int sp_IntLargeRect(struct sp_LargeRect *r1, struct
sp_LargeRect *r2, struct sp_LargeRect *result);
int sp_IntPtLargeRect(uint x, uint y, struct sp_LargeRect *r);
int sp_IntIntervals(struct sp_Interval *i1, struct sp_Interval
*i2, struct sp_Interval *result);
int sp_IntPtInterval(uint x, struct sp_Interval *i);
```

"sp_IntRect" is the 8-bit version of "sp_IntLargeRect" and can be used to operate on the sprite module's rectangles. No more time will be spent with it here.

"sp_IntLargeRect" intersects two large rectangles, with the resulting intersection stored in "result". The value returned is true (non-zero) if the rectangles do intersect (making "result" valid) and false (zero) if they do not intersect. An intersection of two rectangles with wrapping coordinates can in fact result in 0, 1, 2 or 4 disjoint rectangles. This function will return just one intersected rectangle as "result".

"sp_IntIntervals" instersects two intervals, with the resulting intersection stored in "result". The value returned is true (non-zero) if the intervals do intersect and false (zero) if they do not. With wrapping coordinates, an intersection of two intervals can result in 0, 1 or 2 disjoint intervals as the result. This function will only return one interval as "result".

"sp_IntPtLargeRect" returns true (non-zero) if the point is contained within the rectangle. Similarly, "sp_IntPtInterval" returns true if the point is contained within the interval.

This simple example creates several rectangles outlining clickable regions on the screen. A tap file is available.

```
SORRY EXAMPLE INCOMPLETE
```

Four clickable rectangles are created on screen. This information is stored in the variables "one", "two", "three" and "four". Each of these variables is a "struct assoc" which associates a large rectangle specified in pixel coordinates with a callback function. Z88DK is unable to prototype function pointers properly so the "function" member is declared void* type.

In main() the programs starts as usual by calling "sp_Initialize" to prepare the sprite module. Calls to the "Do_*" functions are made to print the labels. Following that a linked list is created. The linked list data type is explained thoroughly later in the tutorial. You can think of it as an ordered list of items. The "click" list is initially created empty. The four "struct assoc" variables defining the clickable rectangles on screen are added to the list.

In the infinite loop, we query the state of the Kempston mouse and store its location in the "Pt" structure and the current button state in "b". The following "if" statement tests whether the mouse button has changed state (has just been pressed or released). If it has just been pressed, the linked list "click" has its current pointer moved to the beginning. Every linked list has an invisible finger pointing at the current item in its list. This finger can traverse the list by moving forward or backward sequentially through the list. "sp_ListFirst" places this pointer at the start of the list.

The following "sp_ListSearch" function begins a search through the list from the current pointer. For each item in the list, the user supplied "match" function is called with two parameters. One parameter is the third parameter in the "sp_ListSearch" call itself. The other parameter is the item currently pointed at in the list. In this case "sp_ListSearch" calls the function "collide_test" with the first parameter "&Pt" and the second parameter equal to a list item (a "struct assoc *"). If you scroll back a bit you will find this "collide_test" function. "collide_test" checks whether the mouse point intersects with the rectangle in the current "struct assoc *" from the "click" list. If there is an intersection, it returns true (a non-zero value) and the carry flag set. "sp_ListSearch" uses this as a termination signal and will stop the search with the current pointer pointing at that "struct assoc". The value returned by "sp_ListSearch" is NULL (0) if "collide_test" never returns true for all items in the list or it points at a "struct assoc" for which the collide_test returned true.

If the point is found to intersect a clickable rectangle, the action function stored in the corresponding "struct assoc" is executed. The action functions "Do_One", "Do_Two", "Do_Three", "Do_Four" are all defined earlier in the listing.

Detecting collisions among sprites can be done in a similar manner. Each sprite's "struct sp_SS" begins with a "struct sp_Rect" that contains up-to-date information on the position and size of a sprite's character area. Collisions between sprites can be checked by checking for collisions between these covering rectangles. Once a collision on this gross scale is found, each sprite can be further divided into many smaller rectangles that can be searched for a finer collision.

## DYNAMIC BLOCK MEMORY ALLOCATOR

Memory maps and memory allocation are probably new to you if your only programming background is BASIC. Everything that a program needs to remember including variables, the program itself, music and graphics must be stored in memory. BASIC tries its best to hide all these details from you. When moving up from BASIC, knowing where things are and what memory is available becomes important.

The Z80 CPU is an 8-bit CPU with a 16-bit address bus. This means it can directly see 2^16=64K memory locations that can each contain a single byte (8 bits). These memory locations are described by an address from 0 to 65535. An 8-bit value can potentially be stored at any of those addresses. In the 48K Spectrum, this address space is partitioned into a 16K ROM containing the BASIC interpretter in addresses 0-16383 and 48K of RAM in 16384-65535. ROM is read only memory where an 8-bit value has been permanently stored by the manufacturer. RAM is memory that you can change. The Spectrum's ULA further uses 16384-23295 to store display information. The BASIC system uses memory following 23295 to store its variables (the system variables and stream info) and the BASIC program. At the top of memory, the BASIC system reserves memory for the UDG characters and the Z80's machine stack. Somewhere following the BASIC program, the BASIC interpretter maintains a region of memory that contains variables defined in the BASIC program. As new variables are declared, this region expands. Somewhere nearby this BASIC variable region, the GOSUB stack is maintained. As GOSUB and RETURN commands are executed, memory is used to store and retrieve BASIC line numbers for where RETURNs should return to. In short, the BASIC interpretter is doing a lot of fiddling with memory behind the scenes. I have been a little bit vague about the exact locations of many of BASIC's data structures for a couple of reasons. One is that they move around depending on what your BASIC program is doing but the main reason is that it doesn't matter! We're not programming in BASIC anymore so we can take the whole lot of memory and use it for our own purposes (on the other hand, if you are expecting to write a C program that interacts with BASIC, then you should care about where these things are in memory and arrange to have your C program stay out of the way). If you are interested in visualizing the Basic system in action, do check out the Win32 tool "Basin" by Dunny. He has done a fantastic job of presenting a visual interface to the Basic system.

As a C or assembly language programmer we care about four things:

1) ROM occupies 0-16383 and is therefore unusable by our programs.
2) The display occupies 16384-23295 (if using the Timex hi-colour or hi-res video modes, 16384-22527 and 24576-30719; if using Timex dual screen, 16384-23295 and 24576-31487).
3) The memory from 16384-32767 is contended (it's slower than usual).
4) Our C (or assembly) program needs to be launched from BASIC. At a minimum this means a BASIC loader and a RAND USR to start the program. The C program will exist in a CODE file that must be loaded from tape into an area of memory that doesn't kill the small BASIC loader program.

By default, Z88DK compiles C programs starting at 32768. This is well out of the way of a small BASIC program. If memory is at a premium, you should get Z88DK to compile your program at an earlier address.

Sprite Pack will reserve areas of memory to hold its own variables and tables. These areas can be customized by editing the "SPconfig.def" file and then recompiling the Sprite Pack library. By default:

0xf200 - 0xffff (61952 - 65535)
Horizontal rotation table. Constructed when "sp_Initialize()" is called. Not used if sprites and background tiles are not used by your program. Change the location by modifying the 'SProtatetbl' constant in "SPconfig.def" prior to recompiling the Sprite Pack library.

0xf000 - 0xf100 (61440 - 61696)
IM 2 interrupt vector table. Constructed when "sp_InitIM2()" is called. Not used if IM2 functions are not used by your program. Change the location by modifying the 'IM2TABLE' constant in "SPconfig.def" prior to recompiling the Sprite Pack library.

The rest of SP's variables are compiled right into your program's binary as static data when needed. The largest ones include:

- The display list (128 bytes per row of displayed area -- 256 bytes per row in hi-res mode) if using sprites or background tiles.
- The dirty chars array (4 bytes per row of displayed area -- 8 bytes per row in hi-res mode and Timex dual-screen mode and up to an additional 255 bytes in dual-screen mode) if using sprites or background tiles.
- The key translation table (160 bytes) if using "sp_GetKey", "sp_LookupKey" or "sp_Inkey".
- The tile array (512 bytes -- 1024 bytes in hi-colour mode) if using sprites or background tiles.

The display area used by SP's sprite and background tile functions can be a subset of the Spectrum's full row range. The usable row range is set with the constants 'SP_ROWSTART' and 'SP_ROWEND' in the "SPconfig.def" file prior to recompiling the Sprite Pack library. By default the full 24 row display is used. This means a Spectrum display mode program using sprites or background tiles has: 128*24 + 4*24 + 512 = 3680 bytes of table data compiled into the final binary before you even write the first line of code! Thankfully this penalty is only paid once. You will find that small programs written using SP and Z88DK can have binaries typically around 9K in size. As your programs grow, the resulting binary size will grow much more slowly because most of the required machine code subroutines from libraries have already been attached to your binary and the only thing causing binary growth will be your own code.

The last item to worry about is the location of the Z80's stack. The BASIC system places this near the top of memory, which overlaps with SP's horizontal rotation tables. The "#pragma output STACKPTR=61440" directive that you have seen in many of the previous examples moves the top of the Z80 stack to 61440 (0xf000) which is just below the IM2 vector table. It is important to realize that the Z80 stack grows down in memory as registers are pushed and subroutines are called in a machine code program. You must make sure there is enough memory for the stack to grow downward to its maximum size. What is this maximum size? It depends on the program. 256 bytes might be enough, but I'd feel better about 512 bytes for most programs. If your program performs some deep recursive calls you may need more space. An example is SP's "sp_PFill" pattern fill function. It needs around 900 bytes of stack space to fill an arbitrary area on screen (a parameter of the function lets you control the amount of stack space it uses).

So the suggested memory configuration of an SP program looks like this:

```
00000 - 16383  BASIC ROM
16384 - 23295  Spectrum Mode Video Display (see above for other modes)
23296 - 32767  Free Contended (Slow) RAM;
               - Some is used by a BASIC loader when the program is
loaded
               - If you don't replace the BASIC ISR, some system
variables in
                 this area will be updated constantly.
               - Z88DK's stdin functions want to readd keypresses from
                 system variable LAST_K at 23560.
32768 -    up  Your program binary.  By default Z88DK assembles
programs to start here.
60928 - 61439  512-byte Z80 Stack (#pragma output STACKPTR=61440).
61440 - 61696  IM 2 Interrupt Vector Table (optional)
61697 - 61936  Free (240 bytes)
61937 - 61948  Generic Interrupt Service Routine @ 0xf1f1 (optional)
61949 - 61951  Free (3 bytes)
61952 - 65535  Horizontal Rotation Tables (optional)
```

Variables that a C program declares can be static or local. Static variables are your global variables, declared outside the scope of any function declaration. These variables are accessible all the time from within any function (subject to file scoping rules of course -- ignore if this makes no sense to you). Space for their storage is built into the program's final binary. Local variables are declared within a function and only exist while the function executes. Once the function returns, local variables are destroyed. These variables are declared on the Z80 stack and are popped off when the owning function completes. The area reserved for the Z80 stack (512 bytes in the suggested memory configuration above) must be large enough to hold all these variables in the most deeply nested sequence of calls that you make.

Before long you will find that you need to create some variable in memory that must be shared globally and must exist beyond the lifetime of a single function. That rules out a local variable. You may not now how big this variable will be or how many will be needed while writing the program. That rules out a global static variable. What is needed is a way to get some free memory, use it, then return it for use again later under program control. This is the memory allocation problem.

The standard C library provides "malloc" and "free" functions for allocating and freeing blocks of memory respectively. Where this memory comes from is a good question. Modern day systems have virtual 32-bit address spaces, meaning each individual program thinks it has a full 4 GB of memory to itself. No program in existence comes even close to using up that much space (large databases don't count) so the C runtime allocates free memory from the top of this virtual memory space as requested, without fear of overwriting even the most gargantuan of modern day programs. For small 64K memory systems, the issue of what parts of memory are available is a real problem. Z88DK's solution is that you must declare a global array in your program that will be used as "the heap" -- an area of memory available for allocation. This global array is then compiled as part of your program's binary. Do this by declaring the macro "HEAPSIZE(bp)" globally in your main C file and then call "heapinit(bp);" first thing from main() - example. "bp" is the free store's size in bytes and it is up to you to decide on an appropriate size depending on your program's needs. You can then use malloc and free as usual to allocate memory from this array. Don't forget to "include <malloc.h>" and link to the malloc library at compilation time.

This heap managed by C's standard malloc/free functions is able to allocate any size memory blocks as long as there is memory available. In order to do this there is a certain amount of overhead involved in managing the heap. Repeated allocation and freeing of multiply-sized memory blocks can also lead to what is known as fragmentation -- a situation where there are many tiny available pieces in the array, all of which add up to a large single piece but you are unable to allocate any memory block larger than the tiny pieces because they are scattered. Sometimes compaction is used to rearrange the tiny pieces into a single large piece but this requires a sophisticated memory manager that does not come with the standard C library. The overhead needed to manage the memory allocation can also make allocation and freeing a sluggish process.

Sprite Pack comes with an alternative memory manager that can be used in place of or in concert with the standard C library's malloc functions. It is a dynamic block memory allocator which means it allocates fixed-size memory blocks (rather than any ol' size) on demand. The fixed size scheme reduces overhead tremendously and means the allocator can be fast and will not suffer from the external fragmentation problem described above. A strong advantage it has is that you can add memory blocks to the allocator from *anywhere*. If you have free memory from 29000 to 29657, 35647 to 37000 and 46789 to 50000, you can add this memory for use by the allocator. Because you can only request memory in fixed-size pieces this allocator suffers from a different kind of fragmentation - internal fragmentation. This occurs when your program requests a block of memory which doesn't match any of the blocks' sizes. The allocator must give a block that is larger than the block requested, resulting in some wasted space.

The dynamic block memory allocator maintains a single queue for each size memory block. By default, SP is compiled to maintain 5 memory queues for 5 different size memory blocks (modify this by changing 'NUMQUEUES' in "SPconfig.def" prior to recompiling the SP library). Initially these queues contain nothing. You must add available memory to these queues and in the process decide what size memory blocks each queue should contain. Here is a sample code fragment that adds memory to three of the queues.

```
void *addr;

addr = sp_AddMemory(0, 100, 6, 45000);
addr = sp_AddMemory(1, 50, 14, addr);
sp_AddMemory(2, 25, 20, addr);
```

The first "sp_AddMemory" call adds one hundred 6-byte blocks to queue #0 using available memory at 45000. Each block suffers an overhead of one byte so this memory reservation will use addresses 45000 to 45000+(6+1)*100-1=45699 inclusive. The next free memory address is 45700, which is returned by the "sp_AddMemory" call. The next call adds fifty 14-byte blocks to queue #1 starting at 45700. The final call adds twenty-five 20-byte blocks to queue #2.

To retrieve a memory block from a specific queue, make a call to "sp_BlockAlloc":

```
addr = sp_BlockAlloc(1);
```

Here, "addr" will point at a free memory block 14 bytes in size (since it came from queue #1). If none are available, 0 (NULL) will be returned. If there is a danger of running out of memory blocks, your program should always check if the pointer returned is NULL.

Once you are finished with using the memory block, make it available for use later by freeing it:

```
    sp_FreeBlock(addr);
```

"sp_FreeBlock" is smart enough to know which queue a block belongs to without your telling it as long as the memory pointed at was allocated by the dynamic block memory allocator in the first place.

Sometimes you will want a block that is at least some minimum size:

```
    addr = sp_BlockFit(0, 3);  /* block best fit */
```

This function works as long as the queues contain increasingly sized blocks as queue #s increase, as it does in this example. This call will try to return a memory block from one of 3 memory queues starting with queue #0. If queue #0 has none available, queue #1 is checked. If queue #1 has none available, queue #2 is checked. If queue #2 has none available, NULL is returned. The function returns when it can allocate a memory block from one of the queues. In this case, since queue #0 is the first one checked, this function call will return a block at least 6 bytes in size and possibly 14 or 20 bytes in size.

If you ever want to know how many blocks are available in a particular queue, call "sp_BlockCount". "sp_InitAlloc" will clear all queues to empty. You do not have to call this function to initialize the allocator.

Many of SP's functions allocate memory behind the scenes. For example, the creation of a sprite requires the allocation of R*C+1 fourteen byte blocks where R=row size of the sprite in characters and C=column size of the sprite in characters. Each item added to a linked list managed by the linked list data type involves the allocation of a 6-byte node descriptor. This memory allocation occurs without your program's explicitly calling for it to happen. This behind-the-scenes memory allocation is performed through the functions pointed at by "void *u_malloc" and "void *u_free". These function pointers must be declared by your program so that this behind-the-scenes memory allocation can be performed.

The full ANSI prototype for "u_malloc" is "void *(*u_malloc)(uint bytes);". IE- u_malloc is a function that takes an unsigned integer indicating the number of bytes requested and returns a memory address pointing at the free memory or NULL if no memory is available. If you would like to use the standard C malloc function for memory allocation within SP, simply declare "void *u_malloc=malloc;" globally in your main.c file. Several earlier program examples showed how to use the dynamic block memory allocator. You may want to use a mixture of the two by declaring your own "my_malloc" function pointed at by "u_malloc".

Freeing memory allocated behind-the-scenes is done through the function pointed at by "u_free" whose full ANSI declaration is "void (*u_free)(void *addr);". IE- u_free is a function that takes a memory address as parameter and frees the memory block pointed at by this addressfor reuse later. If this address is NULL do nothing. If using the standard C library, declare "void *u_free = free;". If using the dynamic block memory allocator, declare "void *u_free = sp_BlockFree;". Or you can declare your own function with "void *u_free = my_free;".

## ABSTRACT DATA TYPES

LINKED LIST
HASH TABLE
HEAP

Before object-oriented languages like C++ became mainstream, procedural languages like C, Pascal, Fortran and Basic were King. A language is procedural if it supports a programming style that is linear (ie, first print this character, then goto this line number, etc.) The key to writing large, robust, bug-free programs using a procedural language is to write it in a modular fashion. The program is broken up into many small, manageable pieces and you would write these pieces almost independently of the others then forget about each piece once you were sure it was working. The abstract data type grew out of this modular programming paradigm and is really the same as what object-oriented languages would call a 'class'. In the old days 'modular programming' was very much the buzzword just as 'object-oriented programming' is today.

An abstract data type is a self-contained module that provides services to a programmer. It contains and manages its own private variables and exports an interface that the programmer can use. As the user of an abstract data type, you never need to look at the code that implements the data type and you don't care about the internal variables that the data type needs to do its tasks. You only care about the public interface provided by the data type and what the data type can do for you. In short, it's a way for programmers to share and reuse reliable and efficient code.

SP currently comes with three abstract data types: a linked list, a hash table and a heap. These are common data structures in programming circles with a general-purpose utility. Let's talk about each one in turn.

## LINKED LIST DATA TYPE

A linked list is a list of items. The list is called 'linked' because each item in the list is actually linked to a neighbour by a pointer that tells where the neighbour is. SP's linked list data type is a doubly-linked list. This means each item in the list has a pointer to the item following it and the item preceeding it (see "struct sp_ListNode" in the spritpack.h). Singly-linked lists (or just linked-list) contain items that only have pointers to the following item in the list.

You should think of SP's linked list data type as a maintainer of an ordered list of items. The list knows how many items it contains and possesses a 'current pointer' that points at one of the items in the list (it can also point before the start of the list and after the end of the list). You can add items to the end of the list, to the start of the list, just after the item pointed at by the current pointer, just before the item pointed at by the current pointer, etc. You can have a look at the item being pointed at, you can remove an item from the list or you can search the list. In short, you can do pretty much anything you would want to do with a list. Here is the public interface exported by the linked list data type:

**struct sp_List *sp_ListCreate(void);**
Creates a brand new empty list and returns a pointer to the list structure. Behind the scenes, SP will call "u_malloc" to get a block of memory large enough to create the "struct sp_List" returned by this call. If it is unsuccessful, NULL will be returned indicating insufficient memory and the list will not be created.

**uint sp_ListCount(struct sp_List *list);**
Returns how many items are in the list.

**int sp_ListAppend(struct sp_List *list, void *item);**
Adds 'item' to the end of the list. Behind the scenes, SP will call "u_malloc" to allocate a "struct sp_ListNode" to hold the item. If memory is not available for this, the item will not be added and zero will be returned to report failure. If successful, the list's current pointer will be updated to point at the new item.

**int sp_ListPrepend(struct sp_List *list, void *item);**
Adds 'item' to the beginning of the list. Behind the scenes, SP will call "u_malloc" to allocate a "struct sp_ListNode" to hold the item. If memory is not available for this, the item will not be added and zero will be returned to report failure. If successful, the list's current pointer will be updated to point at the new item.

**int sp_ListAdd(struct sp_List *list, void *item);**
Adds 'item' after the item pointed at by the list's invisible current pointer. Behind the scenes, SP will call

"u_malloc" to allocate a "struct sp_ListNode" to hold the item. If memory is not available for this, the item will not be added and zero will be returned to report failure. If successful, the list's current pointer will be updated to point at the new item.

**int sp_ListInsert(struct sp_List *list, void *item);**
Adds 'item' before the item pointed at by the list's invisible current pointer. Behind the scenes, SP will call "u_malloc" to allocate a "struct sp_ListNode" to hold the item. If memory is not available for this, the item will not be added and zero will be returned to report failure. If successful, the list's current pointer will be updated to point at the new item.

**void *sp_ListFirst(struct sp_List *list);**
Changes the current pointer to point at the first item in the list. Returns this first item or NULL if the list is empty.

**void *sp_ListLast(struct sp_List *list);**
Changes the current pointer to point at the last item in the list. Returns this last item or NULL if the list is empty.

**void *sp_ListNext(struct sp_List *list);**
Changes the current pointer to point at the next item in the list. Returns this next item or NULL if the current pointer has moved past the end of the list.

**void *sp_ListPrev(struct sp_List *list);**
Changes the current pointer to point at the previous item in the list. Returns this previous item or NULL if the current pointer has moved past the beginning of the list.

**void *sp_ListCurr(struct sp_List *list);**
Returns the item pointed at by the list's current pointer. Returns NULL if the list is empty or the current pointer points past the end of the list or before the start of the list.

**void *sp_ListRemove(struct sp_List *list);**
Removes the item pointed at by the current pointer. The just-removed item is returned and the current pointer is advanced to the next item in the list. If the current pointer wasn't pointing at anything (the list is empty or it points either past the end of the list or before the start of the list) NULL is returned.

**void *sp_ListTrim(struct sp_List *list);**
Removes the last item in the list and returns it. The current pointer points at the new last item. If the list was empty, NULL is returned.

**void sp_ListConcat(struct sp_List *list1, struct sp_List *list2);**
Concatenates list2 to the end of list1. List2 is destroyed in the process.

**void sp_ListFree(struct sp_List *list, void *free);**
Destroys the list and removes items from the list. "void (*free)(void *item)" is a user-supplied function that is called for each item in the list. It provides the user the opportunity to clean-up each item removed from the list. If free=NULL, no function is called.

**void *sp_ListSearch(struct sp_List *list, void *match, void *item1);**
Searches the list FROM THE LIST'S CURRENT POINTER (this is easy to forget; to search the entire list, call "sp_ListFirst()" first). The search is really a user-controlled iteration over the list's items. For each item visited, the user-supplied function "int (*match)(void *item1, void *item2)" is called with "item2" equal to the list item and "item1" equal to the parameter passed into the function. If the user-supplied function returns with carry set and non-zero, the list search function terminates with the list's current pointer pointing at "item2". The value returned is this "item2" or NULL if the user-supplied function never "matches." The PacMen Demo and the Minesweeper game on the examples page both use ListSearch to determine if the user has clicked on a pacman or a button, respectively.

## HASH TABLE DATA TYPE

A hash table is a fast table lookup. Some people call them dictionaries and some call them maps. What gets stored into a hash table is a (key,value) pair. When you want to find 'value', you provide 'key' and the hash table returns 'value'. It is very much like a dictionary: the key can be thought of as the word and the value is the word's meaning.

What's an alternative for implementing an English dictionary? You could store the words in alphabetical order in memory and have a pointer following each word point at a string stored elsewhere containing the word's meaning. Now suppose you want to look up the word "irresponsible". You would have to start at the beginning of this list of words and search until you found "irresponsible". This takes a lot of time, a time proportional to the number of words you have in your dictionary. This is denoted by "O(n)", big-Oh notation meaning "in time proportional to n" where in this case n=# words in the dictionary.

With a hash table, you could find the word immediately in O(1) time. How is that possible? We take the key, in this case the word "irresponsible" and apply some function to it (called the "hash function") that returns a table index. Stored in the hash table at this index is the pointer to the meaning of the word.

This works great until real life comes in. First of all, for this to work as I described, we would need one table entry for each word in our dictionary -- this takes a lot of space. Secondly, we would have to come up with a hash function that condensed each English word into a unique single integer index. That's not easy. What we do instead is have a hash table with some number of table indices -- the hash table's size. We have a hash function that maps keys to a table index from 0..hashtable size - 1. We accept that several keys (words in our case) may map to the same table index. Then we get a "table collision". There are several ways to deal with a table collision. SP implements a hash table with buckets, where each table entry is a linked list of keys. Once the key is mapped to the table index, the linked list at that index is searched for the key. The items in the linked list store (key,value) pairs.

For best performance, you want a hashfunction that distributes keys (words) evenly across all indices in the hash table. Making the hash table's size a prime number can help a great deal towared that goal. You will also find that good hashfunctions are data dependent -- a good hashfunction for a general word (ie English string) will not necessarily be a good hashfunction for polygon nodes, for example. You will also want to make the hash table's size big enough that there aren't more than a handful of (key,value) pairs stored in the linked list at each index.

That's a crash course on hash tables. Here is the hash table data type supplied by SP:

**struct sp_HashTable *sp_HashCreate(uint size, void *hashfunc, void *match, void *delete);**
Creates a hash table of size "size" using hash function "uint (*hashfunc)(void *key, uint size)". You must also provide "uchar (*match)(void *key1, void *key2)" which is a function that returns non-zero AND carry set if the two keys are equal. "void (*delete)(struct sp_HashCell *hc)" is called to delete items in the hash table. Your delete function is responsible for deleting the "hc->key" and "hc->value" items. All these functions are stored inside the "struct sp_HashTable" data structure so that you don't have to provide them as parameters to every single hash table function you call. Behind the scenes, SP will call "u_malloc" to get memory for a new "struct sp_HashTable". If unsuccessful, NULL is returned. SP will also call "u_malloc" to allocate "size*sizeof(struct sp_HashCell*)" bytes to create the hash table itself.

**void *sp_HashAdd(struct sp_HashTable *ht, void *key, void *value);**
Adds a (key,value) pair to the hash table. Behind the scenes, SP will call "u_malloc" to create a "struct sp_HashCell" to store the new pair. If insufficient memory is available, returns NULL and the carry flag set. If another (key,value) pair already exists in the table with the same key as the one being added, the new pair is added and a pointer to the old value is returned. Otherwise NULL is returned with the carry flag reset.

**struct sp_HashCell *sp_HashRemove(struct sp_HashTable *ht, void *key);**
Removes the (key,value) pair from the hash table if present. If the key was not found in the hash table, NULL is returned. Otherwise a "struct sp_HashCell*" is returned containing the (key,value) pair. The "struct sp_HashCell" was allocated using "u_malloc" -- you are responsible for freeing this memory by calling "u_free" when you are done with it.

**void *sp_HashLookup(struct sp_HashTable *ht, void *key);**
Looks up "key" and returns the associated "value". If the key was not found, NULL is returned.

**void sp_HashDelete(struct sp_HashTable *ht);**
Deletes the hash table, freeing any allocated memory used to create it. The user-supplied function "delete" (as provided during the hash table's creation) will be called for each "struct sp_HashCell" in the hash table. This user function is responsible for cleanup on the (key,value) pairs stored in the hash cell.

## HEAP DATA TYPE

A heap (sometimes called a priority queue) is an array whose contents are maintained in a special order that allows retrieving the smallest item in the array (a min heap) or the largest item in the array (a max heap) in O(1) time (ie immediately). Normal use of a heap involves adding items to it and then removing items from the top of the heap in increasing order (a min heap) or in decreasing order (a max heap). One application of a heap is "heapsort", a sorting algorithm that uses a heap to insert random items into and then to retrieve sorted items in order from it.

SP maintains a heap in an existing array of "void*" that you provide. The items in the array are "void*" so that you can make a heap of any type you want. A heap containing N items (again, you must keep track of N) is indexed using 1..N. This is unusual in C since arrays are normally indexed starting at 0. The use of 1..N simplifies the heap algorithm; the consequence is that either you make your array bigger by one item, leaving the very first item unused, or you supply the start address of the array one item earlier so that array[1] indexes the actual start of the array.

A very brief example:

```
void  *my_heap[50];        /* maximum 50 items in heap */
uchar  N = 1;              /* actual number of items in heap */

myheap[0] = first_item;    /* always have one item in the heap
*/
sp_HeapAdd(my_item, my_heap-1, N++, my_compare);
```

This adds "my_item" to the heap. I've supplied "my_heap-1" as the start of the heap array so that index 1 will actually map to the physical start of "my_heap". The number of items in the heap is passed in as "N" which is also automatically incremented afterward to record one more item in the heap. The "my_compare" function can compare items; this is explained later.

**void sp_Heapify(void **array, uint n, void *compare);**
Since the heap exists in an array that you declare, a heap does not have to be created in the same sense as the linked list and hash table had to be created. If you start with an empty array (n=0), you can simply add items one at a time using "sp_HeapAdd" and have the heap property maintained within the array. This function is for arrays that do not have the heap property: they can be any ordinary array that you have built up. This function will rearrange the items in the array to make it into a heap. The user-supplied "int (*compare)(void *item1, void *item2)" function compares two items in the heap and should return carry flag set AND non-zero if item1<item2 for a min-heap (one where you remove items in increasing order) or if item1>item2 for a max-heap (one where you remove items in decreasing order).

**void sp_HeapAdd(void *item, void **array, uint n, void *compare);**

Adds an item to the heap, making sure the array maintains the heap property. After this call N should be incremented. The "compare" function is described in "sp_Heapify()".

**void sp_HeapSiftDown(uint start, void **array, uint n, void *compare);**
**void sp_HeapSiftUp(uint start, void **array, void *compare);**
These two functions are internally used to maintain the heap property for arrays. You wouldn't normally call them directly.

**void sp_HeapExtract(void **array, uint n, void *compare);**
Removes an item from the heap, placing it into array[n]. N should be decreased after this call. The compare function is as described in "sp_Heapify()". If you are maintaining a min-heap, the item removed will be the smallest item in the heap. If you are maintaining a max-heap, the item removed will be the largest item in the heap.

## DATA COMPRESSION

Static Huffman Compressor / Decompressor.
Documentation is on the way.

## MISCELLANEOUS

**void *sp_SwapEndian(void *ptr);**
Returns memory address "ptr" with its endianness swapped. Eg: LSB becomes MSB and MSB becomes LSB.

**void sp_Swap(void *addr1, void *addr2, uint bytes);**
Copies "bytes" bytes from address "addr1" to "addr2" and from "addr2" to "addr1". This is an overlap-safe memory swap.

**int sp_PFill(uint xcoord, uchar ycoord, void *pattern, uint stackdepth);**
A pattern fill of an enclosed point on screen. "pattern" points at an 8-byte UDG pattern character. "stackdepth" is the allowed memory usage from the Z80 stack. The number of bytes used is 3*stackdepth +30. If the fill was successful (ie there was sufficient memory to complete it), returns non-zero.

**int sp_StackSpace(void *addr);**
Returns "SP-addr," that is the distance between the Z80 stack pointer and addr. Can be used to determine available stack space.

**uint sp_Random32(uint *hi);**
Returns a 32-bit random number using an algorithm published in "Numerical Recipes in C," courtesy of Nick Fleming. The seed is available as one of Sprite Pack's variables (copy it out of "spritepack.h").

**void sp_Border(uchar colour);**
Sets the border colour. In Timex hi-res mode, sets the paper colour of the entire screen. The border colour is stored in the SP variable "sp_BorderClr".

**uchar sp_inp(uint port);**
Returns the result of a read on 16-bit i/o port "port".

**void sp_outp(uint port, uchar value);**
Writes a single byte "value" to the 16-bit i/o port "port".