SkoolKit Documentation

Release 3.6

Richard Dymond

February 11, 2014

CHAPTER 1

What is SkoolKit?

SkoolKit is a collection of utilities that can be used to disassemble a Spectrum game (or indeed any piece of Spectrum software written in machine code) into a format known as a *skool* file. Then, from this *skool* file, you can use SkoolKit to create a browsable disassembly in HTML format, or a re-assemblable disassembly in ASM format. So the *skool* file is - from start to finish as you develop it by organising and annotating the code - the common 'source' for both the reader-friendly HTML version of the disassembly, and the developer- and assembler-friendly ASM version of the disassembly.

The latest stable release of SkoolKit can always be obtained from pyskool.ca; the latest development version can be found on GitHub.

1.1 Features

Besides disassembling a Spectrum game into a list of Z80 instructions, SkoolKit can also:

- Build PNG or GIF images from graphic data in the game snapshot (using the *#UDG*, *#UDGARRAY*, *#FONT* and *#SCR* macros)
- Create hyperlinks between routines and data blocks that refer to each other (by use of the *#R* macro in annotations, and automatically in the operands of `CALL` and `JP` instructions)
- Neatly render lists of bugs, trivia and POKEs on separate pages (using *[Bug:*:*]*, *[Fact:*:*]* and *[Poke:*:*]* sections in a *ref* file)
- Produce ASM files that include bugfixes declared in the *skool* file (with *@ofix*, *@bfix* and other ASM directives)
- Produce TAP files from assembled code (using *bin2tap.py*)

For a demonstration of SkoolKit's capabilities, take a look at the complete disassemblies of Skool Daze, Back to Skool and Contact Sam Cruise. The latest stable releases of the source *skool* files for these disassemblies can always be obtained from pyskool.ca; the latest development versions can be found on GitHub.

1.2 Licence

SkoolKit is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

See the file 'COPYING' (distributed with SkoolKit) for the full text of the licence.

CHAPTER 2

Installing and using SkoolKit

2.1 Requirements

SkoolKit requires Python 2.7 or 3.2+. If you're running Linux or one of the BSDs, you probably already have Python installed. If you're running Windows, you can get Python here.

2.2 Installation

There are various ways to install the latest stable release of SkoolKit:

- from the zip archive or tarball available at pyskool.ca
- from the DEB package or RPM package available at pyskool.ca
- from PyPI by using easy_install or pip
- from the SkoolKit PPA for Ubuntu

If you choose the zip archive or tarball, note that SkoolKit can be used wherever it is unpacked: it does not need to be installed in any particular location. However, if you would like to install SkoolKit as a Python package, you can do so by using the supplied `setup.py` script.

2.2.1 Windows

To install SkoolKit as a Python package on Windows, open a command prompt, change to the directory where SkoolKit was unpacked, and run the following command:

```
> setup.py install
```

This should install the SkoolKit command scripts in *C:\Python27\Scripts* (assuming you have installed Python in *C:\Python27*), which means you can run them from anywhere (assuming you have added *C:\Python27\Scripts* to the `Path` environment variable).

2.2.2 Linux/*BSD

To install SkoolKit as a Python package on Linux/*BSD, open a terminal window, change to the directory where SkoolKit was unpacked, and run the following command as root:

```
# ./setup.py install
```

This should install the SkoolKit command scripts in */usr/local/bin* (or some other suitable location in your `PATH`), which means you can run them from anywhere.

2.3 Linux/*BSD v. Windows command line

Throughout this documentation, commands that must be entered in a terminal window ('Command Prompt' in Windows) are shown on a line beginning with a dollar sign (`$`), like this:

```
$ some-script.py some arguments
```

On Windows, and on Linux/*BSD if SkoolKit has been installed as a Python package (see above), the commands may be entered exactly as they are shown. On Linux/*BSD, a dot-slash (`./`) prefix should be added to `some-script.py` if it is being run from the current working directory.

CHAPTER 3

Command reference

3.1 bin2tap.py

bin2tap.py converts a binary file produced by an assembler (see *Supported assemblers*) into a TAP file that can be loaded into an emulator. For example:

```
$ bin2tap.py game.bin
```

will create a file called *game.tap*. By default, the origin address (the address of the first byte of code or data), the start address (the first byte of code to run) and the stack pointer are set to 65536 minus the length of *game.bin*. These defaults can be changed by passing options to *bin2tap.py*. Run it with no arguments to see the list of available options:

```
usage: bin2tap.py [options] FILE.bin

Convert a binary snapshot file into a TAP file.

Options:
  -o ORG, --org ORG     Set the origin address (default: 65536 minus the
                        length of FILE.bin)
  -p STACK, --stack STACK
                        Set the stack pointer (default: ORG)
  -s START, --start START
                        Set the start address to JP to (default: ORG)
  -t TAPFILE, --tapfile TAPFILE
                        Set the TAP filename (default: FILE.tap)
  -V, --version         Show SkoolKit version number and exit
```

Note that the ROM tape loading routine at 1366 ($0556) and the load routine used by *bin2tap.py* together require 14 bytes for stack operations, and so STACK must be at least 16384+14=16398 ($400E). This means that if ORG is less than 16398, you should use the `-p` option to set the stack pointer to something appropriate. If the main data block (derived from *game.bin*) overlaps any of the last four bytes of the stack, *bin2tap.py* will replace those bytes with the values required by the tape loading routine for correct operation upon returning. Stack operations will overwrite the bytes in the address range STACK-14 to STACK-1 inclusive, so those addresses should not be used to store essential code or data.

Version	Changes
1.3.1	New
2.2.5	Added the `-p` option
3.4	Added the `-V` option and the long options

3.2 skool2asm.py

skool2asm.py converts a *skool* file into an ASM file that can be fed to an assembler (see *Supported assemblers*). For example:

```
$ skool2asm.py game.skool > game.asm
```

skool2asm.py supports many options; run it with no arguments to see a list:

```
usage: skool2asm.py [options] file

Convert a skool file into an ASM file, written to standard output. FILE may be
a regular file, or '-' for standard input.

Options:
  -c, --create-labels   Create default labels for unlabelled instructions
  -d, --crlf            Use CR+LF to end lines
  -D, --decimal         Write the disassembly in decimal
  -f N, --fixes N       Apply fixes:
                          N=0: None (default)
                          N=1: @ofix only
                          N=2: @ofix and @bfix
                          N=3: @ofix, @bfix and @rfix (implies -r)
  -H, --hex             Write the disassembly in hexadecimal
  -i N, --inst-width N  Set instruction field width (default=23)
  -l, --lower           Write the disassembly in lower case
  -p, --package-dir     Show path to skoolkit package directory and exit
  -q, --quiet           Be quiet
  -r, --rsub            Use relocatability substitutions too (@rsub) (implies
                        '-f 1')
  -s, --ssub            Use safe substitutions (@ssub)
  -t, --tabs            Use tab to indent instructions (default indentation is
                        2 spaces)
  -u, --upper           Write the disassembly in upper case
  -V, --version         Show SkoolKit version number and exit
  -w, --no-warnings     Suppress warnings
```

See *ASM modes and directives* for a description of the `@ssub` and `@rsub` substitution modes, and the `@ofix`, `@bfix` and `@rfix` bugfix modes.

Version	Changes
1.1	Added the `-c` option
2.1.1	Added the `-u`, `-D` and `-H` options
2.2.2	Added the ability to read a *skool* file from standard input
3.4	Added the `-V` and `-p` options and the long options

3.3 skool2ctl.py

skool2ctl.py converts a *skool* file into a *control file*. For example:

```
$ skool2ctl.py game.skool > game.ctl
```

In addition to block types and addresses, *game.ctl* will contain block titles, block descriptions, registers, mid-block comments, block end comments, sub-block types and addresses, instruction-level comments, and some *ASM directives*.

To list the options supported by *skool2ctl.py*, run it with no arguments:

```
usage: skool2ctl.py [options] FILE

Convert a skool file into a control file, written to standard output. FILE may
be a regular file, or '-' for standard input.

Options:
  -a, --no-asm-dirs  Do not write ASM directives
  -h, --hex          Write addresses in hexadecimal format
  -V, --version      Show SkoolKit version number and exit
  -w X, --write X    Write only these elements, where X is one or more of:
                       b = block types and addresses
                       t = block titles
                       d = block descriptions
                       r = registers
                       m = mid-block comments and block end comments
                       s = sub-block types and addresses
                       c = instruction-level comments
```

If you need to preserve any elements that control files do not support (such as data definition entries and ASM block directives), consider using *skool2sft.py* to create a skool file template instead.

Version	Changes
1.1	New
2.0.6	Added the `-h` option
2.2.2	Added the ability to read a *skool* file from standard input
2.4	Added the `-a` option and the ability to preserve some ASM directives
3.4	Added the `-V` option and the long options

3.4 skool2html.py

skool2html.py converts a *skool* file (and its associated *ref* files, if any exist) into a browsable disassembly in HTML format.

For example:

```
$ skool2html.py game.skool
```

will convert the file *game.skool* into a bunch of HTML files. If any files named *game*.ref* (e.g. *game.ref*, *game-bugs.ref*, *game-pokes.ref* and so on) also exist, they will be used to provide further information to the conversion process.

skool2html.py can operate directly on *ref* files, too. For example:

```
$ skool2html.py game.ref
```

In this case, the *skool* file declared in the *[Config]* section of *game.ref* will be used; if no *skool* file is declared in *game.ref*, *game.skool* will be used if it exists. In addition, any existing files besides *game.ref* that are named *game*.ref* (e.g. *game-bugs.ref*, *game-pokes.ref* and so on) will also be used.

If an input file's name ends with '.ref', it will be treated as a *ref* file; otherwise it will be treated as a *skool* file.

skool2html.py supports several options; run it with no arguments to see a list:

```
usage: skool2html.py [options] FILE [FILE...]

Convert skool files and ref files to HTML. FILE may be a regular file, or '-'
for standard input.
```

```
Options:
  -a, --asm-labels      Use ASM labels
  -c S/L, --config S/L  Add the line 'L' to the ref file section 'S'; this
                        option may be used multiple times
  -C, --create-labels   Create default labels for unlabelled instructions
  -d DIR, --output-dir DIR
                        Write files in this directory (default is '.')
  -D, --decimal         Write the disassembly in decimal
  -H, --hex             Write the disassembly in hexadecimal
  -j NAME, --join-css NAME
                        Concatenate CSS files into a single file with this name
  -l, --lower           Write the disassembly in lower case
  -o, --rebuild-images  Overwrite existing image files
  -p, --package-dir     Show path to skoolkit package directory and exit
  -P PAGES, --pages PAGES
                        Write only these custom pages (when '-w P' is
                        specified); PAGES should be a comma-separated list of
                        IDs of pages defined in [Page:*] sections in the ref
                        file(s)
  -q, --quiet           Be quiet
  -s, --search-dirs     Show the locations skool2html.py searches for resources
  -t, --time            Show timings
  -T THEME, --theme THEME
                        Use this CSS theme; this option may be used multiple
                        times
  -u, --upper           Write the disassembly in upper case
  -V, --version         Show SkoolKit version number and exit
  -w X, --write X       Write only these files, where X is one or more of:
                          B = Graphic glitches    m = Memory maps
                          b = Bugs                o = Other code
                          c = Changelog           P = Custom pages
                          d = Disassembly files   p = Pokes
                          G = Game status buffer  t = Trivia
                          g = Graphics            y = Glossary
                          i = Disassembly index
```

skool2html.py searches the following directories for *skool* files, *ref* files, CSS files, JavaScript files, font files, and files listed in the *[Resources]* section of the *ref* file:

- The directory that contains the *skool* or *ref* file named on the command line
- The current working directory
- *./resources*
- *~/.skoolkit*
- */usr/share/skoolkit*
- *$PACKAGE_DIR/resources*

where *$PACKAGE_DIR* is the directory in which the *skoolkit* package is installed (as shown by `skool2html.py -p`). When you need a reminder of these locations, run `skool2html.py -s`.

The `-T` option sets the CSS theme. For example, if *game.ref* specifies the CSS files to use thus:

```
[Game]
StyleSheet=skoolkit.css;game.css
```

then:

```
$ skool2html.py -T dark -T wide game.ref
```

will use the following CSS files, if they exist, in the order listed:

- *skoolkit.css*
- *skoolkit-dark.css*
- *skoolkit-wide.css*
- *game.css*
- *game-dark.css*
- *game-wide.css*

Version	Changes
1.4	Added the `-V` option
2.1	Added the `-o` and `-P` options
2.1.1	Added the `-l`, `-u`, `-D` and `-H` options
2.2	No longer writes the Skool Daze and Back to Skool disassemblies by default; added the `-d` option
2.2.2	Added the ability to read a *skool* file from standard input
2.3.1	Added support for reading multiple *ref* files per disassembly
3.0.2	No longer shows timings by default; added the `-t` option
3.1	Added the `-c` option
3.2	Added *~/.skoolkit* to the search path
3.3.2	Added *$PACKAGE_DIR/resources* to the search path; added the `-p` and `-T` options
3.4	Added the `-a` and `-C` options and the long options
3.5	Added support for multiple CSS themes
3.6	Added the `--join-css` and `--search-dirs` options

3.5 skool2sft.py

skool2sft.py converts a *skool* file into a *skool file template*. For example:

```
$ skool2sft.py game.skool > game.sft
```

To list the options supported by *skool2sft.py*, run it with no arguments:

```
usage: skool2sft.py [options] FILE

Convert a skool file into a skool file template, written to standard output.
FILE may be a regular file, or '-' for standard input.

Options:
  -h, --hex      Write addresses in hexadecimal format
  -V, --version  Show SkoolKit version number and exit
```

Version	Changes
2.4	New
3.4	Added the `-V` option and the long options

3.6 sna2skool.py

sna2skool.py converts a binary (raw memory) file or a SNA, SZX or Z80 snapshot into a *skool* file. For example:

```
$ sna2skool.py game.z80 > game.skool
```

Now *game.skool* can be converted into a browsable HTML disassembly using *skool2html.py*, or into an assembler-ready ASM file using *skool2asm.py*.

sna2skool.py supports several options; run it with no arguments to see a list:

```
usage: sna2skool.py [options] file

Convert a binary (raw memory) file or a SNA, SZX or Z80 snapshot into a skool
file.

Options:
  -c FILE, --ctl FILE   Use FILE as the control file
  -g FILE, --generate-ctl FILE
                        Generate a control file in FILE
  -h, --ctl-hex         Write hexadecimal addresses in the generated control
                        file
  -H, --skool-hex       Write hexadecimal addresses and operands in the
                        disassembly
  -l L, --defm-size L   Set the maximum number of characters per DEFM
                        statement to L (default=66)
  -L, --lower           Write the disassembly in lower case
  -m M, --defb-mod M    Group DEFB blocks by addresses that are divisible by M
  -M FILE, --map FILE   Use FILE as a code execution map when generating a
                        control file
  -n N, --defb-size N   Set the maximum number of bytes per DEFB statement to
                        N (default=8)
  -o ADDR, --org ADDR   Specify the origin address of a binary (.bin) file
                        (default: 65536 - length)
  -p PAGE, --page PAGE  Specify the page (0-7) of a 128K snapshot to map to
                        49152-65535
  -r, --no-erefs        Don't add comments that list entry point referrers
  -R, --erefs           Always add comments that list entry point referrers
  -s ADDR, --start ADDR
                        Specify the address at which to start disassembling
                        (default=16384)
  -t, --text            Show ASCII text in the comment fields
  -T FILE, --sft FILE   Use FILE as the skool file template
  -V, --version         Show SkoolKit version number and exit
  -z, --defb-zfill      Write bytes with leading zeroes in DEFB statements
```

If the input filename does not end with '.sna', '.szx' or '.z80', it is assumed to be a binary file.

By default, any *control file* or *skool file template* whose name (minus the '.ctl' or '.sft' suffix) matches the input filename (minus the '.bin', '.sna', '.szx' or '.z80' suffix, if any) will be used, if present.

The `-M` option may be used (in conjunction with the `-g` option) to specify a code execution map to use when generating a control file. The supported file formats are:

- Profiles created by the Fuse emulator
- Code execution logs created by the SpecEmu, Spud and Zero emulators
- Map files created by the Z80 emulator

If the file specified by the `-M` option is 8192 bytes long, it is assumed to be a Z80 map file; otherwise it is assumed to be in one of the other supported formats.

Version	Changes
1.0.4	Added the `-g` and `-s` options
1.0.5	Added the `-t` option
2.0	Added the `-n`, `-m` and `-z` options
2.0.1	Added the `-o`, `-r` and `-l` options, and the ability to read binary files
2.0.6	Added the `-h` option
2.1	Added the `-H` option
2.1.2	Added the `-L` option
2.4	Added the `-T` option
3.2	Added the `-p` option, and the ability to read SZX snapshots and 128K Z80 snapshots
3.3	Added the `-M` option, along with support for code execution maps produced by Fuse, SpecEmu, Spud, Zero and Z80; added the ability to read 128K SNA snapshots
3.4	Added the `-V` and `-R` options and the long options

3.7 tap2sna.py

tap2sna.py converts a TAP or TZX file (which may be inside a zip archive) into a Z80 snapshot. For example:

```
$ tap2sna.py game.tap game.z80
```

To list the options supported by *tap2sna.py*, run it with no arguments:

```
usage:
  tap2sna.py [options] INPUT snapshot.z80
  tap2sna.py @FILE

Convert a TAP or TZX file (which may be inside a zip archive) into a Z80
snapshot. INPUT may be the full URL to a remote zip archive or TAP/TZX file,
or the path to a local file. Arguments may be read from FILE instead of (or as
well as) being given on the command line.

Options:
  -d DIR, --output-dir DIR
                        Write the snapshot file in this directory.
  -f, --force           Overwrite an existing snapshot.
  --ram OPERATION       Perform a load, move or poke operation on the memory
                        snapshot being built. Do '--ram help' for more
                        information. This option may be used multiple times.
  --reg name=value      Set the value of a register. Do '--reg help' for more
                        information. This option may be used multiple times.
  --state name=value    Set a hardware state attribute. Do '--state help' for
                        more information. This option may be used multiple
                        times.
  -V, --version         Show SkoolKit version number and exit.
```

Note that support for TZX files is limited to block types 0x10 (Standard Speed Data Block) and 0x11 (Turbo Speed Data Block).

By default, *tap2sna.py* loads bytes from every data block on the tape, using the start address given in the corresponding header. For tapes that contain headerless data blocks, headers with incorrect start addresses, or irrelevant blocks, the `--ram` option can be used to load bytes from specific blocks at the appropriate addresses. For example:

```
$ tap2sna.py --ram load=3,30000 game.tzx game.z80
```

loads the third block on the tape at address 30000, and ignores all other blocks. The `--ram` option can also be used to move blocks of bytes from one location to another, and POKE values into individual addresses or address ranges

before the snapshot is saved. For more information on the operations that the `--ram` option can perform, run:

```
$ tap2sna.py --ram help
```

For complex snapshots that require many `--ram`, `--reg` or `--state` options to build, it may be more convenient to store the arguments to *tap2sna.py* in a file. For example, if the file *game.t2s* has the following contents:

```
;
; tap2sna.py file for GAME
;
http://example.com/pub/games/GAME.zip
game.z80
--ram load=4,32768         # Load the fourth block at 32768
--ram move=40960,512,43520 # Move 40960-41471 to 43520-44031
--reg pc=34816             # Start at 34816
--reg sp=32768             # Stack at 32768
--state iff=0              # Disable interrupts
```

then:

```
$ tap2sna.py @game.t2s
```

will create *game.z80* as if the arguments specified in *game.t2s* had been given on the command line.

Version	Changes
3.5	New

CHAPTER 4

Disassembly DIY

The following sections describe how to use SkoolKit to get started on your own Spectrum game disassembly.

4.1 Getting started

The first thing to do is select a Spectrum game to disassemble. For the purpose of this discussion, we'll use Manic Miner (the original Bug Byte version). To build a pristine snapshot of the game, run the following command in the directory where SkoolKit was unpacked:

```
$ tap2sna.py @examples/manic_miner.t2s
```

(If that doesn't work, or you prefer to make your own snapshot, just grab a copy of the game, load it in an emulator, and save a Z80 snapshot named *manic_miner.z80*.)

The next thing to do is create a *skool* file from this snapshot. Run the following command from the SkoolKit directory:

```
$ sna2skool.py manic_miner.z80 > manic_miner.skool
```

Note that the '.skool' file name suffix is merely a convention, not a requirement. In general, any suffix besides '.ref' (which is used by *skool2html.py* to identify *ref* files) will do. If you are fond of the traditional three-letter suffix, then perhaps '.sks' (for 'SkoolKit source') or '.kit' would be more to your liking. However, for the purpose of this particular tutorial, you should stick with '.skool'.

Now take a look at *manic_miner.skool*. As you can see, by default, *sna2skool.py* disassembles everything from 16384 to 65535, treating it all as code. Needless to say, this is not particularly useful - unless you have no idea where the code and data blocks are yet, and want to use this disassembly to find out.

Once you have figured out where the code and data blocks are, it would be handy if you could supply *sna2skool.py* with this information, so that it can disassemble the blocks accordingly. That is where the control file comes in.

4.2 The control file

In its most basic form, a control file contains a list of start addresses of code and data blocks. Each address is marked with a 'control directive', which is a single letter that indicates what the block contains: `c` for a code block, or `b` for a data block (for example). A control file may contain annotations too, which will be interpreted as routine titles, descriptions, instruction-level comments or whatever else depending on the control directive they accompany.

A control file for Manic Miner might start like this:

```
b 32768
b 33280 Miner Willy sprite data
b 33536
c 33792 The game has just loaded
b 33799
t 33816 'AIR'
...
```

This control file declares that there is:

- a data block at 32768
- a data block at 33280 which should be titled 'Miner Willy sprite data'
- a data block at 33536
- a code block (routine) at 33792 which should be titled 'The game has just loaded'
- a data block at 33799
- a text block at 33816 which should be titled 'AIR' (because that's what it contains)

For more information on control file directives and their syntax, see *Control files*.

4.3 A skeleton disassembly

So if we had a control file for Manic Miner, we could produce a much more useful *skool* file. As it happens, SkoolKit includes one: *manic_miner.ctl*. You can use it with *sna2skool.py* thus:

```
$ sna2skool.py -c examples/manic_miner.ctl manic_miner.z80 > manic_miner.skool
```

This time, *manic_miner.skool* is split up into meaningful blocks, with code as code, data as data (DEFBs), and text as text (DEFMs). Much nicer.

By default, *sna2skool.py* produces a disassembly with addresses and instruction operands in decimal notation. If you prefer to work in hexadecimal, however, use the `-H` option:

```
$ sna2skool.py -H -c examples/manic_miner.ctl manic_miner.z80 > manic_miner.skool
```

The next step is to create an HTML disassembly from this *skool* file:

```
$ skool2html.py manic_miner.skool
```

(Don't worry about the warnings that are printed.) Now open *manic_miner/index.html* in a web browser. There's not much there, but it's a base from which you can start adding explanatory comments.

In order to replace 'manic_miner' in the page titles and headers with something more appropriate, or add a game logo image, or otherwise customise the disassembly, we need to create a *ref* file. Again, as it happens, SkoolKit includes an example *ref* file for Manic Miner: *manic_miner.ref*. To use it with the *skool* file we've just created:

```
$ skool2html.py examples/manic_miner.ref
```

This time there should be no warnings printed, and the disassembly should sport a game logo image, and contain images of the caverns and the guardians that populate them.

See *Ref files* for more information on how to use a *ref* file to configure and customise a disassembly.

4.4 Generating a control file

If you are planning to create a disassembly of some game other than Manic Miner, you will need to create your own control file. To get started, you can use the `-g` option with *sna2skool.py* to perform a rudimentary static code analysis of the snapshot file and generate a corresponding control file:

```
$ sna2skool.py -g game.ctl game.z80 > game.skool
```

This will do a reasonable job of splitting the snapshot into blocks, but won't be 100% accurate (except by accident); you will need to examine the resultant *skool* file (*game.skool* in this case) to see which blocks have been incorrectly marked as text, data or code, and then edit the generated control file (*game.ctl*) accordingly.

To generate a better control file, you could use a code execution map produced by an emulator to tell *sna2skool.py* where at least some of the code is in the snapshot. *sna2skool.py* will read a map (otherwise known as a profile or trace) produced by Fuse, SpecEmu, Spud, Zero or Z80 when specified by the `-M` option:

```
$ sna2skool.py -M game.map -g game.ctl game.z80 > game.skool
```

Needless to say, in general, the better the map, the more accurate the resulting control file will be. To create a good map file, you should ideally play the game from start to finish in the emulator, in an attempt to exercise as much code as possible. If that sounds like too much work, and your emulator supports playing back RZX files, you could grab a recording of your chosen game from the RZX Archive, and set the emulator's profiler or tracer going while the recording plays back.

By default, *sna2skool.py* generates a control file and a *skool* file with addresses and instruction operands in decimal notation. If you prefer to work in hexadecimal, however, use the `-h` option to produce a hexadecimal control file, and the `-H` option to produce a hexadecimal *skool* file:

```
$ sna2skool.py -h -H -g game.ctl game.z80 > game.skool
```

4.5 Developing the skool file

When you're happy that your control file does a decent job of distinguishing the code blocks from the data blocks in your memory snapshot, it's time to start work on the *skool* file.

Figuring out what the code blocks do and what the data blocks contain can be a time-consuming job. It's probably not a good idea to go through each block one by one, in order, and move to the next only when it's fully documented - unless you're looking for a nervous breakdown. Instead it's better to approach the job like this:

1. Skim the code blocks for any code whose purpose is familiar or obvious, such as drawing something on the screen, or producing a sound effect.
2. Document that code (and any related data) as far as possible.
3. Find another code block that calls the code block just documented, and figure out when, why and how it uses it.
4. Document that code (and any related data) as far as possible.
5. If there's anything left to document, return to step 3.
6. Done!

It also goes without saying that figuring out what a piece of code or data might be used for is easier if you've played the game to death already.

Annotating the code and data in a *skool* file is done by adding comments just as you would in a regular ASM file. For example, you might add a comment to the instruction at 35136 in *manic_miner.skool* thus:

```
 35136 DEC (HL)      ; Decrement the number of lives
```

See the *skool file format* reference for a full description of the kinds of annotations that are supported in *skool* files. Note also that SkoolKit supports many *skool macros* that can be used in comments and will be converted into hyperlinks and images (for example) in the HTML version of the disassembly.

As you become more familiar with the layout of the code and data blocks in the disassembly, you may find that some blocks need to be split up, joined, or otherwise reorganised. You could do this manually in the *skool* file itself, or you could regenerate the *skool* file from a new control file. To ensure that you don't lose all the annotations you've already added to the *skool* file, though, you should use *skool2ctl.py* to preserve them.

First, create a control file that keeps your annotations intact:

```
$ skool2ctl.py game.skool > game-2.ctl
```

Now edit *game-2.ctl* to fit your better understanding of the layout of the code and data blocks. Then generate a new *skool* file:

```
$ sna2skool.py -c game-2.ctl game.z80 > game-2.skool
```

This new skool file, *game-2.skool*, should contain your reorganised code and data blocks, and all the annotations you carefully added to *game.skool*.

4.6 Adding pokes, bugs and trivia

Adding 'Pokes', 'Bugs', and 'Trivia' pages to a disassembly is done by adding `Poke`, `Bug`, and `Fact` sections to the *ref* file. For any such sections that are present, *skool2html.py* will add links to the disassembly index page.

For example, let's add a poke. Add the following lines to *manic_miner.ref*:

```
[Poke:infiniteLives:Infinite lives]
The following POKE gives Miner Willy infinite lives:

POKE 35136,0
```

Now run *skool2html.py* again:

```
$ skool2html.py examples/manic_miner.ref
```

Open *manic_miner/index.html* and you should see a link to the 'Pokes' page in the 'Reference' section.

The format of a `Bug` or `Fact` section is the same, except that the section name prefix is `Bug:` or `Fact:` (instead of `Poke:`) as appropriate.

One `Poke`, `Bug` or `Fact` section should be added for each poke, bug or trivia item to be documented. Entries will appear on the 'Pokes', 'Bugs' or 'Trivia' page in the same order as the sections appear in the *ref* file.

See *Ref files* for more information on the format of the `Poke`, `Bug`, and `Fact` (and other) sections that may appear in a *ref* file.

4.7 Themes

In addition to the default theme (defined in *skoolkit.css*), SkoolKit includes some alternative themes:

- dark (dark colours): *skoolkit-dark.css*
- green (mostly green): *skoolkit-green.css*

- plum (mostly purple): *skoolkit-plum.css*
- spectrum (Spectrum colours and font): *skoolkit-spectrum.css*
- wide (wide comment fields on the disassembly pages, and wide boxes on the Changelog, Glossary, Trivia, Bugs and Pokes pages): *skoolkit-wide.css*

In order to use a theme, run *skool2html.py* with the `-T` option; for example, to use the 'dark' theme:

```
$ skool2html.py -T dark game.skool
```

To use the 'spectrum' theme, the spectrum font file should also be specified thus:

```
$ skool2html.py -T spectrum -c Game/Font=spectrum.ttf game.skool
```

Themes may also be combined; for example, to use both the 'dark' and 'wide' themes:

```
$ skool2html.py -T dark -T wide game.skool
```

CHAPTER 5

Supported assemblers

If you want to use SkoolKit to generate an ASM version of your disassembly, you will need to use a supported assembler. At the time of writing, the assemblers listed below are known to work with the ASM format generated by *skool2asm.py*:

- pasmo (v0.5.3)
- SjASMPlus (v1.07-rc7)
- z80asm, the assembler distributed with z88dk (v1.8)

The following sections give examples of how to use each of these assemblers to create binary (raw memory) files or tape files that can be used with an emulator.

5.1 pasmo

First, create an ASM version of the disassembly:

```
$ skool2asm.py game.skool > game.asm
```

Then use *pasmo* to create a binary file thus:

```
$ pasmo game.asm game.bin
```

To create a TAP file from *game.bin*, use the *bin2tap.py* utility, included with SkoolKit:

```
$ bin2tap.py game.bin
```

The resultant TAP file, *game.tap*, can then be loaded into an emulator.

5.2 SjASMPlus

First, create an ASM version of the disassembly:

```
$ skool2asm.py game.skool > game.asm
```

Then create a file called *game.sjasm* with the following contents:

```
; SjASMPlus source file for game.asm
  device zxspectrum48
  include game.asm
  savebin "game.bin",ORG,LENGTH
```

replacing `ORG` and `LENGTH` with the origin address and the length of the assembled program. Now run *sjasmplus* on this source file:

```
$ sjasmplus game.sjasm
```

and a binary file called *game.bin* will be created.

To create a TAP file from *game.bin*, use the *bin2tap.py* utility, included with SkoolKit:

```
$ bin2tap.py game.bin
```

The resultant TAP file, *game.tap*, can then be loaded into an emulator.

5.3 z80asm (z88dk)

First, create an ASM version of the disassembly:

```
$ skool2asm.py game.skool > game.asm
```

Then use *z80asm* to create a binary file thus:

```
$ z80asm -rORG -b game.asm
```

replacing `ORG` with the origin address (in hexadecimal notation) of the program.

To create a TAP file from *game.bin*, use the *bin2tap.py* utility, included with SkoolKit:

```
$ bin2tap.py game.bin
```

The resultant TAP file, *game.tap*, can then be loaded into an emulator.

CHAPTER 6

General info

6.1 Contact details

To make complaints about or suggest improvements to SkoolKit, or to submit some other piece of constructive criticism, contact me (Richard Dymond) at <*rjdymond AT gmail.com*>.

6.2 Bugs

If you find any bugs in SkoolKit, please report them using the bug tracker.

CHAPTER 7

Changelog

7.1 3.6 (2013-11-02)

- Enhanced the *#UDGARRAY* macro so that it can create an animated image from an arbitrary sequence of frames
- Enhanced the *#FONT* macro so that it can create an image of arbitrary text
- Added support for copying arbitrary files into an HTML disassembly by using the *[Resources]* section in the *ref* file
- Added the `--join-css` option to *skool2html.py* (to concatenate CSS files into a single file)
- Added the `--search-dirs` option to *skool2html.py* (to show the locations that *skool2html.py* searches for resources)
- Added support for creating disassemblies with a start address below 10000
- Added an example control file for the 48K Spectrum ROM: *48.rom.ctl*
- Control files can now preserve blank comments that span two or more instructions
- The `[Config]` section no longer has to be in the *ref* file named on the *skool2html.py* command line; it can be in any secondary *ref* file
- Fixed the bug that makes *skool2html.py* fail if the `FontPath`, `JavaScriptPath` or `StyleSheetPath` parameter in the `[Paths]` section of the *ref* file is set to some directory other than the default

7.2 3.5 (2013-09-01)

- Added the *tap2sna.py* command (for building snapshots from TAP/TZX files)
- Added support to *skool2html.py* for multiple CSS themes
- Added the 'green', 'plum' and 'wide' CSS themes: *skoolkit-green.css*, *skoolkit-plum.css*, *skoolkit-wide.css*
- Moved the `Font` and `StyleSheet` parameters from the `[Paths]` section to the *[Game]* section
- Moved the `JavaScript` parameter from the `[Paths]` section to the *[Page:*]* section
- Moved the `Logo` parameter from the `[Paths]` section to the *[Game]* section and renamed it `LogoImage`
- The *#R* macro now renders the addresses of remote entries in the specified case and base, and can resolve the addresses of remote entry points
- *skool2asm.py* now writes ORG addresses in the specified case and base

- Annotated the source code remnants at 39936 in *jet_set_willy.ctl*

7.3 3.4 (2013-07-08)

- Dropped support for Python 2.6 and 3.1
- Added long options to every command
- Added the `--asm-labels` and `--create-labels` options to *skool2html.py* (to use ASM labels defined by *@label* directives, and to create default labels for unlabelled instructions)
- Added the `--erefs` option to *sna2skool.py* (to always add comments that list entry point referrers)
- Added the `--package-dir` option to *skool2asm.py* (to show the path to the skoolkit package directory)
- Added support for the `LinkOperands` parameter in the *[Game]* section of the *ref* file, which may be used to enable the address operands of LD instructions to be hyperlinked
- Added support for defining image colours by using hex triplets in the *[Colours]* section of the *ref* file
- Added support to the *@set* ASM directive for the *handle-unsupported-macros* and *wrap-column-width-min* properties
- Fixed the *#EREFS* and *#REFS* macros so that they work with hexadecimal address parameters
- Fixed the bug that crashes *sna2skool.py* when generating a control file from a code execution map and a snapshot with a code block that terminates at 65535
- Fixed how *skool2asm.py* renders table cells with rowspan > 1 and wrapped contents alongside cells with rowspan = 1
- Removed support for the `#NAME` macro (what it did can be done by the *#HTML* macro instead)
- Removed the documentation sources and man page sources from the SkoolKit distribution (they can be obtained from GitHub)

7.4 3.3.2 (2013-05-13)

- Added the `-T` option to *skool2html.py* (to specify a CSS theme)
- Added the `-p` option to *skool2html.py* (to show the path to the skoolkit package directory)
- *setup.py* now installs the *resources* directory (so a local copy is no longer required when SkoolKit has been installed via `setup.py install`)
- Added *jet_set_willy-dark.css* (to complete the 'dark' theme for that disassembly)
- Added *documentation* on how to write an instruction-level comment that contains opening or closing braces when rendered
- Fixed the appearance of transparent table cells in HTML output
- Fixed *sna2skool.py* so that a control file specified by the `-c` option takes precedence over a default skool file template
- Fixed *manic_miner.ctl* so that the comments at 40177-40191 apply to a pristine snapshot (before stack operations have corrupted those addresses)

7.5 3.3.1 (2013-03-04)

- Added support to the *@set* ASM directive for the *comment-width-min*, *indent*, *instruction-width*, *label-colons*, *line-width* and *warnings* properties
- Added support to the `HtmlWriterClass` parameter (in the *[Config]* section) and the *@writer* directive for specifying a module outside the module search path (e.g. a standalone module that is not part of an installed package)
- *sna2skool.py* now correctly renders an empty block description as a dot (`.`) on a line of its own

7.6 3.3 (2013-01-08)

- Added support to *sna2skool.py* for reading code execution maps produced by the Fuse, SpecEmu, Spud, Zero and Z80 emulators (to generate more accurate control files)
- Increased the speed at which *sna2skool.py* generates control files
- Added support to *sna2skool.py* for disassembling 128K SNA snapshots

7.7 3.2 (2012-11-01)

- Added support to *sna2skool.py* for disassembling 128K Z80 snapshots and 16K, 48K and 128K SZX snapshots
- Added the *#LIST* macro (for rendering lists of bulleted items in both HTML mode and ASM mode)
- Added the *@set* ASM directive (for setting properties on the ASM writer)
- Added trivia entries to *jet_set_willy.ref*
- Annotated the source code remnants at 32768 and 37708 in *manic_miner.ctl*

7.8 3.1.4 (2012-10-11)

- Added support to *skool2ctl.py* and *skool2sft.py* for DEFB and DEFM statements that contain both strings and bytes
- *skool2ctl.py* now correctly processes lower case DEFB, DEFM, DEFS and DEFW statements
- The length of a string (in a DEFB or DEFM statement) that contains one or more backslashes is now correctly calculated by *skool2ctl.py* and *skool2sft.py*
- DEFB and DEFM statements that contain both strings and bytes are now correctly converted to lower case, upper case, decimal or hexadecimal (when using the `-l`, `-u`, `-D` and `-H` options of *skool2asm.py* and *skool2html.py*)
- Operations involving (IX+n) or (IY+n) expressions are now correctly converted to lower case decimal or hexadecimal (when using the `-l`, `-D` and `-H` options of *skool2asm.py* and *skool2html.py*)

7.9 3.1.3 (2012-09-11)

- The 'Glossary' page is formatted in the same way as the 'Trivia', 'Bugs', 'Pokes' and 'Graphic glitches' pages
- When the link text of a *#LINK* macro is left blank, the link text of the page is substituted

- The disassembler escapes backslashes and double quotes in DEFM statements (so that *skool2asm.py* no longer has to)
- DEFB and DEFM statements that contain both strings and bytes are parsed correctly for the purpose of building a memory snapshot

7.10 3.1.2 (2012-08-01)

- Added the *#HTML* macro (for rendering arbitrary text in HTML mode only)
- Added support for distinguishing input values from output values in a routine's register section (by using prefixes such as 'Input:' and 'Output:')
- Added support for the `InputRegisterTableHeader` and `OutputRegisterTableHeader` parameters in the *[Game]* section of the *ref* file
- Added the 'default' CSS class for HTML tables created by the *#TABLE* macro

7.11 3.1.1 (2012-07-17)

- Enhanced the *#UDGARRAY* macro so that it accepts both horizontal and vertical steps in UDG address ranges
- Added support for the `Font` and `FontPath` parameters in the *[Paths]* section of the *ref* file (for specifying font files used by CSS @*font-face* rules)
- Added a Spectrum theme CSS file that uses the Spectrum font and colours: *skoolkit-spectrum.css*
- Fixed *skool2asm.py* so that it escapes backslashes and double quotes in DEFM statements

7.12 3.1 (2012-06-19)

- Dropped support for Python 2.5
- Added documentation on *extending SkoolKit*
- Added the @*writer* ASM directive (to specify the class to use for producing ASM output)
- Added the *#CHR* macro (for rendering arbitrary unicode characters); removed support for the redundant `#C` macro accordingly
- Added support for the *#CALL*, *#REFS*, *#EREFS*, *#PUSHS*, *#POKES* and *#POPS* macros in ASM mode
- Added the `-c` option to *skool2html.py* (to simulate adding lines to the *ref* file)
- Added a dark theme CSS file: *skoolkit-dark.css*

7.13 3.0.2 (2012-05-01)

- Added room images and descriptions to *manic_miner.ctl* and *jet_set_willy.ctl* (based on reference material from Andrew Broad and J. G. Harston)
- Fixed the bug that prevents the 'Data tables and buffers' section from appearing on the disassembly index page when the default `DataTables` link group is used

7.14 3.0.1 (2012-04-11)

- Added support for creating GIF files (including transparent and animated GIFs)
- Added support for creating animated PNGs in APNG format
- Added support for transparency in PNG images (by using the `PNGAlpha` parameter in the *[ImageWriter]* section of the *ref* file)
- Added an example control file: *jet_set_willy.ctl*
- Fixed the bug in how images are cropped by the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros when using non-zero `X` and `Y` parameters

7.15 3.0 (2012-03-20)

- SkoolKit now works with Python 3.x
- Added a native image creation library, which can be configured by using the *[ImageWriter]* section of the *ref* file; *gd* and *PIL* are no longer required or supported
- Enhanced the *#SCR* macro so that graphic data and attribute bytes in places other than the display file and attribute file may be used to build a screenshot
- Added image-cropping capabilities to the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros

7.16 Older versions

7.16.1 SkoolKit 2.x changelog

2.5 (2012-02-22)

- Added support for *[MemoryMap:*]* sections in *ref* files (for defining the properties of memory map pages); removed support for the `[MapDetails]` section accordingly
- Added support for multiple style sheets per HTML disassembly (by separating file names with a semicolon in the `StyleSheet` parameter in the *[Paths]* section of the *ref* file)
- Added support for multiple JavaScript files per HTML disassembly (by separating file names with a semicolon in the `JavaScript` parameter in the *[Paths]* section of the *ref* file)

2.4.1 (2012-01-30)

- The *@ignoreua* directive can now be used on entry titles, entry descriptions, mid-block comments and block end comments in addition to instruction-level comments; the *@ignoredua* and *@ignoremrcua* directives are correspondingly deprecated
- The *#SPACE* macro now supports the syntax `#SPACE([num])`, which can be useful to distinguish it from adjacent text where necessary

2.4 (2012-01-10)

- Added the *skool2sft.py* command (for creating *skool file templates*)
- Added support to *skool2ctl.py* for preserving some ASM directives in control files
- Enhanced the *#UDG* and *#UDGARRAY* macros so that images can be rotated
- Added the ability to separate paragraphs in a *skool* file by using a dot (`.`) on a line of its own; removed support for the redundant `#P` macro accordingly

2.3.1 (2011-11-15)

- Added support to *skool2html.py* for multiple *ref* files per disassembly
- Enhanced the *#UDG* and *#UDGARRAY* macros so that images can be flipped horizontally and vertically
- Enhanced the *#POKES* macro so that multiple pokes may be specified
- Added support for the *#FACT* and *#POKE* macros in ASM mode
- When the link text of a *#BUG*, *#FACT* or *#POKE* macro is left blank, the title of the corresponding bug, trivia or poke entry is substituted
- Fixed the parsing of link text in skool macros in ASM mode so that nested parentheses are handled correctly
- Fixed the rendering of table borders in ASM mode where cells with rowspan > 1 in columns other than the first extend to the bottom row

2.3 (2011-10-31)

- Fixed the bug where the operands in substitute instructions defined by `@bfix`, `@ofix`, `@isub`, `@ssub` and `@rsub` directives are not converted to decimal or hexadecimal when using the `-D` or `-H` option of *skool2asm.py* or *skool2html.py*
- Removed the source files for the Skool Daze, Back to Skool and Contact Sam Cruise disassemblies from the SkoolKit distribution; they are now available as separate downloads

2.2.5 (2011-10-17)

- Enhanced the *#UDGARRAY* macro so that masks can be specified
- Added the `-p` option to *bin2tap.py* (to set the stack pointer)
- Fixed the parsing of link text in `#BUG`, `#FACT`, `#POKE` and other skool macros so that nested parentheses are handled correctly
- Fixed the handling of version 1 Z80 snapshots by *sna2skool.py*
- Added support for the `IndexPageId` and `Link` parameters in `[OtherCode:*]` sections of the *ref* file
- Reintroduced support for `[Changelog:*]` sections in *ref* files
- Added 'Changelog' pages to the Skool Daze, Back to Skool and Contact Sam Cruise disassemblies
- Updated the Contact Sam Cruise disassembly

2.2.4 (2011-08-10)

- Added support for the *@ignoredua* ASM directive
- *skool2asm.py* automatically decreases the width of the comment field for a 'wide' instruction instead of printing a warning
- *bin2tap.py* can handle binary snapshot files with ORG addresses as low as 16398
- Fixed the bug in *bin2tap.py* that prevents the START address from defaulting to the ORG address when the ORG address is specified with the `-o` option
- Added ASM directives to *csc.skool* so that it works with *skool2asm.py*
- Updated the Contact Sam Cruise disassembly

2.2.3 (2011-07-15)

Updated the Contact Sam Cruise disassembly; it is now 'complete'.

2.2.2 (2011-06-02)

- Added support for the *@end* ASM directive
- Added ASM directives to *{bts,csc,sd}-{load,save,start}.skool* to make them work with *skool2asm.py*
- *skool2asm.py*, *skool2ctl.py* and *skool2html.py* can read from standard input
- Fixed the bug that made *sna2skool.py* generate a control file with a code block at 65535 for a snapshot that ends with a sequence of zeroes
- Unit test *test_skool2html.py:Skool2HtmlTest.test_html* now works without an internet connection

2.2.1 (2011-05-24)

- SkoolKit can now be installed as a Python package using `setup.py install`
- Unit tests are included in the *tests* directory
- Man pages for SkoolKit's *command scripts* are included in the *man* directory
- Added 'Developer reference' documentation
- Fixed the bugs that made *skool2html.py* produce invalid XHTML

2.2 (2011-05-10)

- Changed the syntax of the *skool2html.py* command (it no longer writes the Skool Daze and Back to Skool disassemblies by default)
- Fixed the bug that prevented *skool2asm.py* from working with a zero-argument skool macro (such as `#C`) at the end of a sentence
- Fixed the `-w` option of *skool2asm.py* (it really does suppress all warnings now)
- Fixed how *sna2skool.py* handles `#P` macros (it now writes a newline before and after each one)
- Fixed the bug that made *sna2skool.py* omit the '*' control directive from routine entry points when the `-L` option was used

- ASM labels are now unaffected by the `-l` (lower case) and `-u` (upper case) options of *skool2asm.py*
- Added support for the '*' notation in statement length lists in sub-block control directives (e.g. `B 32768,239,16*14,15`)
- Updated the Skool Daze disassembly
- Updated the Back to Skool disassembly

2.1.2 (2011-04-28)

- Added the `-L` option to *sna2skool.py* (to write the disassembly in lower case)
- Added the `-i` option to *skool2html.py* (to specify the image library to use)
- In control files, DEFM, DEFW and DEFS statement lengths in `T`, `W` and `Z` sub-blocks may be declared
- Fixed the bug in *skool2asm.py*'s handling of the `#SPACE` macro that prevented it from working with *csc.skool*
- Fixed the bug that made *skool2asm.py* produce invalid output when run on *sd.skool* with the `-H` and `-f3` options

2.1.1 (2011-04-16)

- Added the `-l`, `-u`, `-D` and `-H` options to *skool2html.py* (to write the disassembly in lower case, upper case, decimal or hexadecimal)
- Added the `-u`, `-D` and `-H` options to *skool2asm.py* (to write the disassembly in upper case, decimal or hexadecimal)
- In control files, an instruction-level comment that spans a group of two or more sub-blocks of different types may be declared with an `M` directive
- Updated the incomplete Contact Sam Cruise disassembly

2.1 (2011-04-03)

- Added support for hexadecimal disassemblies
- Added the *#LINK* macro (for creating hyperlinks to other pages in an HTML disassembly)
- Added the ability to define custom pages in an HTML disassembly using `[Page:*]` and `[PageContent:*]` sections in the *ref* file
- Added the `-o` option to *skool2html.py* (to overwrite existing image files)
- Optional parameters in any position in a skool macro may be left blank
- In control files, DEFB statement lengths in multi-line `B` sub-blocks may be declared
- Updated the Skool Daze disassembly
- Updated the Back to Skool disassembly
- Updated the incomplete Contact Sam Cruise disassembly

2.0.6 (2011-03-09)

- *sna2skool.py* can read and write hexadecimal numbers in a control file
- *skool2ctl.py* can write hexadecimal numbers in a control file
- *sna2skool.py* no longer chokes on blank lines in a control file
- Updated the incomplete Contact Sam Cruise disassembly

2.0.5 (2011-02-09)

- Added the *#UDGARRAY* macro (for creating images of blocks of UDGs)
- Enhanced the *#FONT* macro so that it works with regular 8x8 fonts as well as the Skool game fonts
- Enhanced the *#SCR* macro so that it can take screenshots of rectangular portions of the screen
- The contents of the 'Other graphics' page of a disassembly are now defined in the `[Graphics]` section of the *ref* file
- Added the ability to define the layout of the disassembly index page in the `[Index]` and `[Index:*:*]` sections of the *ref* file
- Added the ability to define page titles in the `[Titles]` section of the *ref* file
- Added the ability to define page link text in the `[Links]` section of the *ref* file
- Added the ability to define the image colour palette in the `[Colours]` section of the *ref* file
- Fixed the bug in *sna2skool.py* that prevented it from generating a control file for a snapshot with the final byte of a 'RET', 'JR d', or 'JP nn' instruction at 65535
- Updated the incomplete Contact Sam Cruise disassembly

2.0.4 (2010-12-16)

Updated the incomplete Contact Sam Cruise disassembly.

2.0.3 (2010-12-08)

Updated the incomplete Contact Sam Cruise disassembly.

2.0.2 (2010-12-01)

- Fixed the `#EREFS`, `#REFS` and `#TAPS` macros
- Fixed the bug where the end comment for the last entry in a *skool* file is not parsed
- Updated the incomplete Contact Sam Cruise disassembly

2.0.1 (2010-11-28)

- Added the `-r` option to *skool2html.py* (for specifying a *ref* file)
- Added the `-o`, `-r`, and `-l` options to *sna2skool.py*, along with the ability to read binary (raw memory) files

- Fixed *skool2ctl.py* so that it correctly creates sub-blocks for commentless DEF{B,M,S,W} statements, and writes the length of a sub-block that is followed by a mid-routine comment
- Updated the incomplete Contact Sam Cruise disassembly

2.0 (2010-11-23)

- Updated the Back to Skool disassembly
- Enhanced the *#R* macro to support 'other code' disassemblies, thus making the `#ASM`, `#LOAD`, `#SAVE` and `#START` macros obsolete
- Split *load.skool*, *save.skool* and *start.skool* into separate files for each Skool game
- Added documentation on the *ref file sections*
- Simplified SkoolKit by removing all instances of and support for ref file macros and skool directives
- Added files that were missing from SkoolKit 1.4: *csc-load.skool*, *csc-save.skool* and *csc-start.skool*

7.16.2 SkoolKit 1.x changelog

1.4 (2010-11-11)

- Updated the Skool Daze disassembly
- Updated the Back to Skool disassembly
- Updated the incomplete Contact Sam Cruise disassembly

1.3.1 (2010-10-18)

- Added documentation on *supported assemblers*
- Added the *bin2tap.py* utility
- Documentation sources included in *docs-src*
- When running *skool2asm.py* or *skool2html.py* on Linux/BSD, show elapsed time instead of CPU time

1.3 (2010-07-23)

- Updated the Skool Daze disassembly
- Updated the Back to Skool disassembly
- Updated the incomplete Contact Sam Cruise disassembly

1.2 (2010-05-03)

Updated the Back to Skool disassembly.

1.1 (2010-02-25)

- Updated the Skool Daze disassembly
- Updated the Back to Skool disassembly
- Updated *contact_sam_cruise.ctl*
- Added *csc.ref* (to supply extra information to the Contact Sam Cruise disassembly)
- Added the *skool2ctl.py* utility

1.0.7 (2010-02-12)

- Extended the control file syntax to support block titles, descriptions, registers and comments, and sub-block types and comments
- Added two example control files: *contact_sam_cruise.ctl* and *manic_miner.ctl*
- Fixed the bug in *sna2skool.py* that made it list referrers of entry points in non-code blocks
- Added support to *sna2skool.py* for the `LD IXh,r` and `LD IXl,r` instructions

1.0.6 (2010-02-04)

Above each entry point in a code block, *sna2skool.py* will insert a comment containing a list of the routines that call or jump to that entry point.

1.0.5 (2010-02-03)

Made the following changes to *sna2skool.py*:

- Added the `-t` option (to show ASCII text in the comment fields)
- Set block titles according to the apparent contents (code/text/data) when using the `-g` option

1.0.4 (2010-02-02)

Made the following changes to *sna2skool.py*:

- Fixed the bug that caused the last instruction before the 64K boundary to be disassembled as a `DEFB` statement
- Added the `-g` option (to generate a control file using rudimentary static code analysis)
- Added the `-s` option (to specify the disassembly start address)

1.0.3 (2010-02-01)

- *sna2skool.py* copes with instructions that cross the 64K boundary
- *skool2html.py* writes the 'Game status buffer', 'Glossary', 'Trivia', 'Bugs' and 'Pokes' pages for a *skool* file specified by the `-f` option (in addition to the disassembly files and memory maps)

1.0.2 (2010-01-31)

Modified *sna2skool.py* so that it:

- recognises instructions that are unchanged by a DD or FD prefix
- recognises instructions with a DDCB or FDCB prefix
- produces a 4-byte `DEFB` for the ED-prefixed `LD HL,(nn)` and `LD (nn),HL` instructions
- produces a 2-byte `DEFB` for a relative jump across the 64K boundary

1.0.1 (2010-01-30)

Fixed the following bugs in *sna2skool.py*:

- 'X' was replaced by 'Y' instead of 'IX' by 'IY' (leading to nonsense mnemonics such as `YOR IYh`)
- ED72 was disassembled as `SBC HL,BC` instead of `SBC HL,SP`
- ED7A was disassembled as `ADD HL,SP` instead of `ADC HL,SP`
- ED63 and ED6B were disassembled as `LD (nn),HL` and `LD HL,(nn)` (which is correct, but won't assemble back to the same bytes)

1.0 (2010-01-28)

Initial public release.

CHAPTER 8

Technical reference

8.1 Parsing, rendering, and modes

The following subsections explain at a high level the two phases involved in transforming a *skool* file (and its related *ref* files, if any exist) into HTML or ASM format by using *skool2html.py* or *skool2asm.py*: parsing and rendering.

8.1.1 Parsing

In the first phase, the *skool* file is parsed. Parsing a *skool* file entails reading each line of the file, and processing any relevant *ASM directives* that are found along the way.

After an ASM directive has been processed, it is discarded, so that it cannot be 'seen' during the rendering phase. The purpose of the ASM directives is to transform the *skool* file into something suitable for rendering (in either HTML or ASM format) later on.

Whether a particular ASM directive is processed depends on the mode in which the parsing is being done: HTML mode or ASM mode.

HTML mode

HTML mode is used when the target output format is HTML, as is the case when running *skool2html.py*. In HTML mode, most ASM directives are ignored because they are irrelevant to the purpose of creating the HTML version of the disassembly. The only ASM directives that are processed in HTML mode are the following:

- @*keep*
- @*label*
- @*bfix block directives*
- @*isub block directives*
- @*ofix block directives*
- @*rfix block directives*
- @*rsub block directives*

The reason that the block directives are processed is that they may define two different versions of a section of code or data: first, a version to include in the output if the corresponding ASM mode (@*bfix*, @*isub*, @*ofix*, @*rfix*, @*rsub*) is in effect; and second, a version to include in the output if the corresponding ASM mode is not in effect - which will always be the case when parsing in HTML mode.

For example:

```
; @bfix-begin
 32459 CP 26  ; This is a bug; it should be 'CP 27'
; @bfix+else
       CP 27  ;
; @bfix+end
```

This instance of a `@bfix` block directive defines two versions of a section of code. The first version (between `@bfix-begin` and `@bfix+else`) will be included in the HTML output, and the second version (between `@bfix+else` and `@bfix+end`) will be omitted.

ASM mode

ASM mode is used when the target output format is ASM, as is the case when running *skool2asm.py*. In ASM mode, all ASM directives are processed.

8.1.2 Rendering

In the second phase, the *skool* file (stripped of all its ASM directives during the parsing phase) is 'rendered' - as either HTML or ASM, depending on the mode.

HTML mode

HTML mode is used to render the *skool* file (and its related *ref* file, if one exists) as a bunch of HTML files. During rendering, any *skool macros* found along the way are expanded to the required HTML markup.

ASM mode

ASM mode is used to render the *skool* file as a single, assembler-ready ASM file. During rendering, any *skool macros* found along the way are expanded to some appropriate plain text.

8.2 Control files

A control file contains a list of start addresses of code and data blocks. This information can be used by *sna2skool.py* to organise a *skool* file into corresponding code and data blocks.

Each block address in a control file is marked with a 'control directive', which is a single letter that indicates what the block contains:

- `b` indicates a data block
- `c` indicates a code block
- `g` indicates a game status buffer entry
- `i` indicates a block that should be ignored
- `t` indicates a block containing text
- `u` indicates an unused block of memory
- `w` indicates a block containing words (two-byte values)
- `z` indicates an unused block containing all zeroes

(If these letters remind you of the valid characters that may appear in the first column of each line of a *skool file*, that is no coincidence.)

For example:

```
c 24576 Do stuff
b 24832 Important data
t 25088 Interesting messages
u 25344 Unused
```

This control file declares that:

- Everything before 24576 should be ignored
- There is a routine at 24576-24831 which should be titled 'Do stuff'
- There is data at 24832-25087
- There is text at 25088-25343
- Everything from 25344 onwards is unused (but should still be disassembled as data)

Addresses may be written as hexadecimal numbers, too; the equivalent example control file using hexadecimal notation would be:

```
c $6000 Do stuff
b $6100 Important data
t $6200 Interesting messages
u $6300 Unused
```

Besides the declaration of block types, addresses and titles, the control file syntax also supports the declaration of the following things:

- Block descriptions
- Register values
- Mid-block comments
- Block end comments
- Sub-block types and comments
- DEFB/DEFM/DEFW/DEFS statement lengths in data, text and unused sub-blocks
- ASM directives (except block directives)

The syntax for declaring these things is described in the following sections.

8.2.1 Block descriptions

To provide a description for a code block at 24576 (for example), use the `D` directive thus:

```
c 24576 This is the title of the routine at 24576
D 24576 This is the description of the routine at 24576.
```

If the description consists of two or more paragraphs, each one should be declared with a separate `D` directive:

```
D 24576 This is the first paragraph of the description of the routine at 24576.
D 24576 This is the second paragraph of the description of the routine at 24576.
```

8.2.2 Register values

To declare the values of the registers upon entry to the routine at 24576, add one line per register with the `R` directive thus:

```
R 24576 A An important value in the accumulator
R 24576 DE Display file address
```

8.2.3 Mid-block comments

To declare a mid-block comment that will appear above the instruction at 24592, use the `D` directive thus:

```
D 24592 The next section of code does something really important.
```

If the mid-block comment consists of two or more paragraphs, each one should be declared with a separate `D` directive:

```
D 24592 This is the first paragraph of the mid-block comment.
D 24592 This is the second paragraph of the mid-block comment.
```

8.2.4 Block end comments

To declare a comment that will appear at the end of the routine at 24576, use the `E` directive thus:

```
E 24576 And so the work of this routine is done.
```

If the block end comment consists of two or more paragraphs, each one should be declared with a separate `E` directive:

```
E 24576 This is the first paragraph of the end comment for the routine at 24576.
E 24576 This is the second paragraph of the end comment for the routine at 24576.
```

8.2.5 Sub-block syntax

Sometimes a block marked as one type (code, data, text, or whatever) may contain instructions or statements of another type. For example, a word (`w`) block may contain the odd non-word here and there. To declare such sub-blocks whose type does not match that of the containing block, use the following syntax:

```
w 32768 A block containing mostly words
B 32800,3 But here's a sub-block of 3 bytes at 32800
T 32809,8 And an 8-byte text string at 32809
C 32821,10 And 10 bytes of code at 32821 too?
```

The directives (`B`, `T` and `C`) used here to mark the sub-blocks are the upper case equivalents of the directives used to mark top-level blocks (`b`, `t` and `c`). The comments at the end of these sub-block declarations are taken as instruction-level comments and will appear as such in the resultant *skool* file.

If an instruction-level comment spans a group of two or more sub-blocks of different types, it must be declared with an `M` directive:

```
M 40000,21 This comment covers the following 3 sub-blocks
B 40000,3
W 40003,10
T 40013,8
```

If the length parameter is omitted from an `M` directive, the comment is assumed to cover all sub-blocks from the given start address to the end of the top-level block.

Three bits of sub-block syntax left. First, the blank sub-block directive:

```
c 24576 A great routine
  24580,11 A great section of code at 24580
```

This is equivalent to:

```
c 24576 A great routine
C 24580,11 A great section of code at 24580
```

That is, the the type of a blank sub-block directive is taken to be the same as that of the parent block.

Next, the address range:

```
c 24576 A great routine
  24580-24590 A great section of code at 24580
```

This is equivalent to:

```
c 24576 A great routine
  24580,11 A great section of code at 24580
```

That is, you can specify the extent of a sub-block using either an address range, or an address and a length.

Finally, the implicit sub-block extent:

```
c 24576 A great routine
  24580 A great section of code at 24580
  24588,10 Another great section of code at 24590
```

This is equivalent to:

```
c 24576 A great routine
  24580,8 A great section of code at 24580
  24588,10 Another great section of code at 24588
```

But the declaration of the length (8) of the sub-block at 24580 is redundant, because the sub-block is implicitly terminated by the declaration of the sub-block at 24588 that follows. This is exactly how top-level block declarations work: each top-level block is implicitly terminated by the declaration of the next one.

8.2.6 Statement lengths in 'B', 'T', 'W' and 'Z' sub-blocks

Normally, a `B` sub-block declared thus:

```
B 24580,12 Interesting data
```

would result in something like this in the corresponding skool file:

```
24580 DEFB 1,2,3,4,5,6,7,8 ; {Interesting data
24588 DEFB 9,10,11,12      ; }
```

But what if you wanted to split the data in this sub-block into groups of 3 bytes each? That can be achieved with:

```
B 24580,12,3 Interesting data
```

which would give:

```
24580 DEFB 1,2,3    ; {Interesting data
24583 DEFB 4,5,6
24586 DEFB 7,8,9
24589 DEFB 10,11,12 ; }
```

That is, in a `B` directive, the desired DEFB statement lengths may be given as a comma-separated list of numbers following the sub-block length parameter, and the final number in the list is used for all remaining data in the block. So, for example:

```
B 24580,12,1,2,3 Interesting data
```

would give:

```
24580 DEFB 1        ; {Interesting data
24581 DEFB 2,3
24583 DEFB 4,5,6
24586 DEFB 7,8,9
24589 DEFB 10,11,12 ; }
```

If the statement length list contains sequences of two or more identical lengths, as in:

```
B 24580,21,2,2,2,2,2,2,1,1,1,3
```

then it may be abbreviated thus:

```
B 24580,21,2*6,1*3,3
```

The same syntax can be used for `T`, `W` and `Z` sub-blocks too. For example:

```
Z 32768,100,25 Four 25-byte chunks of zeroes
```

would give:

```
32768 DEFS 25 ; {Four 25-byte chunks of zeroes
32793 DEFS 25
32818 DEFS 25
32843 DEFS 25 ; }
```

DEFB and DEFM statements may contain both bytes and strings; for example:

```
40000 DEFM "Hi ",5
40004 DEFB 4,"go"
```

Such statements can be encoded in a control file thus:

```
T 40000,4,3:B1
B 40004,3,1:T2
```

That is, the length of a string in a DEFB statement is prefixed by `T`, the length of a sequence of bytes in a DEFM statement is prefixed by `B`, and the lengths of all strings and byte sequences are separated by colons. This notation can also be combined with the '*' notation; for example:

```
T 50000,8,2:B2*2
```

which is equivalent to:

```
T 50000,8,2:B2,2:B2
```

8.2.7 ASM directives

To declare an ASM directive for a block or an individual instruction, use the following syntax:

```
; @directive:address[=value]
```

where:

- `directive` is the directive name
- `address` is the address of the block or instruction to which the directive applies
- `value` is the value of the directive (if it requires one)

For example, to declare a *@label* directive for the instruction at 32768:

```
; @label:32768=LOOP
```

Note that neither ASM block directives (such as the *@bfix block directives*) nor the exact location of *@org*, *@writer*, *@start*, *@end*, *@ignoreua* and *@set* ASM directives can be preserved using this syntax.

8.2.8 Instruction-level comments

One limitation of storing instruction-level comments as shown so far is that there is no way to distinguish between a blank comment that spans two or more instructions and no comment at all. For example, both:

```
30000 DEFB 0 ; {
30001 DEFB 0 ; }
```

and:

```
30000 DEFB 0 ;
30001 DEFB 0 ;
```

would be preserved thus:

```
B 30000,2,1
```

To solve this problem, a special syntax is used to preserve blank multi-instruction comments:

```
B 30000,2,1 .
```

When restored, this comment is reduced to an empty string.

But how then to preserve a multi-instruction comment consisting of a single dot (`.`), or a sequence of two or more dots? In that case, another dot is prefixed to the comment. So:

```
30000 DEFB 0 ; {...
30001 DEFB 0 ; }
```

is preserved thus:

```
B 30000,2,1 ....
```

Note that this scheme does not apply to multi-instruction comments that contain at least one character other than a dot; such comments are preserved verbatim (that is, without a dot prefix).

8.2.9 Control file comments

A comment may be added to a control file by starting a line with something other than a space, a control directive, or `; @`. For example:

```
; This is a comment
# This is another comment
% This is yet another comment
```

8.2.10 Limitations

A control file can be useful in the early stages of developing a *skool* file for reorganising code and data blocks, but it cannot preserve the following elements:

- ASM block directives
- the exact locations of @*org*, @*writer*, @*start*, @*end*, @*ignoreua* and @*set* ASM directives
- data definition entries ('d' blocks) and remote entries ('r' blocks)
- comments that are not part of a code or data block

Skool file templates, however, can preserve all of these elements, and so may be a better choice for *skool* files that contain any of them.

8.2.11 Revision history

Version	Changes
1.0.7	Added support for block titles, block descriptions, register values, mid-block comments, block end comments, sub-block types and instruction-level comments
2.0.6	Added support for hexadecimal numbers
2.1	Added support for DEFB statement lengths in `B` sub-blocks
2.1.1	Added the `M` directive
2.1.2	Added support for DEFM, DEFW and DEFS statement lengths in `T`, `W` and `Z` sub-blocks
2.2	Added support for the `*` notation in DEFB, DEFM, DEFW and DEFS statement length lists in `B`, `T`, `W` and `Z` sub-blocks
2.4	Added support for non-block ASM directives
3.1.4	Added support for DEFB and DEFM statements that contain both strings and bytes
3.6	Added support for preserving blank comments that span two or more instructions

8.3 Skool files

A *skool* file contains the list of Z80 instructions that make up the routines and data blocks of the program being disassembled, with accompanying comments (if any).

8.3.1 Skool file format

A *skool* file must be in a certain format to ensure that it is processed correctly by *skool2html.py*, *skool2asm.py*, *skool2ctl.py* and *skool2sft.py*. The rules are as follows:

1. Entries (an 'entry' being a routine or data block) must be separated by blank lines, and an entry must not contain any blank lines.
2. Lines in an entry may start with one of `;*bcdgirtuwz`, where:
   - `;` begins a comment line
   - `*` denotes an entry point in a routine
   - `b` denotes the first instruction in a data block
   - `c` denotes the first instruction in a code block (routine)
   - `d` denotes the first instruction in a *data definition entry*

- `g` denotes the first instruction in a game status buffer entry
- `i` denotes an ignored entry
- `r` denotes the first instruction in a *remote entry*
- `t` denotes the first instruction in a data block that contains text
- `u` denotes the first instruction in an unused code or data block
- `w` denotes the first instruction in a data block that contains two-byte values (words)
- `z` denotes the first instruction in a data block that contains only zeroes
- a space begins a line that does not require any of the markers listed above

The format of a non-comment line is:

```
C##### INSTRUCTION ; comment
```

where:

- `C` is one of the characters listed above: `* bcdgirtuwz`
- `#####` is an address (e.g. `24576`, or `$6000` if you prefer hexadecimal notation)
- `INSTRUCTION` is an instruction (e.g. `LD A,(HL)`)
- `comment` is a comment (which may be blank)

The comment for a single instruction may span multiple lines thus:

```
c24296 CALL 57935    ; This comment is too long to fit on a single line, so
                     ; we use two lines
```

A comment may also be associated with more than one instruction by the use of braces ('{' and '}') to indicate the start and end points, thus:

```
*24372 SUB D         ; {This comment applies to the two instructions at
 24373 JR NZ,24378   ; 24372 and 24373}
```

The opening and closing braces are removed before the comment is rendered in ASM or HTML mode. (See *Braces in comments*.)

Comments may appear between instructions, or after the last instruction in an entry; paragraphs in such comments must be separated by a comment line containing a dot (`.`) on its own. For example:

```
*28975 JR 28902
; This is a mid-block comment between two instructions.
; .
; This is the second paragraph of the comment.
 28977 XOR A
```

Lines that start with `*` will have their addresses shown in bold in the HTML version of the disassembly (generated by *skool2html.py*), and will have labels generated for them in the ASM version (generated by *skool2asm.py*).

3. Tables (grids) have their own markup syntax. See *#TABLE* for details.
4. Entry headers are a sequence of comment lines broken into three sections:
   - Entry title
   - Entry description (optional)

- Registers (optional)

The sections are separated by an empty comment line, and paragraphs within the entry description must be separated by a comment line containing a dot (`.`) on its own. For example:

```
; This is the entry title
;
; This is the first paragraph of the entry description.
; .
; This is the second paragraph of the entry description.
;
; A An important parameter
; B Another important parameter
```

If a register section is required, but an entry description is not, a blank entry description may be specified by using a dot (`.`) thus:

```
; This is the title of an entry that has no description
;
; .
;
; A An important parameter
; B Another important parameter
```

Registers may be listed as shown above, or with colon-terminated prefixes (such as 'Input:' and 'Output:', or simply 'I:' and 'O:') to distinguish input values from output values:

```
;  Input:A An important parameter
;        B Another important parameter
; Output:C The result
```

In the HTML version of the disassembly, input values and output values are shown in separate tables. If a register's prefix begins with the letter 'O', it is regarded as an output value; if it begins with any other letter, it is regarded as an input value. If a register has no prefix, it will be placed in the same table as the previous register; if there is no previous register, it will be placed in the table of input values.

8.3.2 Braces in comments

As noted above, opening and closing braces (`{`, `}`) are used to mark the start and end points of an instruction-level comment that is associated with more than one instruction, and the braces are removed before the comment is rendered. This means that if the comment requires an opening or closing brace *when rendered*, some care must be taken to get the syntax correct.

The rules regarding an instruction-level comment that starts with an opening brace are as follows:

- The comment terminates on the line where the total number of closing braces in the comment becomes equal to or greater than the total number of opening braces
- Adjacent opening braces at the start of the comment are removed before rendering
- Adjacent closing braces at the end of the comment are removed before rendering

By these rules, it is possible to craft an instruction-level comment that contains matched or unmatched opening and closing braces when rendered.

For example:

```
b50000 DEFB 0  ; {{This comment (which spans two instructions) has an
 50001 DEFB 0  ; unmatched closing brace} }
```

will render in ASM mode as:

```
DEFB 0                  ; This comment (which spans two instructions) has an
DEFB 0                  ; unmatched closing brace}
```

And:

```
b50002 DEFB 0  ; { {{Matched opening and closing braces}} }
```

will render as:

```
DEFB 0                  ; {{Matched opening and closing braces}}
```

Finally:

```
b50003 DEFB 0  ; { {Unmatched opening brace}}
```

will render as:

```
DEFB 0                  ; {Unmatched opening brace
```

8.3.3 Data definition entries

If the first instruction line in an entry starts with `d`, the entry is regarded as a data definition entry. Such entries do not appear in the memory map generated by *skool2html.py*, but may contain DEFB, DEFW, DEFM and DEFS assembler directives that will be parsed, and so can be used to insert data into the memory snapshot.

For example:

```
; The eight bytes of code in this routine are also used as UDG data.
; .
; #HTML(#UDG44919)
c44919 LD DE,46572   ;
 44922 CP 200        ;
 44924 JP 45429      ;

d44919 DEFB 17,236,181,254,200,195,117,177
```

This data definition entry is required to define the bytes for addresses 44919-44926. If it were not present, the memory snapshot would contain zeroes at those addresses, and the UDG created by *skool2html.py* would be blank. The reason for this is that the skool file parser will only convert DEFB, DEFW, DEFM and DEFS assembler directives into a sequence of bytes; it does not convert assembly language instructions into the equivalent byte values (it is not a Z80 assembler).

8.3.4 Remote entries

If the first instruction line in an entry starts with `r`, the entry is regarded as a remote entry. Such entries do not appear in the memory map generated by *skool2html.py*, but they enable `JR`, `JP` and `CALL` instructions to be hyperlinked to entries defined in other *skool* files.

For example:

```
r26880 main
```

This entry, if it were present in a secondary *skool* file, would enable any `JR`, `JP` and `CALL` instruction with 26880 as the operand to be hyperlinked to that routine in the main disassembly (the entry for which should be defined in the main *skool* file).

If the desired target of the hyperlink is an entry point within a routine that is defined in another *skool* file (as opposed to the address of the routine itself), both the routine address and the entry point address should be declared in the remote entry. For example:

```
r29012 main
 29015
```

This enables hyperlinks to 29015 in the main disassembly, which is an entry point in the routine at 29012. It also enables the *#R* macro to create hyperlinks to remote entry points using the short form:

```
#R29015@main
```

instead of the longer form (which would be required if the remote entry were not defined):

```
#R29012@main#29015(29015)
```

8.3.5 Revision history

Version	Changes
2.0	Added support for data definition entries and remote entries
2.1	Added support for hexadecimal numbers
2.4	Added the ability to separate paragraphs and specify a blank entry description by using a dot (`.`) on a line of its own
3.1.2	Added support for 'Input' and 'Output' prefixes in register sections

8.4 Skool file templates

A skool file template defines the basic structure of a *skool* file, but, unlike a *skool* file, contains directives on how to disassemble a program into Z80 instructions instead of the Z80 instructions themselves. The directives are similar to those that may appear in a control file.

The *skool2sft.py* command can generate a skool file template from an existing *skool* file; the *sna2skool.py* command can then generate a *skool* file from the template and an appropriate snapshot.

8.4.1 Skool file template format

A skool file template has the same layout as a *skool* file, except that the lines in 'b', 'c', 'g', 't', 'u', 'w' and 'z' blocks that correspond to Z80 instructions look like this:

```
xX#####,n[;c[ comment]]
```

where:

- `x` is one of the characters `* bcgtuwz` (with the same meaning as in a *skool file*)
- `X` is one of the characters `BCTWZ` (with the same meaning as in a *control file*)
- `#####` is the address at which to start disassembling
- `n` is the number of bytes to disassemble
- `c` is the index of the column in which the comment marker (`;`) appears in the line (if it does appear)
- `comment`, if present, is the instruction-level comment for the line on which the instruction occurs

If a comment for a single instruction spans two or more lines in a *skool* file, as in:

```
c24296 CALL 57935    ; This comment is too long to fit on a single line, so
                     ; we use two lines
```

then it will be rendered in the skool file template thus:

```
cC24296,3;21 This comment is too long to fit on a single line, so
 ;21 we use two lines
```

Sequences of DEFB statements can be declared on a single line thus:

```
bB40960,8*2,5
```

which is equivalent to:

```
bB40960,8
 B40968,8
 B40976,5
```

The same syntax also applies for declaring sequences of DEFM, DEFW and DEFS statements.

DEFB and DEFM statements may contain both strings and bytes; for example:

```
b30000 DEFB 1,2,3,4,"Hello!"
 30010 DEFM "A",5,6
 30013 DEFM "B",7,8
```

Such statements will be rendered in the skool file template thus:

```
bB30000,4:T6
 T30010,1:B2*2
```

Finally, any line that begins with a hash character (#) is ignored by *sna2skool.py*, and will not show up in the *skool* file.

8.4.2 Data definition entries

In the same way as *skool2html.py* uses data definition entries ('d' blocks) in a *skool* file to insert data into the memory snapshot it constructs, *sna2skool.py* uses data definition entries in a skool file template to replace data in the snapshot given on the command line. This feature can be used to make sure that a 'volatile' part of memory is set to a specific value before being disassembled.

For example, if address 32400 holds the number of lives, you could make sure that its contents are set to 0 so that it will disassemble to `DEFB 0` (whatever the contents may be in the snapshot itself) thus:

```
d32400 DEFB 0

; Number of lives
bB32400,1
```

8.4.3 Revision history

Version	Changes
2.4	New
3.1.4	Added support for DEFB and DEFM statements that contain both strings and bytes

8.5 Skool macros

skool files and *ref* files may contain skool macros that are 'expanded' to an appropriate piece of HTML markup (when rendering in HTML mode), or to an appropriate piece of plain text (when rendering in ASM mode).

Skool macros have the following general form:

```
#MACROrparam1,rparam2,...[,oparam1,oparam2,...]
```

where:

- `MACRO` is the macro name
- `rparam1`, `rparam2` etc. are required parameters
- `oparam1`, `oparam2` etc. are optional parameters

If an optional parameter is left blank or omitted entirely, it assumes its default value. So, for example:

```
#UDG39144
```

is equivalent to:

```
#UDG39144,56,4,1,0,0,0
```

and:

```
#UDG30115,23,,2,1
```

is equivalent to:

```
#UDG30115,23,4,2,1
```

Numeric parameters may be given in decimal notation (as already shown in the examples above), or in hexadecimal notation (prefixed by `$`):

```
#UDG$98E8,$06
```

The skool macros recognised by SkoolKit are described in the following subsections.

8.5.1 #BUG

In HTML mode, the `#BUG` macro expands to a hyperlink (`<a>` element) to the 'Bugs' page, or to a specific entry on that page.

```
#BUG[#name][(link text)]
```

- `#name` is the named anchor of a bug (if linking to a specific one)
- `link text` is the link text to use

The anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

In HTML mode, if the link text is blank, the title of the bug entry (if linking to a specific one) is substituted; if the link text is omitted entirely, 'bug' is substituted.

In ASM mode, the `#BUG` macro expands to the link text, or 'bug' if the link text is blank or omitted.

For example:

```
 42726 DEFB 130 ; This is a #BUG#bug1; it should be 188
```

In HTML mode, this instance of the `#BUG` macro expands to a hyperlink to an entry on the 'Bugs' page.

In ASM mode, this instance of the `#BUG` macro expands to 'bug'.

See also *#FACT* and *#POKE*.

Version	Changes
2.3.1	If left blank, the link text defaults to the bug entry title in HTML mode

8.5.2 #CALL

In HTML mode, the `#CALL` macro expands to the return value of a method on the *HtmlWriter* class or subclass that is being used to create the HTML disassembly (as defined by the `HtmlWriterClass` parameter in the *[Config]* section of the *ref* file).

In ASM mode, the `#CALL` macro expands to the return value of a method on the *AsmWriter* class or subclass that is being used to generate the ASM output (as defined by the *@writer* ASM directive in the *skool* file).

```
#CALL:methodName(args)
```

- `methodName` is the name of the method to call
- `args` is a comma-separated list of arguments to pass to the method

For example:

```
; The byte at address 32768 is #CALL:peek(32768).
```

This instance of the `#CALL` macro expands to the return value of the *peek* method (on the *HtmlWriter* or *AsmWriter* subclass being used) when called with the argument `32768`.

For information on writing methods that may be called by a `#CALL` macro, see the documentation on *extending SkoolKit*.

Version	Changes
2.1	New
3.1	Added support for ASM mode

8.5.3 #CHR

In HTML mode, the `#CHR` macro expands to a numeric character reference (`&#num;`). In ASM mode, it expands to a unicode character in the UTF-8 encoding.

```
#CHRnum
```

or:

```
#CHR(num)
```

For example:

```
 26751 DEFB 127   ; This is the copyright symbol: #CHR169
```

In HTML mode, this instance of the `#CHR` macro expands to `&#169;`. In ASM mode, it expands to the copyright symbol.

Version	Changes
3.1	New

8.5.4 #D

The `#D` (Description) macro expands to the title of an entry (a routine or data block) in the memory map.

```
#Daddr
```

- `addr` is the address of the entry.

For example:

```
; Now we make an indirect jump to one of the following routines:
; .
; #TABLE(default,centre)
; { =h Address | =h Description }
; { #R27126    | #D27126 }
```

This instance of the `#D` macro expands to the title of the routine at 27126.

8.5.5 #EREFS

The `#EREFS` (Entry point REFerenceS) macro expands to a comma-separated sequence of hyperlinks to (in HTML mode) or addresses of (in ASM mode) the routines that jump to or call a given address.

```
#EREFSaddr
```

- `addr` is the address to search for references to

See also *#REFS*.

Version	Changes
3.1	Added support for ASM mode

8.5.6 #FACT

In HTML mode, the `#FACT` macro expands to a hyperlink (`<a>` element) to the 'Trivia' page, or to a specific entry on that page.

```
#FACT[#name][(link text)]
```

- `#name` is the named anchor of a trivia entry (if linking to a specific one)
- `link text` is the link text to use

The anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

In HTML mode, if the link text is blank, the title of the trivia entry (if linking to a specific one) is substituted; if the link text is omitted entirely, 'fact' is substituted.

In ASM mode, the `#FACT` macro expands to the link text, or 'fact' if the link text is blank or omitted.

For example:

```
See the trivia entry #FACT#interestingFact() for details.
```

In HTML mode, this instance of the `#FACT` macro expands to a hyperlink to an entry on the 'Trivia' page, with link text equal to the title of the entry.

See also *#BUG* and *#POKE*.

Version	Changes
2.3.1	If left blank, the link text defaults to the trivia entry title in HTML mode; added support for ASM mode

8.5.7 #FONT

In HTML mode, the `#FONT` macro expands to an `<img>` element for an image of text rendered in the game font.

```
#FONT[:(text)]addr[,chars,attr,scale][{X,Y,W,H}][(fname)]
```

- `text` is the text to render (default: the 96 characters from code 32 to code 127)
- `addr` is the base address of the font graphic data
- `chars` is the number of characters to render (default: the length of `text`)
- `attr` is the attribute byte to use (default: 56)
- `scale` is the required scale of the image (default: 2)
- `X` is the x-coordinate of the leftmost pixel column of the constructed image to include in the final image (if greater than 0, the image will be cropped on the left)
- `Y` is the y-coordinate of the topmost pixel row of the constructed image to include in the final image (if greater than 0, the image will be cropped on the top)
- `W` is the width of the final image (if less than the full width of the constructed image, the image will be cropped on the right)
- `H` is the height of the final image (if less than the full height of the constructed image, the image will be cropped on the bottom)
- `fname` is the name of the image file (default: '*font*'); '*.png*' or '*.gif*' will be appended (depending on the default image format specified in the *[ImageWriter]* section of the *ref* file) if not present

If `text` contains a closing bracket - `)` - then the macro will not expand as required. In that case, square brackets, braces or any character that does not appear in `text` may be used as delimiters; for example:

```
#FONT:[(0) OK]$3D00
#FONT:{(0) OK}$3D00
#FONT:/(0) OK/$3D00
```

The `#FONT` macro is not supported in ASM mode.

If an image with the given filename doesn't already exist, it will be created. If `fname` starts with a '/', the filename is taken to be relative to the root of the HTML disassembly; otherwise the filename is taken to be relative to the directory defined by the `FontImagePath` parameter in the *[Paths]* section of the *ref* file.

For example:

```
; Font graphic data
;
; #HTML[#FONT:(0123456789)49152]
```

In HTML mode, this instance of the `#FONT` macro expands to an `<img>` element for the image of the digits 0-9 in the 8*8 font whose graphic data starts at 49152.

Version	Changes
2.0.5	Added the `fname` parameter and support for regular 8x8 fonts
3.0	Added image-cropping capabilities
3.6	Added the `text` parameter, and made the `chars` parameter optional

8.5.8 #HTML

The `#HTML` macro expands to arbitrary text (in HTML mode) or to an empty string (in ASM mode).

```
#HTML(text)
```

The `#HTML` macro may be used to render HTML (which would otherwise be escaped) from a *skool* file. For example:

```
; #HTML(For more information, go <a href="http://example.com/">here</a>.)
```

If `text` contains a closing bracket - `)` - then the macro will not expand as required. In that case, square brackets, braces or any character that does not appear in `text` (except for an upper case letter) may be used as delimiters:

```
#HTML[text]
#HTML{text}
#HTML@text@
```

`text` may contain other skool macros, which will be expanded before rendering. For example:

```
; #HTML[The UDG defined here (32768) looks like this: #UDG32768,4,1]
```

See also *#UDGTABLE*.

Version	Changes
3.1.2	New

8.5.9 #LINK

In HTML mode, the `#LINK` macro expands to a hyperlink (`<a>` element) to another page.

```
#LINK:PageId[#name](link text)
```

- `PageId` is the ID of the page to link to
- `name` is the name of an anchor on the page to link to
- `link text` is the link text to use

In HTML mode, if the link text is blank, the page's link text (as defined in the *[Links]* section or the relevant *[Page:*]* section of the *ref* file) is substituted.

In ASM mode, the `#LINK` macro expands to the link text.

The page IDs that may be used are the same as the file IDs that may be used in the *[Paths]* section of a *ref* file, or the page IDs defined by *[Page:*]* sections.

For example:

```
; See the #LINK:Glossary(glossary) for a definition of 'chuntey'.
```

In HTML mode, this instance of the `#LINK` macro expands to a hyperlink to the 'Glossary' page, with link text 'glossary'.

In ASM mode, this instance of the `#LINK` macro expands to 'glossary'.

Version	Changes
2.1	New
3.1.3	If left blank, the link text defaults to the page's link text in HTML mode

8.5.10 #LIST

The `#LIST` macro marks the beginning of a list of bulleted items; `LIST#` is used to mark the end. Between these markers, the list items are defined.

```
#LIST[(class)]<items>LIST#
```

- `class` is the CSS class to use for the `<ul>` element

Each item in a list must start with `{` followed by a whitespace character, and end with `}` preceded by a whitespace character.

For example:

```
; #LIST(data)
; { Item 1 }
; { Item 2 }
; LIST#
```

This list has two items, and will have the CSS class 'data'.

In ASM mode, lists are rendered as plain text, with each item on its own line, and an asterisk as the bullet character. The bullet character can be changed by using a *@set* directive to set the `bullet` property on the ASM writer.

Version	Changes
3.2	New

8.5.11 #POKE

In HTML mode, the `#POKE` macro expands to a hyperlink (`<a>` element) to the 'Pokes' page, or to a specific entry on that page.

```
#POKE[#name][(link text)]
```

- `#name` is the named anchor of a poke (if linking to a specific one)
- `link text` is the link text to use

The anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

In HTML mode, if the link text is blank, the title of the poke entry (if linking to a specific one) is substituted; if the link text is omitted entirely, 'poke' is substituted.

In ASM mode, the `#POKE` macro expands to the link text, or 'poke' if the link text is blank or omitted.

For example:

```
; Of course, if you feel like cheating, you can always give yourself
; #POKE#infiniteLives(infinite lives).
```

In HTML mode, this instance of the `#POKE` macro expands to a hyperlink to an entry on the 'Pokes' page, with link text 'infinite lives'.

In ASM mode, this instance of the `#POKE` macro expands to 'infinite lives'.

See also *#BUG* and *#FACT*.

Version	Changes
2.3.1	If left blank, the link text defaults to the poke entry title in HTML mode; added support for ASM mode

8.5.12 #POKES

The `#POKES` (POKE Snapshot) macro POKEs values into the current memory snapshot.

```
#POKESaddr,byte[,length,step][;addr,byte[,length,step];...]
```

- `addr` is the address to POKE
- `byte` is the value to POKE `addr` with
- `length` is the number of addresses to POKE (default: 1)
- `step` is the address increment to use after each POKE (if `length`>1; default: 1)

For example:

```
The UDG looks like this:

#UDG32768(udg_orig)

But it's supposed to look like this:

#PUSHS
#POKES32772,254;32775,136
#UDG32768(udg_fixed)
#POPS
```

This instance of the `#POKES` macro does `POKE 32772,254` and `POKE 32775,136`, which fixes a graphic glitch in the UDG at 32768.

The `#POKES` macro expands to an empty string.

See also *#PUSHS* and *#POPS*.

Version	Changes
2.3.1	Added support for multiple addresses
3.1	Added support for ASM mode

8.5.13 #POPS

The `#POPS` (POP Snapshot) macro removes the current memory snapshot and replaces it with the one that was previously saved by a `#PUSHS` macro.

```
#POPS
```

The `#POPS` macro expands to an empty string.

See also *#PUSHS* and *#POKES*.

Version	Changes
3.1	Added support for ASM mode

8.5.14 #PUSHS

As a *skool* file is being parsed, a memory snapshot is built up from all the `DEFB`, `DEFW`, `DEFM` and `DEFS` instructions. After the file has been parsed, the memory snapshot may be used to build images of the game's graphic elements (for example).

The `#PUSHS` (PUSH Snapshot) macro saves the current snapshot, and replaces it with an identical copy with a given name.

```
#PUSHS[name]
```

- `name` is the snapshot name (defaults to an empty string)

The snapshot name must be limited to the characters '$', '#', 0-9, A-Z and a-z; it must not start with a capital letter.

For example:

```
The UDG at 32768 is supposed to look like this:

#PUSHS
#POKES32772,254
#UDG32768
#POPS
```

The `#PUSHS` macro expands to an empty string.

See also *#POKES* and *#POPS*.

Version	Changes
3.1	Added support for ASM mode

8.5.15 #R

In HTML mode, the `#R` (Reference) macro expands to a hyperlink (`<a>` element) to the disassembly page for a routine or data block, or to a line at a given address within that page.

```
#Raddr[@code][#name][(link text)]
```

- `addr` is the address of the routine or data block (or entry point thereof)
- `code` is the ID of the disassembly that contains the routine or data block (if not given, the current disassembly is assumed; otherwise this should be an ID defined in an `[OtherCode:*]` section of the ref file)
- `#name` is the named anchor of an item on the disassembly page
- `link text` is the link text to use (default: `addr`)

The disassembly ID (`code`) and anchor name (`name`) must be limited to the characters '$', '#', 0-9, A-Z and a-z.

In ASM mode, the `#R` macro expands to the link text if it is specified, or to the label for `addr`, or to `addr` if no label is found.

For example:

```
; Prepare for a new game
;
; Used by the routine at #R25820.
```

In HTML mode, this instance of the `#R` macro expands to a hyperlink to the disassembly page for the routine at 25820.

In ASM mode, this instance of the `#R` macro expands to the label for the routine at 25820 (or simply `25820` if that routine has no label).

Version	Changes
2.0	Added support for the `@code` notation
3.5	Added the ability to resolve (in HTML mode) the address of an entry point in another disassembly when an appropriate *remote entry* is defined

8.5.16 #REFS

The `#REFS` (REFerenceS) macro expands to a comma-separated sequence of hyperlinks to (in HTML mode) or addresses of (in ASM mode) the routines that jump to or call a given routine, or jump to or call any entry point within that routine.

```
#REFSaddr[(prefix)]
```

- `addr` is the address of the routine to search for references to
- `prefix` is the text to display before the sequence of hyperlinks or addresses if there is at least one reference (default: no text)

If there are no references, the macro expands to the following text:

```
Not used directly by any other routines
```

See also *#EREFS*.

Version	Changes
1.0.6	Added the `prefix` parameter
3.1	Added support for ASM mode

8.5.17 #REG

In HTML mode, the `#REG` (REGister) macro expands to a styled `<span>` element containing a register name.

```
#REGreg
```

- `reg` is the name of the register (e.g. 'a', 'bc')

In ASM mode, the `#REG` macro expands to the name of the register.

The register name must contain 1, 2 or 3 of the following characters:

```
abcdefhlirspxy'
```

For example:

```
 24623 LD C,31        ; #REGbc'=31
```

8.5.18 #SCR

In HTML mode, the `#SCR` (SCReenshot) macro expands to an `<img>` element for an image constructed from the display file and attribute file (or suitably arranged graphic data and attribute bytes elsewhere in memory) of the current memory snapshot (in turn constructed from the contents of the *skool* file).

```
#SCR[scale,x,y,w,h,dfAddr,afAddr][{X,Y,W,H}][(fname)]
```

- `scale` is the required scale of the image (default: 1)
- `x` is the x-coordinate of the top-left tile of the screen to include in the screenshot (default: 0)
- `y` is the y-coordinate of the top-left tile of the screen to include in the screenshot (default: 0)
- `w` is the width of the screenshot in tiles (default: 32)
- `h` is the height of the screenshot in tiles (default: 24)
- `dfAddr` is the base address of the display file (default: 16384)
- `afAddr` is the base address of the attribute file (default: 22528)
- `X` is the x-coordinate of the leftmost pixel column of the constructed image to include in the final image (if greater than 0, the image will be cropped on the left)

- `Y` is the y-coordinate of the topmost pixel row of the constructed image to include in the final image (if greater than 0, the image will be cropped on the top)
- `W` is the width of the final image (if less than the full width of the constructed image, the image will be cropped on the right)
- `H` is the height of the final image (if less than the full height of the constructed image, the image will be cropped on the bottom)
- `fname` is the name of the image file (default: '*scr*'); '*.png*' or '*.gif*' will be appended (depending on the default image format specified in the *[ImageWriter]* section of the *ref* file) if not present

The `#SCR` macro is not supported in ASM mode.

If an image with the given filename doesn't already exist, it will be created. If `fname` starts with a '/', the filename is taken to be relative to the root of the HTML disassembly; otherwise the filename is taken to be relative to the directory defined by the `ScreenshotImagePath` parameter in the *[Paths]* section of the *ref* file.

For example:

```
; #UDGTABLE
; { #SCR(loading) | This is the loading screen. }
; TABLE#
```

Version	Changes
2.0.5	Added the `scale`, `x`, `y`, `w`, `h` and `fname` parameters
3.0	Added image-cropping capabilities and the `dfAddr` and `afAddr` parameters

8.5.19 #SPACE

The `#SPACE` macro expands to one or more ` ` expressions (in HTML mode) or spaces (in ASM mode).

```
#SPACE[num]
```

or:

```
#SPACE([num])
```

- `num` is the number of spaces required (default: 1)

For example:

```
; '#SPACE8' (8 spaces)
t56832 DEFM "        "
```

In HTML mode, this instance of the `#SPACE` macro expands to:

```
        
```

In ASM mode, this instance of the `#SPACE` macro expands to a string containing 8 spaces.

The form `SPACE([num])` may be used to distinguish the macro from adjacent text where necessary. For example:

```
; 'Score:#SPACE(5)0'
t49152 DEFM "Score:     0"
```

Version	Changes
2.4.1	Added support for the `#SPACE([num])` syntax

8.5.20 #TABLE

The #TABLE macro marks the beginning of a table; TABLE# is used to mark the end. Between these markers, the rows of the table are defined.

```
#TABLE[([class[,class1[:w][,class2[:w]...]]])]<rows>TABLE#
```

- class is the CSS class to use for the <table> element
- class1, class2 etc. are the CSS classes to use for the <td> elements in columns 1, 2 etc.

Each row in a table must start with { followed by a whitespace character, and end with } preceded by a whitespace character. The cells in a row must be separated by | with a whitespace character on each side.

For example:

```
; #TABLE(default,centre)
; { 0 | Off }
; { 1 | On }
; TABLE#
```

This table has two rows and two columns, and will have the CSS class 'default'. The cells in the first column will have the CSS class 'centre'.

By default, cells will be rendered as <td> elements. To specify that a <th> element should be used instead, use the =h indicator before the cell contents:

```
; #TABLE
; { =h Header 1  | =h Header 2 }
; { Regular cell | Another one }
; TABLE#
```

It is also possible to specify colspan and rowspan attributes using the =c and =r indicators:

```
; #TABLE
; { =r2 2 rows  | X | Y }
; { =c2           2 columns }
; TABLE#
```

Finally, the =t indicator specifies that a cell should be transparent (i.e. have the same background colour as the page body).

If a cell requires more than one indicator, the indicators should be separated by commas:

```
; #TABLE
; { =h,c2 Wide header }
; { Column 1 | Column 2 }
; TABLE#
```

The CSS files included in SkoolKit provide two classes that may be used when defining tables:

- default - a class for <table> elements that provides a background colour to make the table stand out from the page body
- centre - a class for <td> elements that centres their contents

In ASM mode, tables are rendered as plain text, using dashes (−) and pipes (|) for the borders, and plus signs (+) where a horizontal border meets a vertical border.

ASM mode also supports the :w indicator in the #TABLE macro's parameters. The :w indicator marks a column as a candidate for having its width reduced (by wrapping the text it contains) so that the table will be no more than 79 characters wide when rendered. For example:

```
; #TABLE(default,centre,:w)
; { =h X | =h Description }
; { 0    | Text in this column will be wrapped in ASM mode to make the table less than 80 characters
; TABLE#
```

See also *#UDGTABLE*.

8.5.21 #UDG

In HTML mode, the `#UDG` macro expands to an `<img>` element for the image of a UDG (an 8x8 block of pixels).

```
#UDGaddr[,attr,scale,step,inc,flip,rotate][:maskAddr[,maskStep]][{X,Y,W,H}][(fname)]
```

- `addr` is the base address of the UDG bytes
- `attr` is the attribute byte to use (default: 56)
- `scale` is the required scale of the image (default: 4)
- `step` is the interval between successive bytes of the UDG (default: 1)
- `inc` will be added to each UDG byte before constructing the image (default: 0)
- `flip` is 1 to flip the UDG horizontally, 2 to flip it vertically, 3 to flip it both ways, or 0 to leave it as it is (default: 0)
- `rotate` is 1 to rotate the UDG 90 degrees clockwise, 2 to rotate it 180 degrees, 3 to rotate it 90 degrees anticlockwise, or 0 to leave it as it is (default: 0)
- `maskAddr` is the base address of the mask bytes to use for the UDG
- `maskStep` is the interval between successive mask bytes (default: `step`)
- `X` is the x-coordinate of the leftmost pixel column of the constructed image to include in the final image (if greater than 0, the image will be cropped on the left)
- `Y` is the y-coordinate of the topmost pixel row of the constructed image to include in the final image (if greater than 0, the image will be cropped on the top)
- `W` is the width of the final image (if less than the full width of the constructed image, the image will be cropped on the right)
- `H` is the height of the final image (if less than the full height of the constructed image, the image will be cropped on the bottom)
- `fname` is the name of the image file (if not given, a name based on `addr`, `attr` and `scale` will be generated); '*.png*' or '*.gif*' will be appended (depending on the default image format specified in the *[ImageWriter]* section of the *ref* file) if not present

The `#UDG` macro is not supported in ASM mode.

If an image with the given filename doesn't already exist, it will be created. If `fname` starts with a '/', the filename is taken to be relative to the root of the HTML disassembly; otherwise the filename is taken to be relative to the directory defined by the `UDGImagePath` parameter in the *[Paths]* section of the *ref* file.

For example:

```
; Safe key UDG
;
; #HTML[#UDG39144,6(safe_key)]
```

In HTML mode, this instance of the `#UDG` macro expands to an `<img>` element for the image of the UDG at 39144 (which will be named *safe_key.png* or *safe_key.gif*), with attribute byte 6 (INK 6: PAPER 0).

Version	Changes
2.0.5	Added the `fname` parameter
2.1	Added support for masks
2.3.1	Added the `flip` parameter
2.4	Added the `rotate` parameter
3.0	Added image-cropping capabilities
3.1.2	Made the `attr` parameter optional

8.5.22 #UDGARRAY

In HTML mode, the `#UDGARRAY` macro expands to an `<img>` element for the image of an array of UDGs (8x8 blocks of pixels).

```
#UDGARRAYwidth[,attr,scale,step,inc,flip,rotate];SPEC1[;SPEC2;...][{X,Y,W,H}](fname)
```

- `width` is the width of the image (in UDGs)
- `attr` is the default attribute byte of each UDG (default: 56)
- `scale` is the required scale of the image (default: 2)
- `step` is the default interval between successive bytes of each UDG (default: 1)
- `inc` is added to each UDG byte before constructing the image (default: 0)
- `flip` is 1 to flip the array of UDGs horizontally, 2 to flip it vertically, 3 to flip it both ways, or 0 to leave it as it is (default: 0)
- `rotate` is 1 to rotate the array of UDGs 90 degrees clockwise, 2 to rotate it 180 degrees, 3 to rotate it 90 degrees anticlockwise, or 0 to leave it as it is (default: 0)
- `X` is the x-coordinate of the leftmost pixel column of the constructed image to include in the final image (if greater than 0, the image will be cropped on the left)
- `Y` is the y-coordinate of the topmost pixel row of the constructed image to include in the final image (if greater than 0, the image will be cropped on the top)
- `W` is the width of the final image (if less than the full width of the constructed image, the image will be cropped on the right)
- `H` is the height of the final image (if less than the full height of the constructed image, the image will be cropped on the bottom)
- `fname` is the name of the image file; '*.png*' or '*.gif*' will be appended (depending on the default image format specified in the *[ImageWriter]* section of the *ref* file) if not present

`SPEC1`, `SPEC2` etc. are UDG specifications for the sets of UDGs that make up the array. Each UDG specification has the form:

```
udgAddr[,udgAttr,udgStep,udgInc][:maskAddr[,maskStep]]
```

- `udgAddr` is the address range specification for the set of UDGs (see below)
- `udgAttr` is the attribute byte of each UDG in the set (overrides `attr` if specified)
- `udgStep` is the interval between successive bytes of each UDG in the set (overrides `step` if specified)
- `udgInc` is added to each byte of every UDG in the set before constructing the image (overrides `inc` if specified)
- `maskAddr` is the address range specification for the set of mask UDGs (see below)

- `maskStep` is the interval between successive bytes of each mask UDG in the set (default: `udgStep`)

Address range specifications (`udgAddr` and `maskAddr`) may be given in one of the following forms:

- a single address (e.g. `39144`)
- a simple address range (e.g. `33008-33015`)
- an address range with a step (e.g. `32768-33792-256`)
- an address range with a horizontal and a vertical step (e.g. `63476-63525-1-16`; this form specifies the step between the base addresses of adjacent UDGs in each row as 1, and the step between the base addresses of adjacent UDGs in each column as 16)

Any of these forms of address ranges can be repeated by appending `xN`, where `N` is the desired number of repetitions. For example:

- `39648x3` is equivalent to `39648;39648;39648`
- `32768-32769x2` is equivalent to `32768;32769;32768;32769`

As many UDG specifications as required may be supplied, separated by semicolons; the UDGs will be arranged in a rectangular array with the given width.

The `#UDGARRAY` macro is not supported in ASM mode.

If an image with the given filename doesn't already exist, it will be created. If `fname` starts with a '/', the filename is taken to be relative to the root of the HTML disassembly; otherwise the filename is taken to be relative to the directory defined by the `UDGImagePath` parameter in the *[Paths]* section of the *ref* file.

For example:

```
; Base sprite
;
; #HTML[#UDGARRAY4;32768-32888-8(base_sprite.png)]
```

In HTML mode, this instance of the `#UDGARRAY` macro expands to an `<img>` element for the image of the 4x4 sprite formed by the 16 UDGs with base addresses 32768, 32776, 32784 and so on up to 32888; the image file will be named *base_sprite.png*.

Animated images

The `#UDGARRAY` macro may also be used to create an animated image from an arbitrary sequence of frames. To create a frame, the `fname` parameter must have one of the following forms:

- `name*` - writes an image file with this name, and also creates a frame with the same name
- `name1*name2` - writes an image file named *name1*, and also creates a frame named *name2*
- `*name` - writes no image file, but creates a frame with this name

Then a special form of the `#UDGARRAY` macro is used to create the animated image from a set of frames:

```
#UDGARRAY*FRAME1[;FRAME2;...](fname)
```

`FRAME1`, `FRAME2` etc. are frame specifications; each one has the form:

```
name[,delay]
```

- `name` is the name of the frame
- `delay` is the delay between this frame and the next in 1/100ths of a second; it also sets the default delay for any frames that follow (default: 32)

For example:

```
; Sprite animation frames
;
; #UDGTABLE {
; #UDGARRAY2;64000-64024-8(sprite1*) |
; #UDGARRAY2;64032-64056-8(sprite2*) |
; #UDGARRAY2;64064-64088-8(sprite3*) |
; #UDGARRAY*sprite1,50;sprite2;sprite3(sprite.gif)
; } TABLE#
```

The first three `#UDGARRAY` macros create the required frames (and write images of them); the last `#UDGARRAY` macro combines the three frames into a single animated image, with a delay of 0.5s between each frame.

Version	Changes
2.0.5	New
2.2.5	Added support for masks
2.3.1	Added the `flip` parameter
2.4	Added the `rotate` parameter
3.0	Added image-cropping capabilities
3.1.1	Added support for UDG address ranges with horizontal and vertical steps
3.6	Added support for creating an animated image from an arbitrary sequence of frames

8.5.23 #UDGTABLE

The `#UDGTABLE` macro behaves in exactly the same way as the `#TABLE` macro, except that the resulting table will not be rendered in ASM mode. Its intended use is to contain images that should be rendered in HTML mode only.

See *#TABLE*, and also *#HTML*.

8.6 Ref files

If you want to configure or augment an HTML disassembly, you will need one or more *ref* files. A *ref* file can be used to (for example):

- add a 'Bugs' page on which bugs are documented
- add a 'Trivia' page on which interesting facts are documented
- add a 'Pokes' page on which useful POKEs are listed
- add a 'Changelog' page
- add a 'Glossary' page
- add a 'Graphic glitches' page
- add any other kind of custom page
- change the title of the disassembly
- define the layout of the disassembly index page
- define the link text and titles for the various pages in the disassembly
- define the location of the files and directories in the disassembly
- define the colours used when creating images

A *ref* file must be formatted into sections separated by section names inside square brackets, like this:

```
[SectionName]
```

The contents of each section that may be found in a *ref* file are described below.

8.6.1 [Bug:*:*]

Each `Bug:*:*` section defines an entry on the 'Bugs' page. The section names and contents take the form:

```
[Bug:anchor:title]
First paragraph.

Second paragraph.

...
```

where:

- `anchor` is the name of the HTML anchor for the entry
- `title` is the title of the entry

To ensure that an entry can be linked to by the *#BUG* macro, the anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

Paragraphs should be separated by blank lines, and may contain HTML markup and *skool macros*.

8.6.2 [Changelog:*]

Each `Changelog:*` section defines an entry on the 'Changelog' page. The section names and contents take the form:

```
[Changelog:title]
Intro text.

First top-level item.
  First subitem.
  Second subitem.
    First subsubitem.

Second top-level item.
...
```

where `title` is the title of the entry, and the intro text and top-level items are separated by blank lines. Lower-level items are created by using indentation, as shown.

If the intro text is a single hyphen (-), it will not be included in the final HTML rendering.

The intro text and changelog items may contain HTML markup and *skool macros*.

Version	Changes
2.2.5	New

8.6.3 [Colours]

The `Colours` section contains colour definitions that will be used when creating images. Each line has the form:

```
name=R,G,B
```

or:

```
name=#RGB
```

where:

- `name` is the colour name
- `R,G,B` is a decimal RGB triplet
- `#RGB` is a hexadecimal RGB triplet (in the usual 6-digit form, or in the short 3-digit form)

Recognised colour names and their default RGB values are:

- `TRANSPARENT`: 0,254,0 (#00fe00)
- `BLACK`: 0,0,0 (#000000)
- `BLUE`: 0,0,197 (#0000c5)
- `RED`: 197,0,0 (#c50000)
- `MAGENTA`: 197,0,197 (#c500c5)
- `GREEN`: 0,198,0 (#00c600)
- `CYAN`: 0,198,197 (#00c6c5)
- `YELLOW`: 197,198,0 (#c5c600)
- `WHITE`: 205,198,205 (#cdc6cd)
- `BRIGHT_BLUE`: 0,0,255 (#0000ff)
- `BRIGHT_RED`: 255,0,0 (#ff0000)
- `BRIGHT_MAGENTA`: 255,0,255 (#ff00ff)
- `BRIGHT_GREEN`: 0,255,0 (#00ff00)
- `BRIGHT_CYAN`: 0,255,255 (#00ffff)
- `BRIGHT_YELLOW`: 255,255,0 (#ffff00)
- `BRIGHT_WHITE`: 255,255,255 (#ffffff)

Version	Changes
2.0.5	New
3.4	Added support for hexadecimal RGB triplets

8.6.4 [Config]

The `Config` section contains configuration parameters in the format:

```
name=value
```

Recognised parameters are:

- `GameDir` - the root directory of the game's HTML disassembly; if not specified, the base name of the *skool* or *ref* file given on the *skool2html.py* command line will be used

- `HtmlWriterClass` - the name of the Python class to use for writing the HTML disassembly of the game (default: `skoolkit.skoolhtml.HtmlWriter`); if the class is in a module that is not in the module search path (e.g. a standalone module that is not part of an installed package), the module's location may be specified thus: `/path/to/moduledir:module.classname`
- `SkoolFile` - the name of the main *skool* file to use if not given on the *skool2html.py* command line; if not specified, the *skool* file with the same base name as the *ref* file will be used

For information on how to create your own Python class for writing an HTML disassembly, see the documentation on *extending SkoolKit*.

Version	Changes
2.0	New
2.2.3	Added the `HtmlWriterClass` parameter
3.3.1	Added support to the `HtmlWriterClass` parameter for specifying a module outside the module search path

8.6.5 [Fact:*:*]

Each `Fact:*:*` section defines an entry on the 'Trivia' page. The section names and contents take the form:

```
[Fact:anchor:title]
First paragraph.

Second paragraph.

...
```

where:

- `anchor` is the name of the HTML anchor for the entry
- `title` is the title of the entry

To ensure that an entry can be linked to by the *#FACT* macro, the anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

Paragraphs should be separated by blank lines, and may contain HTML markup and *skool macros*.

8.6.6 [Game]

The `Game` section contains configuration parameters that control certain aspects of the HTML output. The parameters are in the format:

```
name=value
```

Recognised parameters are:

- `Font` - the base name of the font file to use (default: None); multiple font files can be declared by separating their names with semicolons
- `Game` - the name of the game, which appears in the title of every page, and also in the header of every page (if no logo is defined); if not specified, the base name of the *skool* file is used
- `GameStatusBufferIncludes` - a comma-separated list of addresses of entries to include on the 'Game status buffer' page in addition to those that are marked with a `g` (see the *skool file format reference*)
- `InputRegisterTableHeader` - the text to use in the header of input register tables on routine disassembly pages; if not specified, no header is displayed

- `LinkOperands` - a comma-separated list of instruction types whose operands should be hyperlinked when possible (default: `CALL,DEFW,DJNZ,JP,JR`); add `LD` to the list to enable the address operands of LD instructions to be hyperlinked as well
- `Logo` - the text/HTML that will serve as the game logo in the header of every page (typically a skool macro that creates a suitable image); if not specified, `LogoImage` is used
- `LogoImage` - the path to the game logo image, which appears in the header of every page; if the specified file does not exist, the name of the game is used in place of an image
- `OutputRegisterTableHeader` - the text to use in the header of output register tables on routine disassembly pages; if not specified, no header is displayed
- `StyleSheet` - the base name of the CSS file to use (default: *skoolkit.css*); multiple CSS files can be declared by separating their names with semicolons
- `TitlePrefix` - the prefix to use before the game name or logo in the header of the main index page (default: 'The complete')
- `TitleSuffix` - the suffix to use after the game name or logo in the header of the main index page (default: 'RAM disassembly')

Version	Changes
2.0.3	Added the `GameStatusBufferIncludes` parameter
2.0.5	Added the `Logo` parameter
3.1.2	Added the `InputRegisterTableHeader` and `OutputRegisterTableHeader` parameters
3.4	Added the `LinkOperands` parameter
3.5	Added the `Font`, `LogoImage` and `StyleSheet` parameters (all of which used to live in the *[Paths]* section, `LogoImage` by the name `Logo`)

8.6.7 [Glossary:*]

Each `Glossary:*` section defines an entry on the 'Glossary' page. The section names and contents take the form:

```
[Glossary:term]
First paragraph.

Second paragraph.

...
```

where `term` is the term being defined in the entry.

Paragraphs should be separated by blank lines, and may contain HTML markup and *skool macros*.

Version	Changes
3.1.3	Added support for multiple paragraphs

8.6.8 [GraphicGlitch:*:*]

Each `GraphicGlitch:*:*` section defines an entry on the 'Graphic glitches' page. The section names and contents take the form:

```
[GraphicGlitch:anchor:title]
First paragraph.

Second paragraph.
```

```
...
```

where:

- `anchor` is the name of the HTML anchor for the entry
- `title` is the title of the entry

Paragraphs should be separated by blank lines, and may contain HTML markup and *skool macros*.

8.6.9 [Graphics]

The `Graphics` section, if present, defines the body of the 'Graphics' page; it may contain HTML markup and *skool macros*.

Version	Changes
2.0.5	New

8.6.10 [ImageWriter]

The `ImageWriter` section contains configuration parameters that control SkoolKit's image creation library. The parameters are in the format:

```
name=value
```

Recognised parameters are:

- `DefaultFormat` - the default image format; valid values are `png` (the default) and `gif`
- `GIFCompression` - `1` to create compressed GIFs (which is slower but produces much smaller files), or `0` to create uncompressed GIFs (default: `1`);
- `GIFEnableAnimation` - `1` to create animated GIFs for images that contain flashing cells, or `0` to create plain (unanimated) GIFs for such images (default: `1`)
- `GIFTransparency` - `1` to make the `TRANSPARENT` colour (see *[Colours]*) in GIF images transparent, or `0` to make it opaque (default: `0`)
- `PNGAlpha` - the alpha value to use for the `TRANSPARENT` colour (see *[Colours]*) in PNG images; valid values are in the range 0-255, where 0 means fully transparent, and 255 means fully opaque (default: `255`)
- `PNGCompressionLevel` - the compression level to use for PNG image data; valid values are in the range 0-9, where 0 means no compression, 1 is the lowest compression level, and 9 is the highest (default: `9`)
- `PNGEnableAnimation` - `1` to create animated PNGs (in APNG format) for images that contain flashing cells, or `0` to create plain (unanimated) PNG files for such images (default: `1`)

The image-creating skool macros will create a file in the default image format if the filename is unspecified, or its suffix is omitted, or its suffix is neither `.png` nor `.gif`. For example, if `DefaultFormat` is `png`, then:

```
#FONT32768,26
```

will create an image file named `font.png`. To create a GIF instead (regardless of the default image format):

```
#FONT32768,26(font.gif)
```

For images that contain flashing cells, animated GIFs are recommended over animated PNGs in APNG format, because they are more widely supported in web browsers.

Version	Changes
3.0 3.0.1	New Added the `DefaultFormat`, `GIFCompression`, `GIFEnableAnimation`, `GIFTransparency`, `PNGAlpha` and `PNGEnableAnimation` parameters

8.6.11 [Index]

The `Index` section contains a list of link group IDs in the order in which the link groups should appear on the disassembly index page. The link groups themselves are defined in `[Index:*:*]` sections (see below).

By default, SkoolKit defines the following list of link groups:

```
[Index]
MemoryMaps
Graphics
DataTables
OtherCode
Reference
```

Version	Changes
2.0.5	New

8.6.12 [Index:*:*]

Each `Index:*:*` section defines a link group (a group of links on the disassembly home page). The section names and contents take the form:

```
[Index:groupID:text]
Page1ID
Page2ID
...
```

where:

- `groupID` is the link group ID (as may be declared in the *[Index]* section)
- `text` is the text of the link group header
- `Page1ID`, `Page2ID` etc. are the IDs of the pages that will appear in the link group

The page IDs that may be used in an `[Index:*:*]` section are the same as the file IDs that may be used in the *[Paths]* section, or the IDs defined by *[Page:*]* sections.

By default, SkoolKit defines four link groups with the following names and contents:

```
[Index:MemoryMaps:Memory maps]
MemoryMap
RoutinesMap
DataMap
MessagesMap
UnusedMap

[Index:Graphics:Graphics]
Graphics
GraphicGlitches

[Index:DataTables:Data tables and buffers]
GameStatusBuffer
```

```
[Index:Reference:Reference]
Changelog
Glossary
Facts
Bugs
Pokes
```

Version	Changes
2.0.5	New

8.6.13 [Info]

The `Info` section contains parameters that define the release and copyright information that appears in the footer of every page of the HTML disassembly. Each line has the form:

```
name=text
```

Recognised parameters are:

- `Copyright` - copyright message (default: '')
- `Created` - message indicating the software used to create the disassembly (default: 'Created using SkoolKit $VERSION.')
- `Release` - message indicating the release name and version number of the disassembly (default: '')

If the string `$VERSION` appears anywhere in the `Created` message, it is replaced by the version number of SkoolKit.

Each of these messages may contain HTML markup.

Version	Changes
2.0	New
2.0.3	Added the `Created` parameter
2.2.5	Set the default value for the `Created` parameter

8.6.14 [Links]

The `Links` section defines the link text for the various pages in the HTML disassembly (as displayed on the disassembly index page). Each line has the form:

```
ID=text
```

where:

- `ID` is the ID of the page
- `text` is the link text

Recognised page IDs are:

- `Bugs` - the 'Bugs' page
- `Changelog` - the 'Changelog' page
- `DataMap` - the 'Data' memory map page
- `Facts` - the 'Trivia' page
- `GameStatusBuffer` - the 'Game status buffer' page
- `Glossary` - the 'Glossary' page

- `GraphicGlitches` - the 'Graphic glitches' page
- `Graphics` - the 'Graphics' page
- `MemoryMap` - the 'Everything' memory map page (default: 'Everything')
- `MessagesMap` - the 'Messages' memory map page
- `Pokes` - the 'Pokes' page
- `RoutinesMap` - the 'Routines' memory map page
- `UnusedMap` - the 'Unused addresses' memory map page

The default link text for a page is the same as the page title (see *[Titles]*) except where indicated above.

If the link text starts with some text in square brackets, that text alone is used as the link text, and the remaining text is displayed alongside the hyperlink. For example:

```
MemoryMap=[Everything] (routines, data, text and unused addresses)
```

This declares that the link text for the 'Everything' memory map page will be 'Everything', and '(routines, data, text and unused addresses)' will be displayed alongside it.

Version	Changes
2.0.5	New
2.2.5	Added the `Changelog` page ID
2.5	Added the `UnusedMap` page ID

8.6.15 [MemoryMap:*]

Each `MemoryMap:*` section defines the properties of a memory map page. The section names take the form:

```
[MemoryMap:PageID]
```

where `PageID` is the unique ID of the memory map page (which should be the same as the corresponding page ID that appears in the *[Paths]* section).

Each `MemoryMap:*` section contains parameters in the form:

```
name=value
```

Recognised parameters and their default values are:

- `EntryTypes` - the types of entries to show in the map (by default, every type is shown); entry types are identified by their control directives as follows:
  - `b` - DEFB blocks
  - `c` - routines
  - `g` - game status buffer entries
  - `t` - messages
  - `u` - unused addresses
  - `w` - DEFW blocks
  - `z` - blocks containing all zeroes
- `Intro` - the text (HTML) to display at the top of the memory map page (default: '')
- `PageByteColumns` - `1` if the memory map page should include 'Page' and 'Byte' columns, `0` otherwise (default: `0`)

- `Write` - `1` if the memory map page should be written, `0` otherwise (default: `1`)

By default, SkoolKit defines five memory maps whose property values differ from the defaults as follows:

```
[MemoryMap:MemoryMap]
PageByteColumns=1

[MemoryMap:RoutinesMap]
EntryTypes=c

[MemoryMap:DataMap]
EntryTypes=bw
PageByteColumns=1

[MemoryMap:MessagesMap]
EntryTypes=t

[MemoryMap:UnusedMap]
EntryTypes=uz
PageByteColumns=1
```

Version	Changes
2.5	New

8.6.16 [OtherCode:*]

Each `OtherCode:*` section defines a secondary disassembly that will appear under 'Other code' on the main disassembly home page. The section names take the form:

```
[OtherCode:asm_id]
```

where `asm_id` is a unique ID for the secondary disassembly; it must be limited to the characters '$', '#', 0-9, A-Z and a-z. The unique ID may be used by the *#R* macro when referring to routines or data blocks in the secondary disassembly from another disassembly.

Each `OtherCode:*` section contains parameters in the form:

```
name=value
```

The following parameters are required:

- `Header` - the header text that will appear on each routine or data block disassembly page in the secondary disassembly
- `Index` - the filename of the home page of the secondary disassembly
- `Path` - the directory to which the secondary disassembly files will be written
- `Source` - the *skool* file from which to generate the secondary disassembly
- `Title` - the header text that will appear on the the secondary disassembly index page

The following parameters are optional:

- `IndexPageId` - the ID of the secondary disassembly index page; if defined, it can be used by the *#LINK* macro to create a hyperlink to the page
- `Link` - the link text to use on the main disassembly index page for the hyperlink to the secondary disassembly index page (defaults to the value of the `Title` parameter)

Version	Changes
2.0	New
2.2.5	Added the `IndexPageId` and `Link` parameters

8.6.17 [Page:*]

Each `Page:*` section is used to either declare a page that already exists, or define a custom page in the HTML disassembly (in conjunction with a corresponding *[PageContent:*]* section). The section names take the form:

```
[Page:PageId]
```

where `PageId` is a unique ID for the page. The unique ID may be used in an *[Index:*:*]* section to create a link to the page in the disassembly index.

Each `Page:*` section contains parameters in the form:

```
name=value
```

One of the following two parameters is required:

- `Content` - the path (directory and filename) of a page that already exists
- `Path` - the path (directory and filename) where the custom page will be created

The following parameters are optional:

- `BodyClass` - the CSS class to use for the `<body>` element of the page (default: no CSS class is used)
- `JavaScript` - the base name of the JavaScript file to use (default: None); multiple JavaScript files can be declared by separating their names with semicolons
- `Link` - the link text for the page (defaults to the title)
- `PageContent` - the HTML source of the body of the page; this may contain *skool macros*, and can be used instead of a *[PageContent:*]* section if the source can be written on a single line
- `Title` - the title of the page (defaults to the page ID)

Version	Changes
2.1	New
3.5	The `JavaScript` parameter specifies the JavaScript file(s) to use

8.6.18 [PageContent:*]

Each `PageContent:*` section contains the HTML source of the body of a custom page defined in a *[Page:*]* section. The section names take the form:

```
[PageContent:PageId]
```

where `PageId` is the unique ID of the page (as previously declared in the name of the corresponding *[Page:*]* section).

The HTML source may contain *skool macros*.

Version	Changes
2.1	New

8.6.19 [Paths]

The `Paths` section defines the locations of the files and directories in the HTML disassembly. Each line has the form:

```
ID=path
```

where:

- `ID` is the ID of the file or directory
- `path` is the path of the file or directory relative to the root directory of the disassembly

Recognised file IDs and their default paths are:

- `Bugs` - the 'Bugs' page (default: *reference/bugs.html*)
- `Changelog` - the 'Changelog' page (default: *reference/changelog.html*)
- `DataMap` - the 'Data' memory map page (default: *maps/data.html*)
- `Facts` - the 'Trivia' page (default: *reference/facts.html*)
- `GameIndex` - the disassembly home page (default: *index.html*)
- `GameStatusBuffer` - the 'Game status buffer' page (default: *buffers/gbuffer.html*)
- `Glossary` - the 'Glossary' page (default: *reference/glossary.html*)
- `GraphicGlitches` - the 'Graphic glitches' page (default: *graphics/glitches.html*)
- `Graphics` - the 'Graphics' page (default: *graphics/graphics.html*)
- `MemoryMap` - the 'Everything' memory map page (default: *maps/all.html*)
- `MessagesMap` - the 'Messages' memory map page (default: *maps/messages.html*)
- `Pokes` - the 'Pokes' page (default: *reference/pokes.html*)
- `RoutinesMap` - the 'Routines' memory map page (default: *maps/routines.html*)
- `UnusedMap` - the 'Unused addresses' memory map page (default: *maps/unused.html*)

Recognised directory IDs and their default paths are:

- `CodePath` - the directory in which the disassembly files will be written (default: *asm*)
- `FontImagePath` - the directory in which font images (created by the *#FONT* macro) will be placed (default: *images/font*)
- `FontPath` - the directory in which to store font files specified by the `Font` parameter in the *[Game]* section (default: .)
- `JavaScriptPath` - the directory in which to store JavaScript files specified by the `JavaScript` parameter in *[Page:*]* sections (default: .)
- `ScreenshotImagePath` - the directory in which screenshot images (created by the *#SCR* macro) will be placed (default: *images/scr*)
- `StyleSheetPath` - the directory in which to store CSS files specified by the `StyleSheet` parameter in the *[Game]* section (default: .)
- `UDGImagePath` - the directory in which UDG images (created by the *#UDG* or *#UDGARRAY* macro) will be placed (default: *images/udgs*)

Version	Changes
2.0	New
2.0.5	Added the `FontImagePath` directory ID
2.1.1	Added the `CodePath` directory ID
2.2.5	Added the `Changelog` file ID
2.5	Added the `UnusedMap` file ID
3.1.1	Added the `FontPath` directory ID

8.6.20 [Poke:*:*]

Each `Poke:*:*` section defines an entry on the 'Pokes' page. The section names and contents take the form:

```
[Poke:anchor:title]
First paragraph.

Second paragraph.

...
```

where:

- `anchor` is the name of the HTML anchor for the entry
- `title` is the title of the entry

To ensure that an entry can be linked to by the *#POKE* macro, the anchor name must be limited to the characters '$', '#', 0-9, A-Z and a-z.

Paragraphs should be separated by blank lines, and may contain HTML markup and *skool macros*.

8.6.21 [Resources]

The `Resources` section lists files that will be copied into the disassembly build directory when *skool2html.py* is run. Each line has the form:

```
fname=destDir
```

where:

- `fname` is the name of the file to copy
- `destDir` is the destination directory, relative to the root directory of the disassembly; the directory will be created if it doesn't already exist

The files to be copied must be present in *skool2html.py*'s search path in order for it to find them; to see the search path, run `skool2html.py -s`.

If your disassembly requires pre-built images or other resources that SkoolKit does not build, listing them in this section ensures that they will be copied into place whenever the disassembly is built.

Version	Changes
3.6	New

8.6.22 [Titles]

The `Titles` section defines the titles of the various pages in the HTML disassembly. Each line has the form:

```
ID=title
```

where:

- `ID` is the ID of the page
- `title` is the page title

Recognised page IDs and their default titles are:

- `Bugs` - the 'Bugs' page (default: 'Bugs')
- `Changelog` - the 'Changelog' page (default: 'Changelog')
- `DataMap` - the 'Data' memory map page (default: 'Data')
- `Facts` - the 'Trivia' page (default: 'Trivia')
- `GameIndex` - the disassembly index page (default: 'Index')
- `GameStatusBuffer` - the 'Game status buffer' page (default: 'Game status buffer')
- `Glossary` - the 'Glossary' page (default: 'Glossary')
- `GraphicGlitches` - the 'Graphic glitches' page (default: 'Graphic glitches')
- `Graphics` - the 'Graphics' page (default: 'Graphics')
- `MemoryMap` - the 'Everything' memory map page (default: 'Memory map')
- `MessagesMap` - the 'Messages' memory map page (default: 'Messages')
- `Pokes` - the 'Pokes' page (default: 'Pokes')
- `RoutinesMap` - the 'Routines' memory map page (default: 'Routines')
- `UnusedMap` - the 'Unused addresses' memory map page (default: 'Unused addresses')

Version	Changes
2.0.5	New
2.2.5	Added the `Changelog` page ID
2.5	Added the `UnusedMap` page ID

8.7 ASM modes and directives

A *skool* file may contain directives that are processed during the parsing phase. Exactly how a directive is processed (and whether it is executed) depends on the 'substitution mode' and 'bugfix mode' in which the *skool* file is being parsed.

8.7.1 Substitution modes

There are three substitution modes: `@isub`, `@ssub`, and `@rsub`. These modes are described in the following subsections.

@isub mode

In `@isub` mode, `@isub` directives are executed, but `@ssub`, and `@rsub` directives are not. The main purpose of `@isub` mode is to make the minimum number of instruction substitutions necessary to produce an ASM file that assembles.

For example:

```
; @isub=LD A,(32512)
 25396 LD A,(m)
```

This `@isub` directive ensures that `LD A,(m)` is replaced by the valid instruction `LD A,(32512)` when rendering in ASM mode.

`@isub` mode is invoked by default when running *skool2asm.py*.

@ssub mode

In `@ssub` mode, `@isub` and `@ssub` directives are executed, but `@rsub` directives are not. The main purpose of `@ssub` mode is to replace LSBs, MSBs and full addresses in the operands of instructions with labels, to make the code amenable to some degree of relocation, but without actually removing or inserting any code.

For example:

```
; @ssub=LD (27015+1),A
*27012 LD (27016),A  ; Change the instruction below from SET 0,B to RES 0,B
                     ; or vice versa
 27015 SET 0,B
```

This `@ssub` directive replaces `LD (27016),A` with `LD (27015+1),A`; the `27015` will be replaced by the label for that address before rendering. (`27016` cannot be replaced by a label, since it is not the address of an instruction.)

`@ssub` mode is invoked by passing the `-s` option to *skool2asm.py*.

@rsub mode

In `@rsub` mode, `@isub`, `@ssub` and `@rsub` directives are executed. The main purpose of `@rsub` mode is to make code unconditionally relocatable, even if that requires the removal of existing code or the insertion of new code.

For example:

```
 23997 LD HL,32766
; @ssub=LD (HL),24002%256
 24000 LD (HL),194
; @rsub+begin
       INC L
       LD (HL),24002/256
; @rsub+end
 24002 XOR A
```

This `@rsub` block directive inserts two instructions that ensure that the address stored at 32766 will have the correct MSB as well as the correct LSB, regardless of where the code originally at 24002 now lives.

`@rsub` mode is invoked by passing the `-r` option to *skool2asm.py*. `@rsub` mode also implies `@ofix` mode; see below for a description of `@ofix` mode and the other bugfix modes.

8.7.2 Bugfix modes

There are three bugfix modes: `@ofix`, `@bfix` and `@rfix`. These modes are described in the following subsections.

@ofix mode

In `@ofix` mode, `@ofix` directives are executed, but `@bfix` and `@rfix` directives are not. The main purpose of `@ofix` mode is to fix instructions that have faulty operands.

For example:

```
; @ofix-begin
 27872 CALL 27633    ; This should be CALL 27634
; @ofix+else
       CALL 27634
; @ofix+end
```

These `@ofix` block directives fix the faulty operand of the CALL instruction.

`@ofix` mode is invoked by passing the `-f 1` option to *skool2asm.py*.

@bfix mode

In `@bfix` mode, `@ofix` and `@bfix` directives are executed, but `@rfix` directives are not. The main purpose of `@bfix` mode is to fix bugs by replacing instructions, but without changing the start address of any routines, routine entry points, or data blocks.

For example:

```
; @bfix-begin
 32205 JR Z,32232    ; This should be JR NZ,32232
; @bfix+else
       JR NZ,32232   ;
; @bfix+end
```

`@bfix` mode is invoked by passing the `-f 2` option to *skool2asm.py*.

@rfix mode

In `@rfix` mode, `@ofix`, `@bfix` and `@rfix` directives are executed. The purpose of `@rfix` mode is to fix bugs that cannot be fixed without moving code around (to make space for the fix).

For example:

```
 28432 DEC HL
; @rsub+begin
       LD A,H
       OR L
; @rsub+end
 28433 JP Z,29712
```

These `@rfix` block directives insert some instructions to fix the faulty check on whether HL holds 0.

`@rfix` mode is invoked by passing the `-f 3` option to *skool2asm.py*. `@rfix` mode implies `@rsub` mode (see *@rsub mode*).

8.7.3 ASM directives

The ASM directives recognised by SkoolKit are described in the following subsections.

@bfix

The `@bfix` directive makes an instruction substitution in `@bfix` mode.

```
; @bfix=INSTRUCTION
```

- `INSTRUCTION` is the replacement instruction

For example:

```
; @bfix=DEFM "Phosphorus"
t57532 DEFM "Phosphorous"
```

@bfix block directives

The `@bfix` block directives define a block of lines that will be inserted or removed in `@bfix` mode.

The syntax for defining a block that will be inserted in `@bfix` mode (but left out otherwise) is:

```
; @bfix+begin
...                   ; Lines to be inserted
; @bfix+end
```

The syntax for defining a block that will be removed in `@bfix` mode (but left in otherwise) is:

```
; @bfix-begin
...                   ; Lines to be removed
; @bfix-end
```

Typically, though, it is desirable to define a block that will be removed in `@bfix` mode right next to the block that should be inserted in its place. That may be done thus:

```
; @bfix-begin
...                   ; Instructions to be removed
; @bfix+else
...                   ; Instructions to be inserted
; @bfix+end
```

which is equivalent to:

```
; @bfix-begin
...                   ; Instructions to be removed
; @bfix-end
; @bfix+begin
...                   ; Instructions to be inserted
; @bfix+end
```

For example:

```
; @bfix-begin
 32205 JR Z,32232     ; This should be JR NZ,32232
; @bfix+else
       JR NZ,32232    ;
; @bfix+end
```

@end

The `@end` directive may be used to indicate where to stop parsing the *skool* file for the purpose of generating ASM output. Everything after the `@end` directive is ignored.

```
; @end
```

See also *@start*.

Version	Changes
2.2.2	New

@ignoreua

The `@ignoreua` directive suppresses any warnings that would otherwise be reported concerning addresses not converted to labels in the comment that follows; the comment may be an entry title, an entry description, a mid-block comment, a block end comment, or an instruction-level comment.

```
; @ignoreua
```

To apply the directive to an entry title:

```
; @ignoreua
; Prepare data at 32768
;
; This routine operates on the data in page 128.
```

If the `@ignoreua` directive were not present, a warning would be printed (during the rendering phase) about the entry title containing an address (32768) that has not been converted to a label.

To apply the directive to an entry description:

```
; Prepare data in page 128
;
; @ignoreua
; This routine operates on the data at 32768.
```

If the `@ignoreua` directive were not present, a warning would be printed (during the rendering phase) about the entry description containing an address (32768) that has not been converted to a label.

To apply the directive to a mid-block comment:

```
 28913 LD L,A
; @ignoreua
; #REGhl now holds either 32522 or 32600.
 28914 LD B,(HL)
```

If the `@ignoreua` directive were not present, warnings would be printed (during the rendering phase) about the comment containing addresses (32522, 32600) that have not been converted to labels.

To apply the directive to a block end comment:

```
 44159 JP 63152
; @ignoreua
; This routine continues at 63152.
```

If the `@ignoreua` directive were not present, warnings would be printed (during the rendering phase) about the comment containing an address (63152) that has not been converted to a label.

To apply the directive to an instruction-level comment:

```
; @ignoreua
 60159 LD C,A        ; #REGbc now holds 62818
```

If the `@ignoreua` directive were not present, a warning would be printed (during the rendering phase) about the comment containing an address (62818) that has not been converted to a label.

Version	Changes
2.4.1	Added support for entry titles, entry descriptions, mid-block comments and block end comments

@isub

The `@isub` directive makes an instruction substitution in `@isub` mode.

```
; @isub=INSTRUCTION
```

- `INSTRUCTION` is the replacement instruction

For example:

```
; @isub=LD A,(32512)
 25396 LD A,(m)
```

This `@isub` directive ensures that `LD A,(m)` is replaced by the valid instruction `LD A,(32512)` when rendering in ASM mode.

@isub block directives

The `@isub` block directives define a block of lines that will be inserted or removed in `@isub` mode.

The syntax is equivalent to that for the *@bfix block directives*.

@keep

The `@keep` directive prevents the substitution of a label for the operand in the next instruction (but only when the instruction has not been replaced using an `@isub` or `@ssub` directive).

```
; @keep
```

For example:

```
; @keep
 28328 LD BC,24576   ; #REGb=96, #REGc=0
```

If the `@keep` directive were not present, the operand (24576) of the `LD BC` instruction would be replaced with the label of the routine at 24576 (if there is a routine at that address); however, the operand is meant to be a pure data value, not a variable or routine address.

@label

The `@label` directive sets the label for the next instruction.

```
; @label=LABEL
```

- `LABEL` is the label to apply

For example:

```
; @label=ENDGAME
c24576 XOR A
```

This sets the label for the routine at 24576 to `ENDGAME`.

@nolabel

The `@nolabel` directive prevents the next instruction from having a label automatically generated.

```
; @nolabel
```

For example:

```
; @label=TOGGLE
c48998 LD HL,32769
; @bfix+begin
; @label=LOOP
; @bfix+end
 49001 LD A,(HL)
; @bfix+begin
; @nolabel
; @bfix+end
*49002 XOR L
 49003 LD (HL),A
 49004 INC L
; @bfix-begin
 49005 JR NZ,49002
; @bfix+else
 49005 JR NZ,49001
; @bfix+end
```

The `@nolabel` directive here prevents the instruction at 49002 from being labelled in `@bfix` mode (because no label is required; instead, the previous instruction at 49001 will be labelled).

The output in `@bfix` mode will be:

```
TOGGLE:
  LD HL,32769
LOOP:
  LD A,(HL)
  XOR L
  LD (HL),A
  INC L
  JR NZ,LOOP
```

And the output when not in `@bfix` mode will be:

```
TOGGLE:
  LD HL,32769
  LD A,(HL)
TOGGLE_0:
  XOR L
  LD (HL),A
  INC L
  JR NZ,TOGGLE_0
```

@nowarn

The `@nowarn` directive suppresses any warnings that would otherwise be reported for the next instruction concerning:

- a `LD` operand being replaced with a routine label (if the instruction has not been replaced using `@isub` or `@ssub`)
- an operand not being replaced with a label (because the operand address has no label)

```
; @nowarn
```

For example:

```
; @nowarn
 25560 LD BC,25404   ; Point #REGbc at the routine at #R25404
```

If this `@nowarn` directive were not present, a warning would be printed (during the parsing phase) about the operand (25404) being replaced with a routine label (which would be inappropriate if 25404 were intended to be a pure data value).

For another example:

```
; @ofix-begin
; @nowarn
 27872 CALL 27633    ; This should be CALL #R27634
; @ofix+else
       CALL 27634    ;
; @ofix+end
```

If this `@nowarn` directive were not present, a warning would be printed (during the parsing phase, if not in `@ofix` mode) about the operand (27633) not being replaced with a label (usually you would want the operand of a CALL instruction to be replaced with a label, but not in this case).

@ofix

The `@ofix` directive makes an instruction substitution in `@ofix` mode.

```
; @ofix=INSTRUCTION
```

- `INSTRUCTION` is the replacement instruction (with a corrected operand)

For example:

```
; @ofix=JR NZ,26067
 25989 JR NZ,26068
```

This `@ofix` directive replaces the operand of the `JR NZ` instruction with 26067.

@ofix block directives

The `@ofix` block directives define a block of lines that will be inserted or removed in `@ofix` mode.

The syntax is equivalent to that for the *@bfix block directives*.

@org

The `@org` directive inserts an `ORG` assembler directive.

```
; @org=ADDRESS
```

- `ADDRESS` is the `ORG` address

@rem

The `@rem` directive may be used to make an illuminating comment about a nearby section or other ASM directive in a *skool* file. The directive is ignored by the parser.

```
; @rem=COMMENT
```

- `COMMENT` is a suitably illuminating comment

For example:

```
; @rem=The next section of data MUST start at 64000
; @org=64000
```

Version	Changes
2.4	The = is required

@rfix block directives

The `@rfix` block directives define a block of lines that will be inserted or removed in `@rfix` mode.

The syntax is equivalent to that for the *@bfix block directives*.

@rsub

The `@rsub` directive makes an instruction substitution in `@rsub` mode.

```
; @rsub=INSTRUCTION
```

- `INSTRUCTION` is the replacement instruction

For example:

```
; @rsub=INC BC
 30143 INC C         ; Point #REGbc at the next byte of data
```

@rsub block directives

The `@rsub` block directives define a block of lines that will be inserted or removed in `@rsub` mode.

The syntax is equivalent to that for the *@bfix block directives*.

@set

The `@set` directive sets a property on the ASM writer.

```
; @set-name=value
```

- `name` is the property name
- `value` is the property value

`@set` directives should be placed somewhere after the `@start` directive, and before the `@end` directive (if there is one).

Recognised property names and their default values are:

- `bullet` - the bullet character(s) to use for list items specified in a *#LIST* macro (default: `*`)

- `comment-width-min` - the minimum width of the instruction comment field (default: `10`)
- `crlf` - `1` to use CR+LF to terminate lines, or `0` to use the system default (default: `0`)
- `handle-unsupported-macros` - how to handle an unsupported macro: `1` to expand it to an empty string, or `0` to exit with an error (default: `0`)
- `indent` - the number of spaces by which to indent instructions (default: `2`)
- `instruction-width` - the width of the instruction field (default: `23`)
- `label-colons` - `1` to append a colon to labels, or `0` to leave labels unadorned (default: `1`)
- `line-width` - the maximum width of each line (default: `79`)
- `tab` - `1` to use a tab character to indent instructions, or `0` to use spaces (default: `0`)
- `warnings` - `1` to print any warnings that are produced while writing ASM output (after parsing the *skool* file), or `0` to suppress them (default: `1`)
- `wrap-column-width-min` - the minimum width of a wrappable table column (default: `10`)

For example:

```
; @set-bullet=+
```

This `@set` directive sets the bullet character to '+'.

Version	Changes
3.2	New
3.3.1	Added the *comment-width-min*, *indent*, *instruction-width*, *label-colons*, *line-width* and *warnings* properties
3.4	Added the *handle-unsupported-macros* and *wrap-column-width-min* properties

@ssub

The `@ssub` directive makes an instruction substitution in `@ssub` mode.

```
; @ssub=INSTRUCTION
```

- `INSTRUCTION` is the replacement instruction

For example:

```
; @ssub=LD (27015+1),A
*27012 LD (27016),A  ; Change the instruction below from SET 0,B to RES 0,B
                     ; or vice versa
 27015 SET 0,B
```

This `@ssub` directive replaces `LD (27016),A` with `LD (27015+1),A`; the `27015` will be replaced by the label for that address before rendering. (`27016` cannot be replaced by a label, since it is not the address of an instruction.)

@start

The `@start` directive must be used to indicate where to start parsing the *skool* file for the purpose of generating ASM output. Everything before the `@start` directive is ignored.

```
; @start
```

See also *@end*.

@writer

The `@writer` directive specifies the name of the Python class to use to generate ASM output. It should be placed somewhere after the `@start` directive, and before the `@end` directive (if there is one).

```
; @writer=package.module.classname
```

or:

```
; @writer=/path/to/moduledir:module.classname
```

The second of these forms may be used to specify a class in a module that is outside the module search path (e.g. a standalone module that is not part of an installed package).

The default ASM writer class is skoolkit.skoolasm.AsmWriter. For information on how to create your own Python class for generating ASM output, see the documentation on *extending SkoolKit*.

Version	Changes
3.1	New
3.3.1	Added support for specifying a module outside the module search path

CHAPTER 9

Developer reference

9.1 Extending SkoolKit

9.1.1 Extension modules

While creating a disassembly of a game, you may find that SkoolKit's suite of *skool macros* is inadequate for certain tasks. For example, the game might have large tile-based sprites that you want to create images of for the HTML disassembly, and composing long *#UDGARRAY* macros for them would be too tedious. Or you might want to insert a timestamp in the header of the ASM disassembly so that you (or others) can keep track of when your ASM files were written.

One way to solve these problems is to add custom methods that could be called by a *#CALL* macro. But where to add the methods? SkoolKit's core HTML-writing and ASM-writing classes are skoolkit.skoolhtml.HtmlWriter and skoolkit.skoolasm.AsmWriter, so you could add the methods to those classes. But a better way is to subclass HtmlWriter and AsmWriter in a separate extension module, and add the methods there; then that extension module can be easily used with different versions of SkoolKit, and shared with other people.

A minimal extension module would look like this:

```
# Extension module in the skoolkit package directory
from .skoolhtml import HtmlWriter
from .skoolasm import AsmWriter

class GameHtmlWriter(HtmlWriter):
    pass

class GameAsmWriter(AsmWriter):
    pass
```

The next step is to get SkoolKit to use the extension module for your game. First, place the extension module (let's call it *game.py*) in the *skoolkit* package directory; to locate this directory, run *skool2html.py* with the `-p` option:

```
$ skool2html.py -p
/usr/lib/python2.7/dist-packages/skoolkit
```

(The package directory may be different on your system.) With *game.py* in place, add the following line to the *[Config]* section of your disassembly's *ref* file:

```
HtmlWriterClass=skoolkit.game.GameHtmlWriter
```

If you don't have a *ref* file yet, create one (ideally named *game.ref*, assuming the *skool* file is *game.skool*); if the *ref* file doesn't have a `[Config]` section yet, add one.

Now whenever *skool2html.py* is run on your *skool* file (or *ref* file), SkoolKit will use the GameHtmlWriter class instead of the core HtmlWriter class.

To get *skool2asm.py* to use GameAsmWriter instead of the core AsmWriter class when it's run on your *skool* file, add the following *@writer* ASM directive somewhere after the `@start` directive, and before the `@end` directive (if there is one):

```
; @writer=skoolkit.game.GameAsmWriter
```

The *skoolkit* package directory is a reasonable place for an extension module, but it could be placed in another package, or somewhere else as a standalone module. For example, if you wanted to keep a standalone extension module in *~/.skoolkit*, it should look like this:

```
# Standalone extension module
from skoolkit.skoolhtml import HtmlWriter
from skoolkit.skoolasm import AsmWriter

class GameHtmlWriter(HtmlWriter):
    pass

class GameAsmWriter(AsmWriter):
    pass
```

Then, assuming the extension module is *game.py*, the `HtmlWriterClass` parameter should be set thus:

```
HtmlWriterClass=~/.skoolkit:game.GameHtmlWriter
```

and the `@writer` directive should be set thus:

```
; @writer=~/.skoolkit:game.GameAsmWriter
```

9.1.2 #CALL methods

Implementing a method that can be called by a *#CALL* macro is done by adding the method to the HtmlWriter or AsmWriter subclass in the extension module.

One thing to be aware of when adding a `#CALL` method to a subclass of HtmlWriter is that the method must accept an extra parameter in addition to those passed from the `#CALL` macro itself: *cwd*. This parameter is set to the current working directory of the file from which the `#CALL` macro is executed, which may be useful if the method needs to provide a hyperlink to some other part of the disassembly (as in the case where an image is being created).

Let's say your sprite-image-creating method will accept two parameters (in addition to *cwd*): *sprite_id* (the sprite identifier) and *fname* (the image filename). The method (let's call it *sprite*) would look something like this:

```
from .skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def sprite(self, cwd, sprite_id, fname):
        img_path = self.image_path(fname)
        if self.need_image(img_path):
            udgs = self.build_sprite(sprite_id)
            self.write_image(img_path, udgs)
        return self.img_element(cwd, img_path)
```

With this method (and an appropriate implementation of the *build_sprite* method) in place, it's possible to use a `#CALL` macro like this:

```
#UDGTABLE
{ #CALL:sprite(3,jumping) }
{ Sprite 3 (jumping) }
TABLE#
```

Adding a `#CALL` method to the AsmWriter subclass is equally simple. The timestamp-creating method (let's call it *timestamp*) would look something like this:

```
import time
from .skoolasm import AsmWriter

class GameAsmWriter(AsmWriter):
    def timestamp(self):
        return time.strftime("%a %d %b %Y %H:%M:%S %Z")
```

With this method in place, it's possible to use a `#CALL` macro like this:

```
; This ASM file was generated on #CALL:timestamp()
```

9.1.3 Skool macros

Another way to add a custom method is to implement it as a skool macro. The main differences between a skool macro and a `#CALL` method are:

- a `#CALL` macro's parameters are automatically evaluated and passed to the `#CALL` method; a skool macro's parameters must be parsed and evaluated manually (typically by using one or more of the *macro-parsing utility functions*)
- every optional parameter in a skool macro can be assigned a default value if omitted; in a `#CALL` method, only the optional arguments at the end can be assigned default values if omitted, whereas any others are set to *None*
- numeric parameters in a `#CALL` macro are automatically converted to numbers before being passed to the `#CALL` method; no automatic conversion is done on the parameters of a skool macro

In summary: a `#CALL` method is generally simpler to implement than a skool macro, but skool macros are more flexible.

Implementing a skool macro is done by adding a method named *expand_macroname* to the HtmlWriter or AsmWriter subclass in the extension module. So, to implement a `#SPRITE` or `#TIMESTAMP` macro, we would add a method named *expand_sprite* or *expand_timestamp*.

A skool macro method must accept either two or three parameters, depending on whether it is implemented on a subclass of AsmWriter or HtmlWriter:

- `text` - the text that contains the skool macro
- `index` - the index of the character after the last character of the macro name (that is, where to start looking for the macro's parameters)
- `cwd` - the current working directory of the file from which the macro is being executed; this parameter must be supported by skool macro methods on an HtmlWriter subclass

A skool macro method must return a 2-tuple of the form `(end, string)`, where `end` is the index of the character after the last character of the macro's parameter string, and `string` is the HTML or text to which the macro should be expanded.

The *expand_sprite* method on GameHtmlWriter may therefore look something like this:

```
from .skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    # #SPRITEspriteId[{X,Y,W,H}](fname)
    def expand_sprite(self, text, index, cwd):
        end, img_path, crop_rect, sprite_id = self.parse_image_params(text, index, 1)
        if self.need_image(img_path):
            udgs = self.build_sprite(sprite_id)
            self.write_image(img_path, udgs, crop_rect)
        return end, self.img_element(cwd, img_path)
```

With this method (and an appropriate implementation of the *build_sprite* method) in place, the `#SPRITE` macro might be used like this:

```
#UDGTABLE
{ #SPRITE3(jumping) }
{ Sprite 3 (jumping) }
TABLE#
```

The *expand_timestamp* method on GameAsmWriter would look something like this:

```
import time
from .skoolasm import AsmWriter

class GameAsmWriter(AsmWriter):
    def expand_timestamp(self, text, index):
        return index, time.strftime("%a %d %b %Y %H:%M:%S %Z")
```

9.1.4 Parsing skool macros

The skoolkit.skoolmacro module provides some utility functions that may be used to parse the parameters of a skool macro.

`skoolkit.skoolmacro.`**`parse_ints`**(*text*, *index*, *num*, *defaults=()*)

Parse a string of comma-separated integer parameters. The string will be parsed until either the end is reached, or an invalid character is encountered. The set of valid characters consists of the comma, '$', the digits 0-9, and the letters A-F and a-f.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **num** – The maximum number of parameters to parse.
- **defaults** – The default values of the optional parameters.

Returns

A list of the form `[end, value1, value2...]`, where:

- `end` is the index at which parsing terminated
- `value1`, `value2` etc. are the parameter values

`skoolkit.skoolmacro.`**`parse_params`**(*text*, *index*, *p_text=None*, *chars=''*, *except_chars=''*, *only_chars=''*)

Parse a string of the form `params[(p_text)]`. The parameter string `params` will be parsed until either the end is reached, or an invalid character is encountered. The default set of valid characters consists of '$', '#', the digits 0-9, and the letters A-Z and a-z.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **p_text** – The default value to use for text found in parentheses.
- **chars** – Characters to consider valid in addition to those in the default set.
- **except_chars** – If not empty, all characters except those in this string are considered valid.
- **only_chars** – If not empty, only the characters in this string are considered valid.

Returns

A 3-tuple of the form `(end, params, p_text)`, where:

- `end` is the index at which parsing terminated
- `params` is the parameter string
- `p_text` is the text found in parentheses (if any)

New in version 3.6: The *except_chars* and *only_chars* parameters.

HtmlWriter also provides a method for parsing the parameters of an image-creating skool macro.

`HtmlWriter.`**`parse_image_params`**(*text*, *index*, *num*, *defaults=()*, *path_id='UDGImagePath'*, *fname=''*, *chars=''*, *ints=None*)

Parse a string of the form `params[{X,Y,W,H}][(fname)]`. The parameter string `params` may contain comma-separated values and will be parsed until either the end is reached, or an invalid character is encountered. The default set of valid characters consists of the comma, '$', the digits 0-9, and the letters A-F and a-f.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **num** – The maximum number of parameters to parse.
- **defaults** – The default values of the optional parameters.
- **path_id** – The ID of the target directory for the image file (as defined in the *[Paths]* section of the *ref* file).
- **fname** – The default base name of the image file.
- **chars** – Characters to consider valid in addition to those in the default set.
- **ints** – A list of the indexes (0-based) of the parameters that must evaluate to an integer; if *None*, every parameter must evaluate to an integer.

Returns

A list of the form `[end, image_path, crop_rect, value1, value2...]`, where:

- `end` is the index at which parsing terminated
- `image_path` is either the full path of the image file (relative to the root directory of the disassembly) or `fname` (if *path_id* is blank or *None*)
- `crop_rect` is `(X, Y, W, H)`
- `value1`, `value2` etc. are the parameter values

Changed in version 3.6: If *path_id* is blank or *None*, `image_path` is equal to `fname`.

New in version 3.6: The *ints* parameter.

9.1.5 Parsing ref files

HtmlWriter provides some convenience methods for extracting text and data from *ref* files. These methods are described below.

`HtmlWriter.`**`get_section`**(*section_name*, *paragraphs=False*, *lines=False*)
Return the contents of a *ref* file section.

Parameters

- **section_name** – The section name.
- **paragraphs** – If *True*, return the contents as a list of paragraphs.
- **lines** – If *True*, return the contents (or each paragraph) as a list of lines; otherwise return the contents (or each paragraph) as a single string.

`HtmlWriter.`**`get_sections`**(*section_type*, *paragraphs=False*, *lines=False*)
Return a list of 2-tuples of the form `(suffix, contents)` or 3-tuples of the form `(infix, suffix, contents)` derived from *ref* file sections whose names start with *section_type* followed by a colon. `suffix` is the part of the section name that follows either the first colon (when there is only one) or the second colon (when there is more than one); `infix` is the part of the section name between the first and second colons (when there is more than one).

Parameters

- **section_type** – The section name prefix.
- **paragraphs** – If *True*, return the contents of each section as a list of paragraphs.
- **lines** – If *True*, return the contents (or each paragraph) of each section as a list of lines; otherwise return the contents (or each paragraph) as a single string.

`HtmlWriter.`**`get_dictionary`**(*section_name*)
Return a dictionary built from the contents of a *ref* file section. Each line in the section should be of the form `X=Y`.

`HtmlWriter.`**`get_dictionaries`**(*section_type*)
Return a list of 2-tuples of the form `(suffix, dict)` derived from *ref* file sections whose names start with *section_type* followed by a colon. `suffix` is the part of the section name that follows the first colon, and `dict` is a dictionary built from the contents of that section; each line in the section should be of the form `X=Y`.

9.1.6 Memory snapshots

The *snapshot* attribute on HtmlWriter and AsmWriter is a 65536-element list that is populated with the contents of any `DEFB`, `DEFM`, `DEFS` and `DEFW` statements in the *skool* file.

A simple `#PEEK` macro that expands to the value of the byte at a given address might be implemented by using *snapshot* like this:

```
from .skoolhtml import HtmlWriter
from .skoolasm import AsmWriter
from .skoolmacro import parse_ints

class GameHtmlWriter(HtmlWriter):
    # #PEEKaddress
```

```
    def expand_peek(self, text, index, cwd):
        end, address = parse_ints(text, index, 1)
        return end, str(self.snapshot[address])

class GameAsmWriter(AsmWriter):
    # #PEEKaddress
    def expand_peek(self, text, index):
        end, address = parse_ints(text, index, 1)
        return end, str(self.snapshot[address])
```

HtmlWriter also provides some methods for saving and restoring memory snapshots, which can be useful for temporarily changing graphic data or the contents of data tables. These methods are described below.

`HtmlWriter.`**`push_snapshot`**`(`*name=''*`)`
Save the current memory snapshot for later retrieval (by `pop_snapshot()`), and put a copy in its place.

Parameters **name** – An optional name for the snapshot.

`HtmlWriter.`**`pop_snapshot`**`()`
Discard the current memory snapshot and replace it with the one that was most recently saved (by `push_snapshot()`).

`HtmlWriter.`**`get_snapshot_name`**`()`
Return the name of the current memory snapshot.

9.1.7 Graphics

If you are going to implement custom image-creating `#CALL` methods or skool macros, you will need to make use of the skoolkit.skoolhtml.Udg class.

The Udg class represents an 8x8 graphic (8 bytes) with a single attribute byte, and an optional mask.

class `skoolkit.skoolhtml.`**`Udg`**`(`*attr*, *data*, *mask=None*`)`
Initialise the UDG.

Parameters

- **attr** – The attribute byte.
- **data** – The graphic data (sequence of 8 bytes).
- **mask** – The mask data (sequence of 8 bytes).

A simple `#INVERSE` macro that creates an inverse image of a UDG might be implemented like this:

```
from .skoolhtml import HtmlWriter, Udg
from .skoolmacro import parse_ints

class GameHtmlWriter(HtmlWriter):
    # #INVERSEaddress,attr
    def expand_inverse(self, text, index, cwd):
        end, address, attr = parse_ints(text, index, 2)
        img_path = self.image_path('inverse{0}_{1}'.format(address, attr))
        if self.need_image(img_path):
            udg_data = [b ^ 255 for b in self.snapshot[address:address + 8]]
            udg = Udg(attr, udg_data)
            self.write_image(img_path, [[udg]])
        return end, self.img_element(cwd, img_path)
```

The Udg class provides two methods for manipulating an 8x8 graphic: *flip* and *rotate*.

`Udg.`**`flip`**(*flip=1*)
Flip the UDG.

Parameters **flip** – 1 to flip horizontally, 2 to flip vertically, or 3 to flip horizontally and vertically.

`Udg.`**`rotate`**(*rotate=1*)
Rotate the UDG 90 degrees clockwise.

Parameters **rotate** – The number of rotations to perform.

If you are going to implement `#CALL` methods or skool macros that create animated images, you will need to make use of the skoolkit.skoolhtml.Frame class.

The Frame class represents a single frame of an animated image.

class `skoolkit.skoolhtml.`**`Frame`**(*udgs*, *scale=1*, *mask=False*, *x=0*, *y=0*, *width=None*, *height=None*, *delay=32*)
Create a frame of an animated image.

Parameters

- **udgs** – The two-dimensional array of tiles (instances of `Udg`) from which to build the frame.
- **scale** – The scale of the frame.
- **mask** – Whether to apply masks to the tiles in the frame.
- **x** – The x-coordinate of the top-left pixel to include in the frame.
- **y** – The y-coordinate of the top-left pixel to include in the frame.
- **width** – The width of the frame; if *None*, the maximum width (derived from *x* and width of the array of tiles) is used.
- **height** – The height of the frame; if *None*, the maximum height (derived from *y* and height of the array of tiles) is used.
- **delay** – The delay between this frame and the next in 1/100ths of a second.

New in version 3.6.

HtmlWriter provides the following image-related convenience methods.

`HtmlWriter.`**`image_path`**(*fname*, *path_id='UDGImagePath'*)
Return the full path of an image file relative to the root directory of the disassembly. If *fname* does not end with '.png' or '.gif', an appropriate suffix will be appended (depending on the default image format). If *fname* starts with a '/', it will be removed and the remainder returned (in which case *path_id* is ignored). If *fname* is blank, *None* is returned.

Parameters

- **fname** – The name of the image file.
- **path_id** – The ID of the target directory (as defined in the *[Paths]* section of the *ref* file).

`HtmlWriter.`**`need_image`**(*image_path*)
Return whether an image file needs to be created. This will be true only if the file doesn't already exist, or all images are being rebuilt. Well-behaved image-creating methods will call this to check whether an image file needs to be written, and thus avoid building an image when it is not necessary.

Parameters **image_path** – The full path of the image file relative to the root directory of the disassembly.

`HtmlWriter.`**`write_image`**(*image_path*, *udgs*, *crop_rect=()*, *scale=2*, *mask=False*)
Create an image and write it to a file.

Parameters

- **image_path** – The full path of the file to which to write the image (relative to the root directory of the disassembly).
- **udgs** – The two-dimensional array of tiles (instances of `Udg`) from which to build the image.
- **crop_rect** – The cropping rectangle, `(x, y, width, height)`, where `x` and `y` are the x- and y-coordinates of the top-left pixel to include in the final image, and `width` and `height` are the width and height of the final image.
- **scale** – The scale of the image.
- **mask** – Whether to apply masks to the tiles in the image.

`HtmlWriter.`**`write_animated_image`**(*image_path*, *frames*)
Create an animated image and write it to a file.

Parameters

- **image_path** – The full path of the file to which to write the image (relative to the root directory of the disassembly).
- **frames** – A list of the frames (instances of `Frame`) from which to build the image.

New in version 3.6.

`HtmlWriter.`**`img_element`**(*cwd*, *image_path*, *alt=None*)
Return an `<img .../>` element for an image file.

Parameters

- **cwd** – The current working directory (from which the relative path of the image file will be computed).
- **image_path** – The full path of the image file relative to the root directory of the disassembly.
- **alt** – The alt text to use for the image; if *None*, the base name of the image file (with the '.png' or '.gif' suffix removed) will be used.

`HtmlWriter.`**`screenshot`**(*x=0*, *y=0*, *w=32*, *h=24*, *df_addr=16384*, *af_addr=22528*)
Return a two-dimensional array of tiles (instances of `Udg`) built from the display file and attribute file of the current memory snapshot.

Parameters

- **x** – The x-coordinate of the top-left tile to include (0-31).
- **y** – The y-coordinate of the top-left tile to include (0-23).
- **w** – The width of the array (in tiles).
- **h** – The height of the array (in tiles).
- **df_addr** – The display file address to use.
- **af_addr** – The attribute file address to use.

`HtmlWriter.`**`flip_udgs`**(*udgs*, *flip=1*)
Flip a 2D array of UDGs (instances of `Udg`).

Parameters

- **udgs** – The array of UDGs.
- **flip** – 1 to flip horizontally, 2 to flip vertically, or 3 to flip horizontally and vertically.

`HtmlWriter.`**`rotate_udgs`**(*udgs*, *rotate=1*)
Rotate a 2D array of UDGs (instances of `Udg`) 90 degrees clockwise.

Parameters

- **udgs** – The array of UDGs.
- **rotate** – The number of rotations to perform.

9.1.8 HtmlWriter initialisation

If your HtmlWriter subclass needs to perform some initialisation tasks, such as creating instance variables, or parsing *ref* file sections, the place to do that is the *init()* method.

`HtmlWriter.init()`
Perform post-initialisation operations. This method is called after *__init__()* has completed. By default the method does nothing, but subclasses may override it.

For example:

```
from .skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def init(self):
        # Get character names from the ref file
        self.characters = self.get_dictionary('Characters')
```