SkoolKit Documentation

Release 8.8

Richard Dymond

Nov 19, 2022

CONTENTS

CHAPTER

ONE

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 assembly language. 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 version of the disassembly.

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

1.1 Features

With SkoolKit you can:

- use *sna2ctl.py* to generate a *control file* (an attempt to identify routines and data blocks by static analysis) from a snapshot (SNA, SZX or Z80) or raw memory file
- enable *sna2ctl.py* to generate a much better control file that more reliably distinguishes code from data by using a code execution map produced by an emulator
- use *sna2skool.py* along with this control file to produce a disassembly of a snapshot or raw memory file
- add annotations to this disassembly (or the control file) as you discover the purpose of each routine and data block
- use *skool2html.py* to convert a disassembly into a bunch of HTML files (with annotations in place, and the operands of CALL and JP instructions converted into hyperlinks)
- use *skool2asm.py* to convert a disassembly into an assembler source file (also with annotations in place)
- use *skool2ctl.py* to convert a disassembly back into a control file (with annotations retained)
- use *skool2bin.py* to convert a disassembly into a raw memory file
- use *tap2sna.py* to convert a TAP or TZX file into a 'pristine' Z80 snapshot
- use *snapinfo.py* to analyse a snapshot or raw memory file and list the BASIC program it contains, show register values, produce a call graph, find tile graphic data, find text, or find sequences of arbitrary byte values
- use *tapinfo.py* to analyse the blocks in a TAP or TZX file, and list the BASIC program it contains
- use *bin2tap.py* to convert a snapshot or raw memory file into a TAP file
- use *bin2sna.py* to convert a raw memory file into a Z80 snapshot
- use *snapmod.py* to modify the register values or memory contents in a Z80 snapshot

- use *sna2img.py* to convert graphic data in a disassembly, SCR file, snapshot or raw memory file into a PNG image

In an HTML disassembly produced by *skool2html.py* you can also:

- use the *image macros* to build still and animated PNG images from graphic data
- use the *#AUDIO* macro to build WAV files for sound effects and tunes
- use the *#R* macro in annotations to create hyperlinks between routines and data blocks that refer to each other
- use *[Bug:*]*, *[Fact:*]* and *[Poke:*]* sections in a ref file to neatly render lists of bugs, trivia and POKEs on separate pages

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

1.2 Authors

SkoolKit is developed and maintained by Richard Dymond, and contains contributions from Philip M Anderson.

1.3 Licence

1.4 48K ZX Spectrum ROM

A copy of the 48K ZX Spectrum ROM is included with SkoolKit (*skoolkit/resources/48.rom*). The copyright in this ROM is held by Amstrad, who have kindly given permission for it to be redistributed.

CHAPTER

TWO

INSTALLING AND USING SKOOLKIT

2.1 Requirements

SkoolKit requires Python 3.7+. 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 skoolkit.ca
- from PyPI by using pip
- from the PPA for Ubuntu
- from the copr repo for Fedora

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 following the instructions below.

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:

```
> python3 -m pip install .
```

This will install the SkoolKit command scripts in *C:\Python39\Scripts* (assuming you have installed Python in *C:\Python39*), which means you can run them from anywhere (assuming you have added *C:\Python39\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:

```
# python3 -m pip install .
```

This will 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, use a dot-slash prefix (e.g. `./some-script.py`) if the script is being run from the current working directory.

CHAPTER

THREE

DISASSEMBLY DIY

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

3.1 Getting started

The first thing to do is select a Spectrum game to disassemble. For the purpose of this discussion, we'll use Hungry Horace. To build a pristine snapshot of the game, run the following command in the directory where SkoolKit was unpacked:

```
$ tap2sna.py @examples/hungry_horace.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 *hungry_horace.z80*.)

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

```
$ sna2skool.py hungry_horace.z80 > hungry_horace.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, it would be best to stick with '.skool'.

Now take a look at *hungry_horace.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.

3.2 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 Hungry Horace might start like this:

```
b 16384 Loading screen
i 23296
c 24576 The game has just loaded
c 25167
...
```

This control file declares that there is:

- a data block at 16384 titled 'Loading screen'
- a block at 23296 that should be ignored
- a code block (routine) at 24576 titled 'The game has just loaded'
- another code block at 25167

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

3.3 A skeleton disassembly

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

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

This time, *hungry_horace.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, use the `--hex` option:

```
$ sna2skool.py --hex -c examples/hungry_horace.ctl hungry_horace.z80 > hungry_horace.
↪skool
```

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

```
$ skool2html.py hungry_horace.skool
```

Now open *hungry_horace/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 'hungry_horace' 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 Hungry Horace: *hungry_horace.ref*. To use it with the skool file we've just created:

```
$ skool2html.py hungry_horace.skool examples/hungry_horace.ref
```

Now the disassembly will sport a game logo image.

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

3.4 Generating a control file

If you are planning to create a disassembly of some game other than Hungry Horace, you will need to create your own control file. To get started, you can use *sna2ctl.py* to perform a rudimentary static code analysis of the snapshot file and generate a corresponding control file:

```
$ sna2ctl.py game.z80 > game.ctl
$ sna2skool.py -c 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*) to see which blocks have been incorrectly marked as text, data or code, and then edit the control file (*game.ctl*) accordingly.

To generate a better control file, you could use a code execution map produced by an emulator to tell *sna2ctl.py* where at least some of the code is in the snapshot. *sna2ctl.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:

```
$ sna2ctl.py -m game.map game.z80 > game.ctl
```

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, *sna2ctl.py* and *sna2skool.py* generate control files and skool files with addresses and instruction operands in decimal notation. If you prefer to work in hexadecimal, use the `--hex` option of each command to produce a hexadecimal control file and a hexadecimal skool file:

```
$ sna2ctl.py --hex game.z80 > game.ctl
$ sna2skool.py --hex -c game.ctl game.z80 > game.skool
```

3.5 Developing the disassembly

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 adding annotations that describe what the code does and what the data is for.

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.

As for where to write annotations, you now have a choice. You can add them either to the control file or to the skool file. The recommended approach, unless you are already familiar with the syntax of skool files, is to add annotations to the control file. The benefits of continuing to work on the control file are:

- its syntax is much simpler than that of the skool file
- you are never in danger of breaking the skool file, and potentially causing *skool2asm.py* and *skool2html.py* to fail
- if you ever need to modify how an address range is disassembled, it is usually as simple as replacing one letter (e.g. `c` for code) with another (e.g. `t` for text)

If you would rather edit the skool file, however, then it is highly recommended to do so only for the purpose of adding, removing or updating annotations. Don't be tempted to manually convert code to data, or vice versa. Unless extreme care is taken, doing so could easily result in a broken skool file that is very difficult to fix.

Annotating the code and data in a skool file is done by adding comments just as you would in a regular assembly language source file. For example, you might add a comment to the instruction at 26429 in *hungry_horace.skool* thus:

```
 26429 DEC A         ; 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. If you are working on the skool file, the best way to do this is to 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*, will contain your reorganised code and data blocks, and all the annotations you carefully added to *game.skool*.

3.6 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 *hungry_horace.ref*:

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

POKE 26429,0
```

Now run *skool2html.py* again:

```
$ skool2html.py hungry_horace.skool examples/hungry_horace.ref
```

Open *hungry_horace/index.html* and you will 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.

Add one `Poke`, `Bug` or `Fact` section for each poke, bug or trivia entry 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.

3.7 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*
- 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
```

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

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

3.8 Base switching

If you would like to build both decimal and hexadecimal versions of your disassembly in HTML format and have them link to each other, then one possible approach is to define a custom page footer that contains a link to the corresponding page in the alternative disassembly.

An example of such a page footer can be found in *examples/bases.ref*, and the required Python code that generates the appropriate link for each page can be found in *examples/bases.py*. To use *bases.ref* and *bases.py* with your disassembly, first place copies of them alongside your existing skool and ref files. Then:

```
$ skool2html.py -D -c Config/GameDir=html/dec -c Config/InitModule=:bases game.skool␣
↪bases.ref
$ skool2html.py -H -c Config/GameDir=html/hex -c Config/InitModule=:bases game.skool␣
↪bases.ref
```

The first command here builds the decimal version of the disassembly in the directory *html/dec*, and the second command builds the hexadecimal version in the directory *html/hex*. The footer of each page in the decimal version will contain a link to the corresponding page in the hexadecimal version, and vice versa.

CHAPTER

FOUR

COMMANDS

4.1 bin2sna.py

bin2sna.py converts a binary (raw memory) file into a Z80 snapshot. For example:

```
$ bin2sna.py game.bin
```

will create a file named *game.z80*. 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 values can be changed by passing options to *bin2sna.py*. Run it with no arguments to see the list of available options:

```
usage: bin2sna.py [options] file.bin [file.z80]

Convert a binary (raw memory) file into a Z80 snapshot. 'file.bin' may be a
regular file, or '-' for standard input. If 'file.z80' is not given, it
defaults to the name of the input file with '.bin' replaced by '.z80', or
'program.z80' if reading from standard input.

Options:
  -b BORDER, --border BORDER
                        Set the border colour (default: 7).
  -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).
  -P a[-b[-c]],[^+]v, --poke a[-b[-c]],[^+]v
                        POKE N,v for N in {a, a+c, a+2c..., b}. Prefix 'v'
                        with '^' to perform an XOR operation, or '+' to
                        perform an ADD operation. This option may be used
                        multiple times.
  -r name=value, --reg name=value
                        Set the value of a register. Do '--reg help' for more
                        information. This option may be used multiple times.
  -s START, --start START
                        Set the address at which to start execution (default:
                        ORG).
  -S name=value, --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.
```

Version	Changes
6.3	Added the `--poke` option
6.2	Added the `--reg` and `--state` options; the `--org`, `--stack` and `--start` options accept a hexadecimal integer prefixed by '0x'
5.2	New

4.2 bin2tap.py

bin2tap.py converts a binary (raw memory) file or a SNA, SZX or Z80 snapshot into a TAP file. 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 values 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 [file.tap]

Convert a binary (raw memory) file or a SNA, SZX or Z80 snapshot into a TAP
file. FILE may be a regular file, or '-' to read a binary file from standard
input.

Options:
  -b BEGIN, --begin BEGIN
                        Begin conversion at this address (default: ORG for a
                        binary file, 16384 for a snapshot).
  -c N, --clear N       Use a 'CLEAR N' command in the BASIC loader and leave
                        the stack pointer alone.
  -e END, --end END     End conversion at this address.
  -o ORG, --org ORG     Set the origin address for a binary file (default:
                        65536 minus the length of FILE).
  -p STACK, --stack STACK
                        Set the stack pointer (default: BEGIN).
  -s START, --start START
                        Set the start address to JP to (default: BEGIN).
  -S FILE, --screen FILE
                        Add a loading screen to the TAP file. FILE may be a
                        snapshot or a 6912-byte SCR file.
  -V, --version         Show SkoolKit version number and exit.
```

Note that the ROM tape loading routine at 1366 (0x0556) 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 (0x400E). 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.

If the input file contains a program that returns to BASIC, you should use the `--clear` option to add a CLEAR command to the BASIC loader. This option leaves the stack pointer alone, enabling the program to return to BASIC without crashing. The lowest usable address with the `--clear` option on a bare 48K Spectrum is 23952 (0x5D90).

Version	Changes
8.3	Added the `--begin` option; the `--end` option applies to raw memory files as well as snapshots
6.2	The `--clear`, `--end`, `--org`, `--stack` and `--start` options accept a hexadecimal integer prefixed by '0x'
5.3	Added the `--screen` option
5.2	Added the ability to read a binary file from standard input; added a second positional argument specifying the TAP filename
4.5	Added the `--clear` and `--end` options, and the ability to convert SNA, SZX and Z80 snapshots
3.4	Added the `-V` option and the long options
2.2.5	Added the `-p` option
1.3.1	New

4.3 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 and write it to standard output. FILE may
be a regular file, or '-' for standard input.

Options:
  -c, --create-labels   Create default labels for unlabelled instructions.
  -D, --decimal         Write the disassembly in decimal.
  -E ADDR, --end ADDR   Stop converting at this address.
  -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)
  -F, --force           Force conversion, ignoring @start and @end directives.
  -H, --hex             Write the disassembly in hexadecimal.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -l, --lower           Write the disassembly in lower case.
  -p, --package-dir     Show path to skoolkit package directory and exit.
  -P p=v, --set p=v     Set the value of ASM writer property 'p' to 'v'. This
                        option may be used multiple times.
  -q, --quiet           Be quiet.
  -r, --rsub            Apply safe substitutions (@ssub) and relocatability
                        substitutions (@rsub) (implies '-f 1').
  --show-config         Show configuration parameter values.
  -s, --ssub            Apply safe substitutions (@ssub).
  -S ADDR, --start ADDR
                        Start converting at this address.
  -u, --upper           Write the disassembly in upper case.
  --var name=value      Define a variable that can be used by @if and the SMPL
```

(continues on next page)

(continued from previous page)

```
                        macros. This option may be used multiple times.
  -V, --version         Show SkoolKit version number and exit.
  -w, --no-warnings     Suppress warnings.
  -W CLASS, --writer CLASS
                        Specify the ASM writer class to use.
```

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

See the *@set* directive for information on the ASM writer properties that can be set by the `--set` option.

4.3.1 Configuration

skool2asm.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `Address` - the format of the default link text for the *#R* macro when the target address has no label (default: ''); this format string recognises the replacement field `address`; if the format string is blank, the address is formatted exactly as it appears in the skool file (without any `$` prefix)
- `Base` - convert addresses and instruction operands to hexadecimal (`16`) or decimal (`10`), or leave them as they are (`0`, the default)
- `Case` - write the disassembly in lower case (`1`) or upper case (`2`), or leave it as it is (`0`, the default)
- `CreateLabels` - create default labels for unlabelled instructions (`1`), or don't (`0`, the default)
- `EntryLabel` - the format of the default label for the first instruction in a routine or data block (default: `L{address}`)
- `EntryPointLabel` - the format of the default label for an instruction other than the first in a routine or data block (default: `{main}_{index}`)
- `Quiet` - be quiet (`1`) or verbose (`0`, the default)
- `Set-property` - set an ASM writer property value, e.g. `Set-bullet=+` (see the *@set* directive for a list of available properties)
- `Templates` - file from which to read custom *ASM templates*
- `Warnings` - show warnings (`1`, the default), or suppress them (`0`)

`EntryLabel` and `EntryPointLabel` are standard Python format strings. `EntryLabel` recognises the following replacement fields:

- `address` - the address of the routine or data block as it appears in the skool file
- `location` - the address of the routine or data block as an integer

`EntryPointLabel` recognises the following replacement fields:

- `address` - the address of the instruction as it appears in the skool file
- `index` - 0 for the first unlabelled instruction in the routine or data block, 1 for the second, etc.
- `location` - the address of the instruction as an integer
- `main` - the label of the first instruction in the routine or data block

Configuration parameters must appear in a `[skool2asm]` section. For example, to make *skool2asm.py* write the disassembly in hexadecimal with a line width of 120 characters by default (without having to use the `-H` and `-P` options on the command line), add the following section to *skoolkit.ini*:

```
[skool2asm]
Base=16
Set-line-width=120
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
8.5	Added the `Address`, `EntryLabel` and `EntryPointLabel` configuration parameters
7.2	Added the `Templates` configuration parameter and support for *ASM templates*
7.0	*Non-entry blocks* are reproduced verbatim; added the `--force` option
6.4	Added the `--var` option
6.2	Added the `--show-config` option; the `--end` and `--start` options accept a hexadecimal integer prefixed by '0x'
6.1	Configuration is read from *skoolkit.ini* if present; added the `--ini` option
5.0	Added the `--set` option
4.5	Added the `--start` and `--end` options
4.1	Added the `--writer` option
3.4	Added the `-V` and `-p` options and the long options
2.2.2	Added the ability to read a skool file from standard input
2.1.1	Added the `-u`, `-D` and `-H` options
1.1	Added the `-c` option

4.4 skool2bin.py

skool2bin.py converts a skool file into a binary (raw memory) file. For example:

```
$ skool2bin.py game.skool
```

To list the options supported by *skool2bin.py*, run it with no arguments:

```
usage: skool2bin.py [options] file.skool [file.bin]

Convert a skool file into a binary (raw memory) file. 'file.skool' may be a
regular file, or '-' for standard input. If 'file.bin' is not given, it
defaults to the name of the input file with '.skool' replaced by '.bin'.
'file.bin' may be a regular file, or '-' for standard output.

Options:
  -b, --bfix            Apply @ofix and @bfix directives.
  -d, --data            Process @defb, @defs and @defw directives.
  -E ADDR, --end ADDR   Stop converting at this address.
  -i, --isub            Apply @isub directives.
  -o, --ofix            Apply @ofix directives.
  -r, --rsub            Apply @isub, @ssub and @rsub directives (implies
                        --ofix).
  -R, --rfix            Apply @ofix, @bfix and @rfix directives (implies
                        --rsub).
  -s, --ssub            Apply @isub and @ssub directives.
  -S ADDR, --start ADDR
                        Start converting at this address.
```

(continues on next page)

(continued from previous page)

```
  -v, --verbose         Show info on each converted instruction.
  -V, --version         Show SkoolKit version number and exit.
  -w, --no-warnings     Suppress warnings.
```

The `--verbose` option shows information on each converted instruction, such as whether it was inserted before or after another instruction (by a `@*sub` or `@*fix` directive), and its original address (if it was relocated by the insertion, removal or replacement of other instructions). For example:

```
40000 9C40 > XOR A
40001 9C41 | LD HL,40006    : 40000 9C40 LD HL,40003
40004 9C44 + JR 40006       :            JR 40003
40006 9C46   RET            : 40003 9C43 RET
```

This output shows that:

- The instruction at 40000 (XOR A) was inserted before (>) another instruction
- The instruction at 40001 (LD HL,40006) overwrote (|) the instruction(s) originally at 40000, and had its operand changed from 40003 (because the instruction originally at that address was relocated to 40006)
- The instruction at 40004 (JR 40006) was inserted after (+) another instruction, and also had its operand changed from 40003
- The instruction at 40006 (RET) was originally at 40003 (before other instructions were inserted, removed or replaced)

Version	Changes
8.1	Added the `--data`, `--rsub`, `--rfix`, `--verbose` and `--no-warnings` options
7.0	*@if* directives are processed
6.2	The `--end` and `--start` options accept a hexadecimal integer prefixed by '0x'
6.1	Added the ability to assemble instructions whose operands contain arithmetic expressions
5.2	Added the ability to write the binary file to standard output
5.1	Added the `--bfix`, `--ofix` and `--ssub` options
5.0	New

4.5 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 start and end comments, sub-block types and addresses, instruction-level comments, non-entry blocks, 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 and write it to standard output. FILE
may be a regular file, or '-' for standard input.

Options:
  -b, --preserve-base   Preserve the base of decimal and hexadecimal values in
```

(continues on next page)

(continued from previous page)

```
                        instruction operands and DEFB/DEFM/DEFS/DEFW statements.
  -E ADDR, --end ADDR   Stop converting at this address.
  -h, --hex             Write addresses in upper case hexadecimal format.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -k, --keep-lines      Preserve line breaks in comments.
  -l, --hex-lower       Write addresses in lower case hexadecimal format.
  --show-config         Show configuration parameter values.
  -S ADDR, --start ADDR
                        Start converting at this address.
  -V, --version         Show SkoolKit version number and exit.
  -w X, --write X       Write only these elements, where X is one or more of:
                          a = ASM directives
                          b = block types and addresses
                          t = block titles
                          d = block descriptions
                          r = registers
                          m = mid-block comments and block start/end comments
                          s = sub-block types and addresses
                          c = instruction-level comments
                          n = non-entry blocks
```

4.5.1 Configuration

skool2ctl.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `Hex` - write addresses in decimal (`0`, the default), lower case hexadecimal (`1`), or upper case hexadecimal (`2`)
- `KeepLines` - preserve line breaks in comments (`1`), or don't (`0`, the default)
- `PreserveBase` - preserve the base of decimal and hexadecimal values in instruction operands and DEFB/DEFM/DEFS/DEFW statements (`1`), or don't (`0`, the default)

Configuration parameters must appear in a `[skool2ctl]` section. For example, to make *skool2ctl.py* write upper case hexadecimal addresses by default (without having to use the `-h` option on the command line), add the following section to *skoolkit.ini*:

```
[skool2ctl]
Hex=2
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
7.2	Configuration is read from *skoolkit.ini* if present; added the `--ini`, `--show-config` and `--keep-lines` options
7.0	Added support for the 'n' identifier in the `--write` option
6.2	The `--end` and `--start` options accept a hexadecimal integer prefixed by '0x'
6.0	Added support for the 'a' identifier in the `--write` option
5.1	A terminal `i` directive is appended if the skool file ends before 65536
4.5	Added the `--start` and `--end` options
4.4	Added the `--hex-lower` option
3.7	Added the `--preserve-base` option
3.4	Added the `-V` option and the long options
2.4	Added the ability to preserve some ASM directives
2.2.2	Added the ability to read a skool file from standard input
2.0.6	Added the `-h` option
1.1	New

4.6 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 in the same directory as *game.skool*, they will be used to provide further information to the conversion process, along with any extra files named in the `RefFiles` parameter in the *[Config]* section, and any other ref files named on the command line.

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

```
usage: skool2html.py [options] SKOOLFILE [REFFILE...]

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

Options:
  -1, --asm-one-page    Write all routines and data blocks to a single page.
  -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.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -j NAME, --join-css NAME
                        Concatenate CSS files into a single file with this name.
  -l, --lower           Write the disassembly in lower case.
```

(continues on next page)

(continued from previous page)

```
  -o, --rebuild-images  Overwrite existing image files.
  -O, --rebuild-audio   Overwrite existing audio files.
  -p, --package-dir     Show path to skoolkit package directory and exit.
  -P PAGES, --pages PAGES
                        Write only these pages (when using '--write P').
                        PAGES is a comma-separated list of page IDs.
  -q, --quiet           Be quiet.
  -r PREFIX, --ref-sections PREFIX
                        Show default ref file sections whose names start with
                        PREFIX and exit.
  -R, --ref-file        Show the entire default ref file and exit.
  -s, --search-dirs     Show the locations skool2html.py searches for resources.
  -S DIR, --search DIR  Add this directory to the resource search path. This
                        option may be used multiple times.
  --show-config         Show configuration parameter values.
  -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.
  --var name=value      Define a variable that can be used by @if and the SMPL
                        macros. This option may be used multiple times.
  -V, --version         Show SkoolKit version number and exit.
  -w X, --write X       Write only these files, where X is one or more of:
                          d = Disassembly files   o = Other code
                          i = Disassembly index   P = Other pages
                          m = Memory maps
  -W CLASS, --writer CLASS
                        Specify the HTML writer class to use; shorthand for
                        '--config Config/HtmlWriterClass=CLASS'.
```

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

- The directory that contains the skool file named on the command line
- The current working directory
- *./resources*
- *~/.skoolkit*
- *$PACKAGE_DIR/resources*
- Any other directories specified by the `-S`/`--search` option

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.skool
```

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*
- *dark.css*
- *wide.css*

4.6.1 Configuration

skool2html.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `AsmLabels` - use ASM labels (`1`), or don't (`0`, the default)
- `AsmOnePage` - write all routines and data blocks to a single page (`1`), or to multiple pages (`0`, the default)
- `Base` - convert addresses and instruction operands to hexadecimal (`16`) or decimal (`10`), or leave them as they are (`0`, the default)
- `Case` - write the disassembly in lower case (`1`) or upper case (`2`), or leave it as it is (`0`, the default)
- `CreateLabels` - create default labels for unlabelled instructions (`1`), or don't (`0`, the default)
- `EntryLabel` - the format of the default label for the first instruction in a routine or data block (default: `L{address}`)
- `EntryPointLabel` - the format of the default label for an instruction other than the first in a routine or data block (default: `{main}_{index}`)
- `JoinCss` - if specified, concatenate CSS files into a single file with this name
- `OutputDir` - write files in this directory (default: `.`)
- `Quiet` - be quiet (`1`) or verbose (`0`, the default)
- `RebuildAudio` - overwrite existing audio files (`1`), or leave them alone (`0`, the default)
- `RebuildImages` - overwrite existing image files (`1`), or leave them alone (`0`, the default)
- `Search` - directory to add to the resource search path; to specify two or more directories, separate them with commas
- `Theme` - CSS theme to use; to specify two or more themes, separate them with commas
- `Time` - show timings (`1`), or don't (`0`, the default)

`EntryLabel` and `EntryPointLabel` are standard Python format strings. `EntryLabel` recognises the following replacement fields:

- `address` - the address of the routine or data block as it appears in the skool file
- `location` - the address of the routine or data block as an integer

`EntryPointLabel` recognises the following replacement fields:

- `address` - the address of the instruction as it appears in the skool file
- `index` - 0 for the first unlabelled instruction in the routine or data block, 1 for the second, etc.

- `location` - the address of the instruction as an integer
- `main` - the label of the first instruction in the routine or data block

Configuration parameters must appear in a `[skool2html]` section. For example, to make *skool2html.py* use ASM labels and write the disassembly in hexadecimal by default (without having to use the `-H` and `-a` options on the command line), add the following section to *skoolkit.ini*:

```
[skool2html]
AsmLabels=1
Base=16
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
8.7	Added the `--rebuild-audio` option and the `RebuildAudio` configuration parameter
8.5	Added the `EntryLabel` and `EntryPointLabel` configuration parameters
7.0	Writes a single disassembly from the skool file given by the first positional argument
6.4	Added the `--var` option
6.2	Added the `--show-config` option
6.1	Configuration is read from *skoolkit.ini* if present; added the `--ini` option
5.4	Added the `--asm-one-page` option
5.0	The `--theme` option also looks for a CSS file whose base name matches the theme name
4.1	Added the `--search` and `--writer` options
4.0	Added the `--ref-sections` and `--ref-file` options
3.6	Added the `--join-css` and `--search-dirs` options
3.5	Added support for multiple CSS themes
3.4	Added the `-a` and `-C` options and the long options
3.3.2	Added *$PACKAGE_DIR/resources* to the search path; added the `-p` and `-T` options
3.2	Added *~/.skoolkit* to the search path
3.1	Added the `-c` option
3.0.2	No longer shows timings by default; added the `-t` option
2.3.1	Added support for reading multiple ref files per disassembly
2.2.2	Added the ability to read a skool file from standard input
2.2	No longer writes the Skool Daze and Back to Skool disassemblies by default; added the `-d` option
2.1.1	Added the `-l`, `-u`, `-D` and `-H` options
2.1	Added the `-o` and `-P` options
1.4	Added the `-V` option

4.7 sna2ctl.py

sna2ctl.py generates a control file for a binary (raw memory) file or a SNA, SZX or Z80 snapshot. For example:

```
$ sna2ctl.py game.z80 > game.ctl
```

Now *game.ctl* can be used by *sna2skool.py* to convert *game.z80* into a skool file split into blocks of code and data.

sna2ctl.py supports several options; run it with no arguments to see a list:

```
usage: sna2ctl.py [options] FILE

Generate a control file for a binary (raw memory) file or a SNA, SZX or Z80
```

(continues on next page)

(continued from previous page)

```
snapshot. FILE may be a regular file, or '-' for standard input.

Options:
  -e ADDR, --end ADDR   Stop at this address (default=65536).
  -h, --hex             Write upper case hexadecimal addresses.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -l, --hex-lower       Write lower case hexadecimal addresses.
  -m FILE, --map FILE   Use FILE as a code execution map.
  -o ADDR, --org ADDR   Specify the origin address of a binary file (default:
                        65536 - length).
  -p PAGE, --page PAGE  Specify the page (0-7) of a 128K snapshot to map to
                        49152-65535.
  --show-config         Show configuration parameter values.
  -s ADDR, --start ADDR
                        Start at this address.
  -V, --version         Show SkoolKit version number and exit.
```

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

The `-m` option may be used 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 SpecEmu and Z80 emulators

If the file specified by the `-m` option is 8192 bytes long, it is assumed to be a Z80 map file; if it is 65536 bytes long, it is assumed to be a SpecEmu map file; otherwise it is assumed to be in one of the other supported formats.

4.7.1 Configuration

sna2ctl.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `Dictionary` - the name of a file containing a list of allowed words, one per line; if specified, a string of characters will be marked as text only if it contains at least one of the words in this file
- `Hex` - write addresses in decimal (`0`, the default), lower case hexadecimal (`1`), or upper case hexadecimal (`2`)
- `TextChars` - characters eligible for being marked as text (default: letters, digits, space, and the following non-alphanumeric characters: `!"$%&\'()*+,-./:;<=>?[]`)
- `TextMinLengthCode` - the minimum length of a string of characters eligible for being marked as text in a block identified as code (default: `12`)
- `TextMinLengthData` - the minimum length of a string of characters eligible for being marked as text in a block identified as data (default: `3`)

Configuration parameters must appear in a `[sna2ctl]` section. For example, to make *sna2ctl.py* write upper case hexadecimal addresses by default (without having to use the `-h` option on the command line), add the following section to *skoolkit.ini*:

```
[sna2ctl]
Hex=2
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
7.2	Added the `Dictionary` configuration parameter
7.1	Configuration is read from *skoolkit.ini* if present; added the `--ini` and `--show-config` options
7.0	New

4.8 sna2img.py

sna2img.py converts the screenshot or other graphic data in a binary (raw memory) file, SCR file, skool file, or SNA/SZX/Z80 snapshot into a PNG file. For example:

```
$ sna2img.py game.scr
```

will create a file named *game.png*.

To list the options supported by *sna2img.py*, run it with no arguments:

```
usage: sna2img.py [options] INPUT [OUTPUT]

Convert a Spectrum screenshot or other graphic data into a PNG file. INPUT may
be a binary (raw memory) file, a SCR file, a skool file, or a SNA, SZX or Z80
snapshot.

Options:
  -b, --bfix            Parse a skool file in @bfix mode.
  -B, --binary          Read the input as a binary (raw memory) file.
  -e MACRO, --expand MACRO
                        Expand a #FONT, #SCR, #UDG or #UDGARRAY macro. The '#'
                        prefix may be omitted.
  -f N, --flip N        Flip the image horizontally (N=1), vertically (N=2),
                        or both (N=3).
  -i, --invert          Invert video for cells that are flashing.
  -m src,size,dest, --move src,size,dest
                        Move a block of bytes of the given size from src to
                        dest. This option may be used multiple times.
  -n, --no-animation    Do not animate flashing cells.
  -o X,Y, --origin X,Y  Top-left crop at (X,Y).
  -O ORG, --org ORG     Set the origin address of a binary file (default:
                        65536 minus the length of the file).
  -p a[-b[-c]],[^+]v, --poke a[-b[-c]],[^+]v
                        POKE N,v for N in {a, a+c, a+2c..., b}. Prefix 'v'
                        with '^' to perform an XOR operation, or '+' to
                        perform an ADD operation. This option may be used
                        multiple times.
  -r N, --rotate N      Rotate the image 90*N degrees clockwise.
  -s SCALE, --scale SCALE
                        Set the scale of the image (default=1).
  -S WxH, --size WxH    Crop to this width and height (in tiles).
  -V, --version         Show SkoolKit version number and exit.
```

Version	Changes
6.2	Added the `--binary` and `--org` options and the ability to read binary (raw memory) files; the `--move` and `--poke` options accept hexadecimal integers prefixed by '0x'
6.1	Added the ability to read skool files; added the `--bfix` and `--move` options
6.0	Added the `--expand` option
5.4	New

4.9 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. FILE may be a regular file, or '-' for standard input.

Options:
  -c PATH, --ctl PATH   Specify a control file to use, or a directory from
                        which to read control files. PATH may be '-' for
                        standard input, or '0' to use no control file. This
                        option may be used multiple times.
  -d SIZE, --defb SIZE  Disassemble as DEFB statements of this size.
  -e ADDR, --end ADDR   Stop disassembling at this address (default=65536).
  -H, --hex             Write hexadecimal addresses and operands in the
                        disassembly.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -l, --lower           Write the disassembly in lower case.
  -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.
  --show-config         Show configuration parameter values.
  -s ADDR, --start ADDR
                        Start disassembling at this address.
  -V, --version         Show SkoolKit version number and exit.
  -w W, --line-width W  Set the maximum line width of the skool file (default:
                        79).
```

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

By default, any files whose names start with the input filename (minus the '.bin', '.sna', '.szx' or '.z80' suffix, if any) and end with '.ctl' will be used as *control files*.

4.9.1 Configuration

sna2skool.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `Base` - write addresses and instruction operands in hexadecimal (`16`) or decimal (`10`, the default)
- `Case` - write the disassembly in lower case (`1`) or upper case (`2`, the default)
- `CommentWidthMin` - minimum width of the instruction comment field in the skool file (default: `10`)
- `DefbSize` - maximum number of bytes in a DEFB statement (default: `8`)
- `DefmSize` - maximum number of characters in a DEFM statement (default: `65`)
- `DefwSize` - maximum number of words in a DEFW statement (default: `1`)
- `EntryPointRef` - template used to format the comment for an entry point with exactly one referrer (default: `This entry point is used by the routine at {ref}.`)
- `EntryPointRefs` - template used to format the comment for an entry point with two or more referrers (default: `This entry point is used by the routines at {refs} and {ref}.`)
- `InstructionWidth` - minimum width of the instruction field in the skool file (default: `13`)
- `LineWidth` - maximum line width of the skool file (default: `79`)
- `ListRefs` - when to add a comment that lists routine or entry point referrers: never (`0`), if no other comment is defined at the entry point (`1`, the default), or always (`2`)
- `Ref` - template used to format the comment for a routine with exactly one referrer (default: `Used by the routine at {ref}.`)
- `RefFormat` - template used to format referrers in the `{ref}` and `{refs}` fields of the `Ref` and `Refs` templates (default: `#R{address}`); the replacement field `address` is the address of the referrer formatted as a decimal or hexadecimal number in accordance with the `Base` and `Case` configuration parameters
- `Refs` - template used to format the comment for a routine with two or more referrers (default: `Used by the routines at {refs} and {ref}.`)
- `Semicolons` - block types (`b`, `c`, `g`, `i`, `s`, `t`, `u`, `w`) in which comment semicolons are written for instructions that have no comment (default: `c`)
- `Text` - show ASCII text in the comment fields (`1`), or don't (`0`, the default)
- `Timings` - show instruction timings in the comment fields (`1`), or don't (`0`, the default)
- `Title-b` - template used to format the title for an untitled 'b' block (default: `Data block at {address}`)
- `Title-c` - template used to format the title for an untitled 'c' block (default: `Routine at {address}`)
- `Title-g` - template used to format the title for an untitled 'g' block (default: `Game status buffer entry at {address}`)
- `Title-i` - template used to format the title for an untitled 'i' block (default: `Ignored`)
- `Title-s` - template used to format the title for an untitled 's' block (default: `Unused`)
- `Title-t` - template used to format the title for an untitled 't' block (default: `Message at {address}`)
- `Title-u` - template used to format the title for an untitled 'u' block (default: `Unused`)
- `Title-w` - template used to format the title for an untitled 'w' block (default: `Data block at {address}`)
- `Wrap` - disassemble an instruction that wraps around the 64K boundary (`1`), or don't (`0`, the default)

Configuration parameters must appear in a `[sna2skool]` section. For example, to make *sna2skool.py* generate hexadecimal skool files with a line width of 120 characters by default (without having to use the `-H` and `-w` options on the command line), add the following section to *skoolkit.ini*:

```
[sna2skool]
Base=16
LineWidth=120
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
8.7	Added the `--defb` option and the `Timings` configuration parameter
8.5	Added the `Wrap` configuration parameter and the ability to disassemble an instruction that wraps around the 64K boundary; added the `RefFormat` configuration parameter
8.4	Changed the default value of the `DefmSize` configuration parameter from 66 to 65
8.3	Added support for reading control files from a directory (`--ctl DIR`)
8.1	Added support for ignoring default control files (`--ctl 0`)
8.0	Added the `DefwSize` configuration parameter
7.1	Added support for reading multiple default control files, and for using the `--ctl` option multiple times; added the `CommentWidthMin`, `InstructionWidth` and `Semicolons` configuration parameters
7.0	The short option for `--lower` is `-l`; the long option for `-H` is `--hex`
6.2	Added the `--show-config` option; the `--end`, `--org` and `--start` options accept a hexadecimal integer prefixed by '0x'
6.1	Configuration is read from *skoolkit.ini* if present; added the `--ini` option
4.4	Added the `--end` option
4.3	Added the `--line-width` option
3.4	Added the `-V` option and the long options, and the ability to add a comment listing referrers at every routine entry point
3.3	Added the ability to read 128K SNA snapshots
3.2	Added the `-p` option, and the ability to read SZX snapshots and 128K Z80 snapshots
2.1.2	Added the ability to write the disassembly in lower case
2.1	Added the `-H` option
2.0.1	Added the `-o` option, and the ability to read binary files, to set the maximum number of characters in a DEFM statement, and to suppress comments that list routine entry point referrers
2.0	Added the ability to set the maximum number of bytes in a DEFB statement
1.0.5	Added the ability to show ASCII text in comment fields
1.0.4	Added the `-s` option

4.10 snapinfo.py

snapinfo.py shows information on the registers or RAM in a binary (raw memory) file or a SNA, SZX or Z80 snapshot. For example:

```
$ snapinfo.py game.z80
```

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

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

```

(continues on next page)

(continued from previous page)

```
Analyse a binary (raw memory) file or a SNA, SZX or Z80 snapshot.

Options:
  -b, --basic           List the BASIC program.
  -c PATH, --ctl PATH   When generating a call graph, specify a control file
                        to use, or a directory from which to read control
                        files. PATH may be '-' for standard input. This option
                        may be used multiple times.
  -f A[,B...[-M[-N]]], --find A[,B...[-M[-N]]]
                        Search for the byte sequence A,B... with distance
                        ranging from M to N (default=1) between bytes.
  -g, --call-graph      Generate a call graph in DOT format.
  -I p=v, --ini p=v     Set the value of the configuration parameter 'p' to
                        'v'. This option may be used multiple times.
  -o ADDR, --org ADDR   Specify the origin address of a binary (raw memory)
                        file (default: 65536 - length).
  -p A[-B[-C]], --peek A[-B[-C]]
                        Show the contents of addresses A TO B STEP C. This
                        option may be used multiple times.
  -P PAGE, --page PAGE  Specify the page (0-7) of a 128K snapshot to map to
                        49152-65535.
  --show-config         Show configuration parameter values.
  -t TEXT, --find-text TEXT
                        Search for a text string.
  -T X,Y[-M[-N]], --find-tile X,Y[-M[-N]]
                        Search for the graphic data of the tile at (X,Y) with
                        distance ranging from M to N (default=1) between
                        bytes.
  -v, --variables       List variables.
  -V, --version         Show SkoolKit version number and exit.
  -w A[-B[-C]], --word A[-B[-C]]
                        Show the words at addresses A TO B STEP C. This option
                        may be used multiple times.
```

With no options, *snapinfo.py* displays register values, the interrupt mode, and the border colour. By using one of the options shown above, it can list the BASIC program and variables (if present), show the contents of a range of addresses, search the RAM for a sequence of byte values or a text string, or generate a call graph.

4.10.1 Call graphs

snapinfo.py can generate a call graph in DOT format from a snapshot and a corresponding control file. For example, if *game.ctl* is present alongside *game.z80*, then:

```
$ snapinfo.py -g game.z80 > game.dot
```

will produce a call graph in *game.dot*, with a node for each routine declared in *game.ctl*, and an edge between two nodes whenever the routine represented by the first node calls, jumps to, or continues into the routine represented by the second node.

To create a PNG image file named *game.png* from *game.dot*, the *dot* utility (included in Graphviz) may be used:

```
$ dot -Tpng game.dot > game.png
```

A call graph may contain one or more 'orphans', an orphan being a node that is not at the head of any arrow, and thus represents a routine that is (as far as *snapinfo.py* can tell) not used by any other routines. To declare the callers of such a routine (in case it is not a true orphan), the *@refs* directive may be used.

To help identify orphan nodes and missing edges, each of the first three lines of the DOT file produced by *snapinfo.py* contains a list of IDs of the following types of node:

- unconnected nodes
- orphan nodes connected to other nodes
- non-orphan nodes whose first instruction is not used

The appearance of nodes and edges in a call graph image can be configured via the `EdgeAttributes`, `GraphAttributes`, `NodeAttributes` and `NodeLabel` configuration parameters (see below).

4.10.2 Configuration

snapinfo.py will read configuration from a file named *skoolkit.ini* in the current working directory or in *~/.skoolkit*, if present. The recognised configuration parameters are:

- `EdgeAttributes` - the default attributes for edges in a call graph (default: none)
- `GraphAttributes` - the default attributes for a call graph (default: none)
- `NodeAttributes` - the default attributes for nodes in a call graph (default: `shape=record`)
- `NodeId` - the format of the node IDs in a call graph (default: `{address}`)
- `NodeLabel` - the format of the node labels in a call graph (default: `"{address} {address:04X}\n{label}"`)
- `Peek` - the format of each line of the output produced by the `--peek` option (default: `{address:>5} {address:04X}:  {value:>3} {value:02X} {value:08b} {char}`)
- `Word` - the format of each line of the output produced by the `--word` option (default: `{address:>5} {address:04X}:  {value:>5} {value:04X}`)

`NodeId` and `NodeLabel` are standard Python format strings that recognise the replacement fields `address` and `label` (the address and label of the first instruction in the routine represented by the node).

Configuration parameters must appear in a `[snapinfo]` section. For example, to make *snapinfo.py* use open arrowheads and a cyan background colour in call graphs by default, add the following section to *skoolkit.ini*:

```
[snapinfo]
EdgeAttributes=arrowhead=open
GraphAttributes=bgcolor=cyan
```

Configuration parameters may also be set on the command line by using the `--ini` option. Parameter values set this way will override any found in *skoolkit.ini*.

Version	Changes
8.4	Added the `Peek` and `Word` configuration parameters
8.3	Added support for reading control files from a directory (`--ctl DIR`)
8.2	Configuration is read from *skoolkit.ini* if present; added the ability to read binary files; added the `--call-graph`, `--ctl`, `--ini`, `--org`, `--page` and `--show-config` options
6.2	The `--find`, `--find-tile`, `--peek` and `--word` options accept hexadecimal integers prefixed by '0x'
6.0	Added support to the `--find` option for distance ranges; added the `--find-tile` and `--word` options; the `--peek` option shows UDGs and BASIC tokens
5.4	Added the `--variables` option; UDGs in a BASIC program are shown as special symbols (e.g. `{UDG-A}`)
5.3	New

4.11 snapmod.py

snapmod.py modifies the registers and RAM in a 48K Z80 snapshot. For example:

```
$ snapmod.py --poke 32768,0 game.z80 poked.z80
```

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

```
usage: snapmod.py [options] in.z80 [out.z80]

Modify a 48K Z80 snapshot.

Options:
  -f, --force           Overwrite an existing snapshot.
  -m src,size,dest, --move src,size,dest
                        Move a block of bytes of the given size from src to
                        dest. This option may be used multiple times.
  -p a[-b[-c]],[^+]v, --poke a[-b[-c]],[^+]v
                        POKE N,v for N in {a, a+c, a+2c..., b}. Prefix 'v'
                        with '^' to perform an XOR operation, or '+' to
                        perform an ADD operation. This option may be used
                        multiple times.
  -r name=value, --reg name=value
                        Set the value of a register. Do '--reg help' for more
                        information. This option may be used multiple times.
  -s name=value, --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.
```

Version	Changes
6.2	The `--move`, `--poke` and `--reg` options accept hexadecimal integers prefixed by '0x'
5.3	New

4.12 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:
```

(continues on next page)

(continued from previous page)

```
  -d DIR, --output-dir DIR
                        Write the snapshot file in this directory.
  -f, --force           Overwrite an existing snapshot.
  -p STACK, --stack STACK
                        Set the stack pointer.
  --ram OPERATION       Perform a load operation or otherwise modify 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.
  -s START, --start START
                        Set the start address to JP to.
  --sim-load            Simulate a 48K ZX Spectrum running LOAD "".
  --state name=value    Set a hardware state attribute. Do '--state help' for
                        more information. This option may be used multiple
                        times.
  -u AGENT, --user-agent AGENT
                        Set the User-Agent header.
  -V, --version         Show SkoolKit version number and exit.
```

Note that support for TZX files is limited to block types 0x10 (standard speed data), 0x11 (turbo speed data) and 0x14 (pure data).

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. (To see information on the blocks in a TAP or TZX file, use the *tapinfo.py* command.)

An alternative to the `--ram load` approach is the `--sim-load` option. It simulates a freshly booted 48K ZX Spectrum running LOAD "" (or LOAD ""CODE, if the first block on the tape is a 'Bytes' header). Whenever the Spectrum ROM's load routine at $0556 is called, a shortcut is taken by fast loading the next block on the tape. All other code (including any custom loader) is fully simulated. Simulation continues until the program counter hits the start address given by the `--start` option, or 10 minutes of simulated Z80 CPU time has elapsed, or the end of the tape is reached and one of the following conditions is satisfied:

- a custom loader was detected
- the program counter hits an address outside the ROM
- more than one second of simulated Z80 CPU time has elapsed since the end of the tape was reached

The simulation can also be aborted by pressing Ctrl-C. When a simulated LOAD has completed or been aborted, the values of the registers (including the program counter) in the simulator are used to populate the Z80 snapshot.

In addition to loading specific blocks, the `--ram` option can also be used to move blocks of bytes from one location to another, POKE values into individual addresses or address ranges, modify memory with XOR and ADD operations, initialise the system variables, or call a Python function to modify the memory snapshot in an arbitrary way before it is saved. For more information on these operations, run:

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

For complex snapshots that require many 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
--ram call=:ram.modify     # Call modify(snapshot) in ./ram.py
--ram sysvars              # Initialise the system variables
--state iff=0              # Disable interrupts
--stack 32768              # Stack at 32768
--start 34816              # Start at 34816
```

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
8.8	The `--sim-load` option performs any `call/move/poke/sysvars` operations specified by `--ram`
8.7	Added the `--sim-load` option; when a headerless block is ignored because no `--ram load` options have been specified, a warning is printed
8.6	Added support to the `--ram` option for the `call` operation
8.4	Added support to the `--ram` option for the `sysvars` operation
6.3	Added the `--user-agent` option
6.2	The `--ram`, `--reg`, `--stack` and `--start` options accept hexadecimal integers prefixed by '0x'
5.3	Added the `--stack` and `--start` options
4.5	Added support for TZX block type 0x14 (pure data), for loading the first and last bytes of a tape block, and for modifying memory with XOR and ADD operations
3.5	New

4.13 tapinfo.py

tapinfo.py shows information on the blocks in a TAP or TZX file. For example:

```
$ tapinfo.py game.tzx
```

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

```
usage: tapinfo.py FILE

Show the blocks in a TAP or TZX file.

Options:
  -b IDs, --tzx-blocks IDs
                        Show TZX blocks with these IDs only. 'IDs' is a comma-
                        separated list of hexadecimal block IDs, e.g.
                        10,11,2a.
  -B N[,A], --basic N[,A]
                        List the BASIC program in block N loaded at address A
                        (default 23755).
```

(continues on next page)

(continued from previous page)

```
  -d, --data            Show the entire contents of header and data blocks.
  -V, --version         Show SkoolKit version number and exit.
```

Version	Changes
8.3	Added the `--data` option
8.1	Shows contents of TZX block types 0x33 (hardware type) and 0x35 (custom info)
7.1	Shows pulse lengths in TZX block type 0x13 and full info for TZX block type 0x14
6.2	The `--basic` option accepts a hexadecimal address prefixed by '0x'
6.0	Added the `--basic` option
5.0	New

4.14 trace.py

trace.py simulates the execution of machine code in a 48K memory snapshot. For example:

```
$ trace.py --start 32768 --stop 49152 game.z80
```

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

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

Trace Z80 machine code execution. FILE may be a binary (raw memory) file, a
SNA, SZX or Z80 snapshot, or '.' for no snapshot.

Options:
  --audio               Show audio delays.
  --depth DEPTH         Simplify audio delays to this depth (default: 2).
  -D, --decimal         Show decimal values in verbose mode.
  --dump FILE           Dump a Z80 snapshot to this file after execution.
  --max-operations MAX  Maximum number of instructions to execute.
  --max-tstates MAX     Maximum number of T-states to run for.
  -o ADDR, --org ADDR   Specify the origin address of a binary (raw memory)
                        file (default: 65536 - length).
  -p a[-b[-c]],[^+]v, --poke a[-b[-c]],[^+]v
                        POKE N,v for N in {a, a+c, a+2c..., b}. Prefix 'v'
                        with '^' to perform an XOR operation, or '+' to
                        perform an ADD operation. This option may be used
                        multiple times.
  -r name=value, --reg name=value
                        Set the value of a register. Do '--reg help' for more
                        information. This option may be used multiple times.
  --rom FILE            Patch in a ROM at address 0 from this file. By default
                        the 48K ZX Spectrum ROM is used.
  -s ADDR, --start ADDR
                        Start execution at this address.
  -S ADDR, --stop ADDR  Stop execution at this address.
  --stats               Show stats after execution.
  -v, --verbose         Show executed instructions. Repeat this option to show
                        register values too.
  -V, --version         Show SkoolKit version number and exit.
```

By default, *trace.py* silently simulates code execution beginning with the instruction at the address specified by the `--start` option (or the program counter in the snapshot) and ending when the instruction at the address specified by

`--stop` (if any) is reached. Use the `--verbose` option to show each instruction executed. Repeat the `--verbose` option (`-vv`) to show register values too.

When the `--audio` option is given, *trace.py* tracks changes in the state of the ZX Spectrum speaker, and then prints a list of the delays (in T-states) between those changes. This list can be supplied to the *#AUDIO* macro to produce a WAV file for the sound effect that would be produced by the same code running on a real ZX Spectrum.

Version	Changes
8.8	New

CHAPTER

FIVE

SUPPORTED ASSEMBLERS

If you use SkoolKit to generate an ASM version of your disassembly, and you want to assemble it, 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)

Note: SjASMPlus does not recognise instructions that operate on the high or low half of the IX and IY registers in the default form used by SkoolKit (e.g. `LD A,IXl`). A workaround is to write the ASM file in lower case using the `--lower` option of *skool2asm.py*.

Note: z80asm does not recognise binary constants in the form supported by SkoolKit (e.g. `%10101010`). If your skool file contains any such constants, the `%` characters must be replaced by `@` (e.g. `@10101010`) after conversion to ASM format.

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

5.1 Using pasmo

First, create an ASM version of the disassembly:

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

Then use *pasmo* to create a binary file named *game.bin* thus:

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

5.2 Using SjASMPlus

First, create an ASM version of the disassembly:

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

Then create a file named *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 named *game.bin* will be created.

5.3 Using z80asm (z88dk)

First, create an ASM version of the disassembly:

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

Then use *z80asm* to create a binary file named *game.bin* thus:

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

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

5.4 Creating a TAP file

Having created *game.bin* by using your chosen assembler, you can now create a TAP file by using *bin2tap.py*:

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

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

5.5 Creating a Z80 snapshot

Having created *game.bin* by using your chosen assembler, you can now create a Z80 snapshot by using *bin2sna.py*:

```
$ bin2sna.py game.bin
```

The resultant snapshot, *game.z80*, can then be loaded into an emulator.

CHAPTER

SIX

MIGRATING FROM SKOOLKIT 7

SkoolKit 8 includes some changes that make it incompatible with SkoolKit 7. If you have developed a disassembly using SkoolKit 7 and find that the SkoolKit commands no longer work with your source files, or produce broken output, look through the following sections for tips on how to migrate your disassembly to SkoolKit 8.

6.1 GIF images

Creating GIF images is not supported in SkoolKit 8. The *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros now always create PNG images. Accordingly, the following parameters from the *[ImageWriter]* section that were available in SkoolKit 7 are no longer supported:

- `DefaultAnimationFormat`
- `DefaultFormat`
- `GIFEnableAnimation`
- `GIFTransparency`

6.2 Skool file templates

Skool file templates are not supported in SkoolKit 8. The *skool2sft.py* command has been removed, along with the `--sft` option of *sna2skool.py*.

Where you might have used skool file templates with SkoolKit 7, you should now use *control files* instead. However, note that control files cannot preserve ASM block directives that occur inside a regular entry, and so any such directives should be replaced before using *skool2ctl.py*. See *Limitations* for more details.

6.3 [Game]

The `TitlePrefix` and `TitleSuffix` parameters are no longer supported. Use the `GameIndex` parameter in the *[PageHeaders]* section instead.

The `AsmSinglePageTemplate` parameter is no longer supported. Use the `AsmSinglePage` parameter instead.

6.4 [Titles]

In SkoolKit 7, the entry address in a disassembly page title was included in the `Asm` template. In SkoolKit 8, the `Asm` template no longer exists, and the entry address appears as a replacement field (`{entry[address]}`) in the `Asm-b`, `Asm-c`, `Asm-g`, `Asm-s`, `Asm-t`, `Asm-u` and `Asm-w` parameters in the *[Titles]* section.

6.5 Control directives

The `B` and `T` control directives no longer recognise the `B` (byte) and `T` (text) indicators. Use the `n` and `c` base indicators instead. For example:

```
B 30000,5,2:T3
T 30005,5,B3:2
```

should be replaced by:

```
B 30000,5,2:c3
T 30005,5,n3:2
```

6.6 sna2skool.py

The `DefbMod` configuration parameter is no longer supported. It could be used to group DEFB blocks by addresses that are divisible by a certain number, but the same effect can be achieved with appropriate control directives.

The `DefbZfill` configuration parameter is also no longer supported.

6.7 HTML templates

The *HTML templates* have been overhauled in SkoolKit 8. As a result, the following templates that were available in SkoolKit 7 no longer exist:

- `Asm`
- `AsmAllInOne`
- `GameIndex`
- `MemoryMap`
- `Page`
- `Reference`
- `anchor`
- `asm_comment`
- `asm_entry`
- `asm_instruction`
- `asm_register`
- `contents_list_item`
- `index_section`

- `index_section_item`
- `javascript`
- `list_entry`
- `list_item`
- `list_items`
- `map_entry`
- `paragraph`
- `reference_entry`
- `stylesheet`
- `table_cell`
- `table_header_cell`
- `table_row`

In addition, the following templates have been rewritten to use the *foreach* and *if* directives, which are new in SkoolKit 8:

- *list*
- *table*

Finally, the signature of the `format_template()` method on HtmlWriter has changed in SkoolKit 8.0: the *default* parameter has been removed.

6.8 CSS selectors

The *class* attributes of some HTML elements have changed in SkoolKit 8.

The following table lists the selectors that appeared in the CSS files in SkoolKit 7, and their replacements (if any) in SkoolKit 8.

SkoolKit 7	SkoolKit 8
div.map-entry-desc-0	
div.map-entry-desc-1	div.map-entry-desc
span.next-0	
span.prev-0	
table.input-0	
table.input-1	table.input
table.output-0	
table.output-1	table.output
td.asm-label-0	
td.asm-label-1	td.asm-label
td.bytes-0	
td.bytes-1	td.bytes
td.comment-01	
td.comment-10	td.comment-0
td.comment-11	td.comment-1
td.map-byte-0	
td.map-byte-1	td.map-byte
td.map-length-0	
td.map-length-1	td.map-length
td.map-page-0	
td.map-page-1	td.map-page
th.map-byte-0	
th.map-length-0	
th.map-page-0	

The following table lists selectors for the classes that were unstyled (i.e. did not appear in any CSS files) in SkoolKit 7, and their replacements (if any) in SkoolKit 8.

SkoolKit 7	SkoolKit 8
span.next-1	
span.prev-1	
th.map-byte-1	th.map-byte
th.map-length-1	th.map-length
th.map-page-1	th.map-page

6.9 skoolkit7to8.py

The skoolkit7to8.py script may be used to convert a control file or ref file that is compatible with SkoolKit 7 into a file that will work with SkoolKit 8. For example, to convert *game.ref*:

```
$ skoolkit7to8.py game.ref > game8.ref
```

CHAPTER

SEVEN

CHANGELOG

7.1 8.8 (2022-11-19)

- Added the *trace.py* command (for tracing the execution of machine code in a 48K memory snapshot)
- The `--sim-load` option of *tap2sna.py* now performs any `call`, `move`, `poke` and `sysvars` operations specified by the `--ram` option
- Improved the performance of the `--sim-load` option of *tap2sna.py*
- Improved the performance of the *#SIM* macro
- Improved the performance of the *#AUDIO* and *#TSTATES* macros when they execute instructions in a simulator
- Removed the `MaxAmplitude` parameter from the *[AudioWriter]* section

7.2 8.7 (2022-10-08)

- Dropped support for Python 3.6
- Added the *#SIM* macro (for simulating the execution of machine code in the internal memory snapshot constructed from the contents of the skool file)
- Added the *#AUDIO* macro (for creating HTML5 `<audio>` elements, and optionally creating audio files in WAV format)
- Added the *#TSTATES* macro (which expands to the time taken, in T-states, to execute one or more instructions)
- Added the `--sim-load` option to *tap2sna.py* (to simulate a 48K ZX Spectrum running LOAD "")
- Added the *@rom* directive (for inserting a copy of the 48K ZX Spectrum ROM into the internal memory snapshot constructed from the contents of the skool file)
- Added the `AudioPath` parameter to the *[Paths]* section (for specifying where the *#AUDIO* macro should look for or create audio files by default)
- Added the *audio* template (for formatting the `<audio>` element produced by the *#AUDIO* macro)
- Added the *[AudioWriter]* section (for configuring audio files created by the *#AUDIO* macro)
- Added the `--rebuild-audio` option to and the `RebuildAudio` configuration parameter for *skool2html.py* (to overwrite existing audio files)
- Added the `AudioFormats` parameter to the *[Game]* section (for specifying the alternative audio file formats that the *#AUDIO* macro should look for before creating a WAV file)
- Added the `--defb` option to *sna2skool.py* (to disassemble as DEFB statements instead of as code)

- Added the `Timings` configuration parameter for *sna2skool.py* (for showing instruction timings in the comment fields)
- Added the `flags` parameter to the *#FOR* macro (for affixing commas to and replacing variable names in each separator)
- Added support to the *M directive* for applying its comment to each instruction in its range
- When *tap2sna.py* ignores a headerless block because no `--ram load` options have been specified, it now prints a warning
- Amended the *register* ASM template so that it can handle empty register names
- Fixed the bug where the `stop` value of the *#FOR* macro is used even when it does not differ from `start` by a multiple of `step`
- Fixed the bug where an *M directive* with an explicit length overrides the sublengths of an earlier sub-block directive at the same address

7.3 8.6 (2021-11-06)

- Added the *#STR* macro (for retrieving the text string at a given address in the memory snapshot)
- Added the *#WHILE* macro (for repeatedly expanding macros until a conditional expression becomes false)
- Added the *#UDGS* macro (as an alternative to the *#UDGARRAY* macro for creating an image of a rectangular array of UDGs)
- Added support to the *#DEF* macro for using replacement fields to represent the defined macro's argument values, and for stripping leading and trailing whitespace from the defined macro's output
- Added support to the *#LET* macro for defining dictionary variables
- Added support to the `--ram` option of *tap2sna.py* for the `call` operation (for calling a Python function to perform arbitrary manipulation of the memory snapshot)
- Added the `flags` parameter to the *#CHR* macro (to produce a character in the UTF-8 encoding in HTML mode, and to map character codes 94, 96 and 127 to '↑', '£' and '©')
- Added the `Expand` parameter to the *[Config]* section (for specifying skool macros to be expanded during HTML writer initialisation)
- Added support to the *#INCLUDE* macro for combining the contents of multiple ref file sections
- Added the `tindex` and `alpha` parameters to the *#COPY* macro (for specifying the transparent colour and its alpha value in the new frame)
- Fixed the bug where macros inside a *#LIST* or *#TABLE* macro are expanded twice in HTML mode (which makes *#RAW* ineffective)

7.4 8.5 (2021-07-03)

- Dropped support for Python 3.5
- Added the *#OVER* macro (for superimposing one frame on another)
- Added the *#COPY* macro (for copying all or part of an existing frame into a new frame)
- Added the *#DEF* macro (as a more powerful alternative to the *#DEFINE* macro, which is now deprecated)
- Added the `Wrap` configuration parameter for *sna2skool.py* (for controlling whether to disassemble an instruction that wraps around the 64K boundary)
- Added the `RefFormat` configuration parameter for *sna2skool.py* (for specifying the format of referrers in a comment that lists them for a routine or entry point)
- Added the `EntryLabel` and `EntryPointLabel` configuration parameters for *skool2asm.py* and *skool2html.py* (for specifying the format of the default labels for routines and data blocks and their entry points)
- Added the `Address` configuration parameter for *skool2asm.py* (for specifying the format of the default link text for the *#R* macro)
- The `SnapshotReferenceOperations` parameter in the *[skoolkit]* section of *skoolkit.ini* is now interpreted as a list of regular expression patterns (which enables any type of instruction to be designated by the *snapshot reference calculator* as one whose address operand identifies an entry point in a routine or data block)
- Added support for identifying entries by address ranges in the *[EntryGroups]* section and the `Includes` parameter in *[MemoryMap:*]* sections
- Added the `case` parameter to the *#FORMAT* macro (for converting formatted text to lower case or upper case)
- Added the `DefaultDisassemblyStartAddress` parameter to the *[skoolkit]* section of *skoolkit.ini* (for specifying the address at which to start disassembling a snapshot when no control file is provided)
- Added the `InitModule` parameter to the *[Config]* section (for specifying a Python module to import before the HTML writer class is imported)
- Fixed the bug where a frame whose pixels are modified by the *#PLOT* macro may have incorrect colours when converted to an image
- Fixed the bug where an `M` directive in a control file is ignored when it is followed by a sub-block that has sublengths

7.5 8.4 (2021-03-06)

- Made the *image writer component* pluggable
- Added support for defining groups of entries (via the *[EntryGroups]* section of the ref file) whose disassembly pages can be given custom titles and headers
- Added the `Address` parameter to the *[Game]* section (for specifying the format of address fields on disassembly pages and memory map pages, and of the default link text for the *#R* macro)
- Added the `Length` parameter to the *[Game]* section (for specifying the format of the new `length` attribute of entry objects in *HTML templates*, which is now used instead of `size` in the Length column on *memory map pages*)
- Added the `Peek` and `Word` configuration parameters for *snapinfo.py* (for specifying the format of each line of the output produced by the `--peek` and `--word` options)

- Added support for specifying an *@expand* directive value over multiple lines by prefixing the second and subsequent lines with +
- Added support to the `--ram` option of *tap2sna.py* for the `sysvars` operation (for initialising the system variables in a snapshot)
- Changed the default value of the `DefmSize` configuration parameter for *sna2skool.py* from 66 to 65; this makes it compliant with the default maximum line width of 79 defined by the `LineWidth` configuration parameter
- Fixed the bug that prevents instruction comments from being repeated in a *control file loop*
- Fixed the bug that makes *sna2skool.py* ignore a given start address below 16384 when converting a snapshot

7.6 8.3 (2020-11-08)

- Added the *#PLOT* macro (for setting, resetting or flipping a pixel in a frame already created by an image macro)
- Added the `--begin` option to *bin2tap.py* (for specifying the address at which to begin conversion)
- The `--end` option of *bin2tap.py* now applies to raw memory files as well as SNA, SZX and Z80 snapshots
- Added the `--data` option to *tapinfo.py* (for showing the entire contents of header and data blocks)
- Added support to the `--ctl` option of *sna2skool.py* and *snapinfo.py* for reading control files from a directory
- Added the `x` and `y` parameters to the frame specification of the *#UDGARRAY** macro (for specifying the coordinates at which to render a frame of an animated image)
- Added support for replacement fields in the `args` parameter of the *#CALL* macro, in the integer parameters of the *#CHR*, *#D*, *#INCLUDE*, *#N*, *#POKES*, *#R* and *#SPACE* macros, and in the integer parameters and cropping specification of the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros
- Fixed the bug that causes 'e+1' to be interpreted as a floating point number when it appears in a BASIC program

7.7 8.2 (2020-07-19)

- Added the `--call-graph` option to *snapinfo.py* (for generating a call graph in DOT format)
- Added the `--ctl` option to *snapinfo.py* (for specifying a control file to use when generating a call graph)
- Added the `--org` option to *snapinfo.py* along with the ability to read binary (raw memory) files
- Added support to *snapinfo.py* for reading configuration from *skoolkit.ini*
- Added the `--ini` and `--show-config` options to *snapinfo.py* (for setting the value of a configuration parameter and for showing all configuration parameter values)
- Added the *#DEFINE* macro (for defining new skool macros)
- Added the *#LET* macro (for defining variables that can be retrieved by other macros via replacement fields)
- Added the *#FORMAT* macro (for performing a Python-style string formatting operation on an arbitrary piece of text)
- Added the *@expand* directive (for specifying skool macros to be expanded during ASM writer or HTML writer initialisation)
- Added the `tindex` parameter to the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros (for specifying a transparent colour to use other than the default)

- Added the `alpha` parameter to the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros (for specifying the alpha value to use for the transparent colour)
- Added the *@refs* directive (for managing the addresses of routines that jump to or call an entry point)
- Added support for replacement fields in the integer parameters of the *#FOR* and *#PEEK* macros
- Added the `--page` option to *snapinfo.py* (for specifying the page of a 128K snapshot to map to 49152-65535)

7.8 8.1 (2020-03-29)

- Added the `--rsub` and `--rfix` options to *skool2bin.py* (for parsing the skool file in *@rsub mode* and *@rfix mode*)
- Added the `--data` option to *skool2bin.py* (for processing *@defb*, *@defs* and *@defw* directives)
- Added the `--verbose` option to *skool2bin.py* (for showing information on each converted instruction)
- Added the `--no-warnings` option to *skool2bin.py* (to suppress the warnings that are now shown by default)
- The `address` parameter of the *@defb*, *@defs* and *@defw* directives is now optional
- *@defb*, *@defs* and *@defw* directives in non-entry blocks are now processed when reading a control file
- Register name fields in the registers section of an *entry header* may now contain whitespace and skool macros
- The *#CALL* macro now accepts keyword arguments
- *tapinfo.py* now shows the contents of TZX block types 0x33 (hardware type) and 0x35 (custom info)
- Added the `LabelColumn` parameter to the *[MemoryMap:*]* section (for specifying whether to display the 'Label' column on a memory map page whenever any entries have ASM labels defined)
- Added the `fmt` parameter to the format specifier for the `bytes` attribute of instruction objects in the *asm* template (for formatting the entire string of byte values)
- Added support to the *@set* directive for the *table-row-separator* property
- The *@ignoreua* and *@nowarn* directives can now specify the addresses for which to suppress warnings
- Added support to *sna2skool.py* for ignoring default control files (by specifying `--ctl 0`)
- Fixed how *sna2skool.py* works with dot directives in a control file when an end address is specified

7.9 8.0 (2019-11-09)

- Dropped support for Python 3.4
- Made several *SkoolKit components* pluggable
- Added support for the *foreach*, *if* and *include* directives in HTML templates
- Added the *#PC* macro (which expands to the address of the closest instruction in the current entry)
- Added support to the *@set* directive for the *table-border-horizontal*, *table-border-join* and *table-border-vertical* properties
- Added the `DefwSize` configuration parameter for *sna2skool.py* (for setting the maximum number of words in a DEFW statement)
- Added support for the `**` pattern (which matches any files and zero or more directories and subdirectories) in the *[Resources]* section

- Added support for replacement fields (such as `{base}` and `{case}`) in the parameter string of the *#EVAL* macro
- Added the `max_reg_len` identifier to the *register* template
- Added support for specifying page header prefixes and suffixes in the *[PageHeaders]* section
- An `entry` dictionary is available when formatting the title and header of a disassembly page (as defined by the `Asm-*` parameters in the *[Titles]* and *[PageHeaders]* sections)
- Added the `GameIndex` parameter to the *[PageHeaders]* section
- Replaced the `AsmSinglePageTemplate` parameter with the `AsmSinglePage` parameter in the *[Game]* section
- Fixed the bug that prevents the `JavaScript` parameter from working for a box page whose `SectionType` is `ListItems` or `BulletPoints`
- Fixed how a table row separator that crosses a cell with rowspan > 1 is rendered in ASM mode
- Fixed the bug that prevents *sna2skool.py* from wrapping referrer comments

7.10 Older versions

7.10.1 SkoolKit 7.x changelog

7.2 (2019-06-02)

- Added support to control files for specifying comments over multiple lines (by using the *dot and colon directives*)
- Added support to *skool2ctl.py* for reading configuration from *skoolkit.ini*
- Added the `--ini` and `--show-config` options to *skool2ctl.py* (for setting the value of a configuration parameter and for showing all configuration parameter values)
- Added the `--keep-lines` option to *skool2ctl.py* (for preserving line breaks in comments)
- Added support for *ASM templates* (used to format each line of output produced by *skool2asm.py*)
- Added the `Templates` configuration parameter for *skool2asm.py* (for reading custom ASM templates from a file)
- Added the `Dictionary` configuration parameter for *sna2ctl.py* (to specify a file containing a list of words allowed in a text string)
- Added the `bytes` and `show_bytes` identifiers to the *asm_instruction* template, along with a table cell for displaying the byte values of an assembled instruction
- Added the `Bytes` parameter to the *[Game]* section (for specifying the format of byte values in the *asm_instruction* template)
- Added the `DisassemblyTableNumCols` parameter to the *[Game]* section (for specifying the number of columns in the disassembly table on disassembly pages)
- In ASM mode, *#LIST* and *#TABLE* macros can now be used in register descriptions
- The *#LINK* and *#R* macros now work with address anchors that start with an upper case letter (as could happen when `AddressAnchor` is `{address:04X}`)
- Fixed how `#LIST` and `#TABLE` markers inside a *#RAW* macro are handled in ASM mode

- Fixed how skool macros are expanded in `*ImagePath` parameters in the *[Paths]* section
- Fixed the hyperlinking of lower case hexadecimal instruction operands

7.1 (2019-02-02)

- Improved the performance and accuracy of the control file generation algorithm used by *sna2ctl.py* when no code map is provided
- Added support to *sna2ctl.py* for reading configuration from *skoolkit.ini*
- Added the `--ini` and `--show-config` options to *sna2ctl.py* (for setting the value of a configuration parameter and for showing all configuration parameter values)
- Added support to *sna2skool.py* for reading multiple default control files, and for using the `--ctl` option multiple times
- The *#UDGARRAY* macro now has the ability to specify attribute addresses (as an alternative to specifying attribute values)
- Added support to control files and skool file templates for specifying that numeric values in instruction operands and DEFB, DEFM, DEFS and DEFW statements be rendered as negative numbers
- The *@isub*, *@ssub*, *@rsub*, *@ofix*, *@bfix* and *@rfix* directives can insert an instruction after the current one (without first specifying a replacement for it) by using the + marker
- *tapinfo.py* now shows pulse lengths in TZX block type 0x13 (pulse sequence) and full info for TZX block type 0x14 (pure data)
- *sna2skool.py* handles unprintable characters in a DEFM statement by rendering them as byte values
- *sna2skool.py* automatically determines the byte value of an 'S' directive and ignores any supplied value
- Added the `CommentWidthMin` configuration parameter for *sna2skool.py* (to specify the minimum width of the instruction comment field in a skool file)
- Added the `InstructionWidth` configuration parameter for *sna2skool.py* (to specify the minimum width of the instruction field in a skool file)
- Added the `Semicolons` configuration parameter for *sna2skool.py* (to specify the block types in which comment semicolons are written for instructions that have no comment)
- Fixed how *sna2skool.py* interprets the base prefix `n` in a 'B' directive
- Fixed how *skool2ctl.py* and *skool2sft.py* handle non-entry blocks when a start address or end address is supplied

7.0 (2018-10-13)

- The *@isub*, *@ssub*, *@rsub*, *@ofix*, *@bfix* and *@rfix* directives can specify the replacement comment over multiple lines, replace the label, and insert, overwrite and remove instructions
- *Non-entry blocks* in a skool file are reproduced by *skool2asm.py* and preserved by *skool2ctl.py*
- Moved the ability to generate a control file from *sna2skool.py* to the new *sna2ctl.py* command
- *skool2bin.py* now processes *@if* directives (in case they contain *@isub*, *@ssub*, *@ofix* or *@bfix* directives)
- The *@label* directive can now add an entry point marker to the next instruction, or remove one if present
- Added the `--force` option to *skool2asm.py* (to force conversion of the entire skool file, ignoring any *@start* and *@end* directives)

- Added support for appending content to an existing ref file section by adding a '+' suffix to the section name (e.g. `[Game+]`)
- Added support for preserving 'inverted' characters (with bit 7 set) in and restoring them from a control file or skool file template
- Added support to the *#LIST*, *#TABLE* and *#UDGTABLE* macros for the `nowrap` and `wrapalign` flags (which control how *sna2skool.py* renders each list item or table row when reading from a control file)
- *skool2html.py* now writes a single disassembly from the the skool file given as the first positional argument; any other positional arguments are interpreted as extra ref files
- Every entry title on a memory map page is now hyperlinked to the disassembly page for the corresponding entry
- Fixed the bug in *skool2ctl.py* that makes it incorrectly compute the length of an `M` directive covering a sub-block containing two or more instructions
- Fixed how blocks of zeroes are detected and how an `--end` address is handled when generating a control file

7.10.2 SkoolKit 6.x changelog

6.4 (2018-03-31)

- Added the *@if* directive (for conditionally processing other ASM directives)
- Added the *#RAW* macro (which prevents any macros or macro-like tokens in its sole string argument from being expanded)
- Added the `--var` option to *skool2asm.py* and *skool2html.py* (for defining a variable that can be used by the *@if* directive and the *#IF* and *#MAP* macros)
- The `asm` replacement field available to the *#IF* and *#MAP* macros now indicates the exact ASM mode: 1 (*@isub mode*), 2 (*@ssub mode*), 3 (*@rsub mode*), or 0 (none)
- The *#IF* and *#MAP* macros can now use the `fix` replacement field, which indicates the fix mode: 1 (*@ofix mode*), 2 (*@bfix mode*), 3 (*@rfix mode*), or 0 (none)
- The *@isub*, *@ssub*, *@rsub*, *@ofix*, *@bfix* and *@rfix* directives can replace comments as well as instructions
- Added the `entry` identifier to the *footer* template when it is part of a disassembly page
- Added `path` to the `SkoolKit` dictionary in *HTML templates*
- In ASM mode, a *#LIST* or *#TABLE* macro can now be used in an instruction-level comment and as a parameter of another macro
- In ASM mode, the *#LIST* macro produces unindented items when the bullet character is an empty string, and the bullet character can be specified by the `bullet` parameter
- Commas that appear between parentheses are retained when a sequence of *string parameters* is split, making it easier to nest macros (e.g. `#FOR0,9(n,#IF(n%2)(Y,N))`)

6.3 (2018-02-19)

- Added the *@defb*, *@defs* and *@defw* directives (for inserting byte values and word values into the memory snapshot)
- Added the *@remote* directive (for creating a remote entry)
- Added the `--poke` option to *bin2sna.py* (for performing POKE operations on the snapshot)
- Added the `--user-agent` option to *tap2sna.py* (for setting the User-Agent header used in an HTTP request)
- Added support to the *[Resources]* section for specifying files using wildcard characters (`*`, `?` and `[]`)
- Added the `ImagePath` parameter to the *[Paths]* section (for specifying the base directory in which to place images) and the ability to define one image path ID in terms of another
- Added support for image path ID replacement fields in the `fname` parameter of the *image macros* (e.g. `#SCR2({UDGImagePath}/scr)`)
- The *@assemble* directive can specify what to assemble in HTML mode and ASM mode separately
- By default in ASM mode, DEFB/DEFM/DEFS/DEFW statements are no longer converted into byte values for the purpose of populating the memory snapshot
- The `address` parameter of the *@org* directive is now optional and defaults to the address of the next instruction
- The `LABEL` parameter of the *@label* directive may be left blank to prevent the next instruction from having a label automatically generated
- Added the `location` identifier to the *asm_instruction* template
- Added support for parsing block-level comments that are not left-padded by a space
- Fixed how an opening brace at the end of a line or a closing brace at the beginning of a line is handled in an instruction-level comment
- Fixed the bug in *skool2ctl.py* that prevents an *@ignoreua* directive on a block end comment from being preserved correctly
- Fixed *sna2skool.py* so that it can generate a control file for a snapshot whose final byte (at 65535) is 24 or 237

6.2 (2018-01-01)

- Added the `--reg` option to *bin2sna.py* (for setting the value of a register)
- Added the `--state` option to *bin2sna.py* (for setting the value of a hardware state attribute)
- *sna2img.py* can now read a binary (raw memory) file when the `--binary` option is used, and with a specific origin address when the `--org` option is used
- Added the `Includes` parameter to the *[MemoryMap:*]* section (for specifying addresses of entries to include on the memory map page in addition to those specified by the `EntryTypes` parameter)
- The *SkoolKit command* options now accept a hexadecimal integer prefixed by '0x' wherever an address, byte, length, step, offset or range limit value is expected
- Added the `hex` parameter to the *#N* macro (for rendering a value in hexadecimal format unless the `--decimal` option is used with *skool2asm.py* or *skool2html.py*)
- Added the `--show-config` option to *skool2asm.py*, *skool2html.py* and *sna2skool.py* (for showing configuration parameter values)

- Added support for substituting labels in instruction operands and DEFB/DEFM/DEFW statements that contain multiple addresses (e.g. `LD BC,30000+40000%256`), or where the address is the second or later term in an expression (e.g. `DEFW 1+30000`)
- The *@keep* directive can now specify the values to keep, and is applied to instructions that have been replaced by an *@isub*, *@ssub* or *@rsub* directive
- The `@nolabel` directive is now processed in HTML mode

6.1 (2017-09-03)

- Added support for converting the base of every numerical term in an instruction operand or DEFB/DEFM/DEFS/DEFW statement that contains two or more (e.g. `LD A,32768/256` to `LD A,$8000/$100`)
- Added support for assembling instructions and DEFB/DEFM/DEFS/DEFW statements whose operands contain arithmetic expressions (e.g. `DEFM "H","i"+$80`)
- Added support to *skool2asm.py*, *skool2html.py* and *sna2skool.py* for reading configuration from a file named *skoolkit.ini*, if present
- Added the `--ini` option to *skool2asm.py*, *skool2html.py* and *sna2skool.py* (for setting the value of a configuration parameter)
- *sna2img.py* can now read skool files, in either the default mode, or `@bfix` mode by using the `--bfix` option
- Added the `--move` option to *sna2img.py* (for copying the contents of a block of RAM to another location)
- Improved how *skool2asm.py* formats a comment that covers two or more instructions: now the comment is aligned to the widest instruction, and even blank lines are prefixed by a semicolon
- Improved how the *#R* macro renders the address of an unavailable instruction (an instruction outside the range of the current disassembly, or in another disassembly) in ASM mode
- Removed the indent from EQU directives in ASM output (for compatibility with SjASMPlus)
- Fixed the bug that prevents the expansion of a macro whose numeric parameters contain a '<', '>' or '&' character
- Fixed how labels are substituted for addresses in DEFB/DEFM/DEFW statements
- Fixed *skool2asm.py* so that it processes `@ssub` directives when `--fixes 3` is specified
- Fixed the styling of entry descriptions for 't' blocks on a memory map page

6.0 (2017-05-06)

- Dropped support for Python 2.7 and 3.3
- Added the `--expand` option to *sna2img.py* (for expanding a *#FONT*, *#SCR*, *#UDG* or *#UDGARRAY* macro)
- Added the `--basic` option to *tapinfo.py* (for listing the BASIC program in a tape block)
- Added the `--find-tile` option to *snapinfo.py* (for searching for the graphic data of a tile currently on screen)
- Added the `--word` option to *snapinfo.py* (for showing the words at a range of addresses)
- Added support to the `--find` option of *snapinfo.py* for specifying a range of distances between byte values (e.g. `--find 1,2,3-1-10`)
- The `--peek` option of *snapinfo.py* now shows UDGs and BASIC tokens

- Added support for replacement fields (such as `{base}` and `{case}`) in the `expr` parameter of the *#IF* macro and the `key` parameter of the *#MAP* macro
- Added support for parsing a *box page* entry section as a sequence of multi-line list items prefixed by '-' (with `SectionType=BulletPoints`)
- The following ref file components may now contain skool macros: the `anchor` and `title` of a *box page* entry section name; every parameter in the *[Game]*, *[MemoryMap:*]*, *[Page:*]*, *[PageHeaders]*, *[Paths]* and *[Titles]* sections
- The *@replace* directive now acts on ref file section names as well as their contents
- The *#EVAL* macro now renders hexadecimal values in lower case when the `--lower` option of *skool2asm.py* or *skool2html.py* is used
- Added the *#VERSION* macro (which expands to the version of SkoolKit)
- Fixed how an image is cropped when the crop rectangle is very narrow
- Fixed how a masked image with flashing cells is built
- Fixed how *sna2skool.py* handles a snapshot that contains a dangling IX/IY prefix (DD/FD) when generating a control file
- Fixed the bug that prevents the expansion of skool macros in a page's link text on the disassembly home page

7.10.3 SkoolKit 5.x changelog

5.4 (2017-01-08)

- Added the *sna2img.py* command (for converting the screenshot in a SCR file or SNA/SZX/Z80 snapshot into a PNG or GIF file)
- Added the *@equ* ASM directive (which produces an EQU directive in the ASM output)
- The *#REG* macro now accepts an arbitrary text parameter (e.g. `#REG(hlh'l')`)
- When the *#LINK* macro links to an entry on a *box page*, the link text defaults to the title of the entry if left blank
- Added the `SectionType` parameter to the *[Page:*]* section (for specifying how to parse and render the ref file sections from which a *box page* is built)
- Added the `--asm-one-page` option to *skool2html.py* (for writing all routines and data blocks to a single page)
- Added the `--variables` option to *snapinfo.py* (for showing the contents of the variables area)
- *snapinfo.py* now shows special symbols for UDGs in a BASIC program (e.g. `{UDG-A}`)
- Improved how *@end*, *@org*, *@replace*, *@set*, *@start* and *@writer* directives are preserved and restored via a control file
- Added support for *page-specific HTML subtemplates*
- The *#UDGARRAY* macro now pads out the bottom row of an array with extra UDGs if necessary (to prevent the creation of a broken image file)

5.3 (2016-09-05)

- Dropped support for Python 3.2
- Added the *snapinfo.py* command (for showing information on the registers and RAM in a SNA, SZX or Z80 snapshot)
- Added the *snapmod.py* command (for modifying the registers and RAM in a 48K Z80 snapshot)
- Added the *#INCLUDE* macro (which expands to the contents of a ref file section)
- Added the ability to write the HTML disassembly to a single page (by using the `AsmSinglePageTemplate` parameter in the *[Game]* section and the *AsmAllInOne* and *asm_entry* templates)
- Added the `SectionPrefix` parameter to the *[Page:*]* section (for specifying the prefix of the names of ref file sections from which to build a *box page*)
- Added the `--screen` option to *bin2tap.py* (for adding a loading screen to the TAP file)
- Added the `--stack` and `--start` options to *tap2sna.py* (for specifying the stack and start addresses)
- Added support to the *#REG* macro for the F and F' registers
- Improved how *skool2asm.py* scans annotations for addresses not converted to labels
- Fixed how a memory block that ends with a single ED byte is compressed in a Z80 snapshot
- Removed the Spectrum ROM disassembly from the SkoolKit distribution; it is now being developed separately here

5.2 (2016-05-02)

- Added the *bin2sna.py* command (for converting a binary file into a Z80 snapshot)
- Added the *#N* macro (which renders a numeric value in hexadecimal format when the `--hex` option is used with *skool2asm.py* or *skool2html.py*)
- Added the *@rfix* ASM directive (which makes an instruction substitution in `@rfix` mode)
- Added the `UDGFilename` parameter to the *[Game]* section (for specifying the format of the default filename for images created by the *#UDG* macro)
- *bin2tap.py* can now read a binary file from standard input
- *skool2bin.py* can now write to standard output (and so its output can be piped to *bin2sna.py* or *bin2tap.py*)
- When the *#LINK* macro links to an entry on a memory map page, the anchor is converted to the format specified by the `AddressAnchor` parameter
- Fixed how required integer macro parameters are handled when left blank (e.g. `#POKES30000,,8`)

5.1 (2016-01-09)

- Added the *@replace* ASM directive (which replaces strings that match a regular expression in skool file annotations and ref file sections)
- Added the *#()*, *#EVAL*, *#FOR*, *#FOREACH*, *#IF*, *#MAP* and *#PEEK* macros (which can be used to programmatically specify the parameters of any macro)
- Added support for arithmetic expressions and skool macros in numeric macro parameters
- Added the `--bfix`, `--ofix` and `--ssub` options to *skool2bin.py* (for parsing the skool file in `@bfix`, `@ofix` and `@ssub` mode)

- Added the `DefaultAnimationFormat` parameter to the *[ImageWriter]* section (for specifying the default format for animated images)
- The *#R* macro now converts an anchor that matches the entry address to the format specified by the `AddressAnchor` parameter (making it easier to link to the first instruction in an entry when using a custom anchor format)
- *skool2ctl.py* now appends a terminal `i` directive if the skool file ends before 65536
- *skool2sft.py* now preserves `i` blocks in the same way as code and data blocks (instead of verbatim), which enables their conversion to decimal or hexadecimal when restored from a skool file template
- Fixed how the colours in flashing blank tiles are detected when writing an uncropped image file
- Fixed how a 2-colour PNG image is created when it contains an attribute with equal INK and PAPER colours

5.0 (2015-10-04)

- Added the *skool2bin.py* command (for converting a skool file into a binary file)
- Added the *tapinfo.py* command (for showing information on the blocks in a TAP or TZX file)
- Converted the *HTML templates* from XHTML 1.0 to HTML5
- Added the *footer* template (for formatting the `<footer>` element of a page)
- Added the *@assemble* ASM directive
- Added the `--set` option to *skool2asm.py* (for setting ASM writer property values)
- Added the `RefFiles` parameter to the *[Config]* section (for specifying extra ref files to use)
- Added support to *sna2skool.py* for reading SpecEmu's 64K code execution map files
- Fixed how *tap2sna.py* does a standard load from a TZX file

7.10.4 SkoolKit 4.x changelog

4.5 (2015-07-04)

- Added support to *tap2sna.py* for TZX block type 0x14 (pure data), for loading the first and last bytes of a tape block (which are usually, but not always, the flag and parity bytes), and for modifying memory with XOR and ADD operations
- Added the `--clear` option to *bin2tap.py* (to use a CLEAR command in the BASIC loader and leave the stack pointer alone)
- Added the `--end` option to *bin2tap.py* and the ability to convert SNA, SZX and Z80 snapshots
- Added `--start` and `--end` options to *skool2asm.py*, *skool2ctl.py* and *skool2sft.py*
- The `--start` and `--end` options of *sna2skool.py* now take effect when reading a control file or a skool file template
- Added support to *skool2ctl.py* and *skool2sft.py* for preserving characters in DEFW statements (e.g. `DEFW "!"`)
- Added support for characters in DEFS statements (e.g. `DEFS 10,"!"`)
- Fixed how *tap2sna.py* compresses a RAM block that contains a single `ED` followed by five or more identical values (e.g. `ED0101010101`)
- Fixed the erroneous replacement of DEFS operands with labels

- Fixed how instruction-level comments that contain braces are restored from a control file
- Fixed the handling of terminal compound sublengths on 'S' directives (e.g. `S 30000,10,5:32`)

4.4 (2015-05-23)

- Added support to control files and skool file templates for specifying that numeric values in instruction operands be rendered as characters or in a specific base
- Added support for *@ssub block directives*
- Added the `--end` option to *sna2skool.py* (for specifying the address at which to stop disassembling)
- Added the `--ctl-hex-lower` option to *sna2skool.py* (for writing addresses in lower case hexadecimal format in the generated control file)
- Added the `--hex-lower` option to *skool2ctl.py* and *skool2sft.py* (for writing addresses in lower case hexadecimal format)
- Fixed the parsing of DEFB and DEFM statements that contain semicolons
- Fixed the base conversion of `LD (HL),n` instructions that contain extraneous whitespace (e.g. `LD ( HL ),5`)
- Fixed the erroneous replacement of RST operands with labels in HTML output
- Fixed the handling of uncompressed version 1 Z80 snapshots by *sna2skool.py*

4.3 (2015-02-14)

- Added support for block start comments (which appear after the register section and before the first instruction in a routine or data block)
- Added the `CodeFiles` parameter to the *[Paths]* section (for specifying the format of a disassembly page filename based on the address of the routine or data block)
- Added the `AddressAnchor` parameter to the *[Game]* section (for specifying the format of the anchors attached to instructions on disassembly pages and entries on memory map pages)
- The *#FONT*, *#SCR* and *#UDG* macros now have the ability to create frames for an animated image
- Added the `--line-width` option to *sna2skool.py* (for specifying the maximum line width of the skool file)
- Writing an ASM directive in a skool file can now be done by starting a line with `@`; writing an ASM directive by starting a line with `; @` is deprecated
- Added the `@` directive for declaring ASM directives in a control file; the old style of declaring ASM directives (`; @directive:address[=value]`) is deprecated
- Fixed the *flip_udgs()* and *rotate_udgs()* methods on HtmlWriter so that they work with a UDG array that contains the same UDG in more than one place
- Fixed the bug that prevents register descriptions from being HTML-escaped
- Fixed the erroneous substitution of address labels in instructions that have 8-bit numeric operands

4.2 (2014-12-07)

- Added support for *control directive loops* using the `L` directive
- Added support to control files for preserving the location of *@ignoreua* directives
- Each *image macro* now has the ability to specify alt text for the `<img>` element it produces
- Added support for splitting register descriptions over multiple lines
- *skool2asm.py* now warns about unconverted addresses in register descriptions, and the *@ignoreua* directive can be used to suppress such warnings
- Added the *table*, *table_cell*, *table_header_cell* and *table_row* templates (for formatting tables produced by the *#TABLE* macro)
- Added the *list* and *list_item* templates (for formatting lists produced by the *#LIST* macro)
- Fixed the bug that prevents the expansion of skool macros in the intro text of a `Changelog:*` section

4.1.1 (2014-09-20)

- Updated links to SkoolKit's new home at skoolkit.ca
- Added example control and ref files for Hungry Horace
- Removed the Manic Miner disassembly from the SkoolKit distribution; it is now being developed separately here

4.1 (2014-08-30)

- Added the `--search` option to *skool2html.py* (to add a directory to the resource search path)
- Added the `--writer` option to *skool2html.py* (for specifying the HTML writer class to use)
- Added the `--writer` option to *skool2asm.py* (for specifying the ASM writer class to use)
- Added the `LinkInternalOperands` parameter to the *[Game]* section (for specifying whether to hyperlink instruction operands that refer to an address in the same entry)
- Register sections in `b`, `g`, `s`, `t`, `u` and `w` blocks are now included in the output of *skool2asm.py* and *skool2html.py*
- Fixed how the address '0' is rendered in HTML output when converted to decimal or hexadecimal
- Fixed the bug that creates a broken hyperlink in a DEFW statement or LD instruction that refers to the address of an ignored entry
- Removed the Jet Set Willy disassembly from the SkoolKit distribution; it is now being developed separately here

4.0 (2014-05-25)

- Every HTML page is built from templates defined in *[Template:*]* sections in the ref file
- Added support for keyword arguments to the *#FONT*, *#SCR*, *#UDG* and *#UDGARRAY* macros
- Added the `mask` parameter to the *#UDG* and *#UDGARRAY* macros (for specifying the type of mask to apply)
- Added support for defining page headers in the *[PageHeaders]* section of the ref file
- Added the `--ref-file` and `--ref-sections` options to *skool2html.py* (to show the entire default ref file or individual sections of it)

- Added the `EntryDescriptions` parameter to the *[MemoryMap:*]* section (for specifying whether to display entry descriptions on a memory map page)
- Added the `LengthColumn` parameter to the *[MemoryMap:*]* section (for specifying whether to display the 'Length' column on a memory map page)

7.10.5 SkoolKit 3.x changelog

3.7 (2014-03-08)

- Added support for numbers in binary notation (e.g. %10101010)
- Added the `s` and `S` control directives for encoding DEFS statements (with optional non-zero byte values); the `z` and `Z` directives are now deprecated
- Added support to control files and skool file templates for specifying the base of numeric values in DEFB, DEFM, DEFS and DEFW statements
- Added the `--preserve-base` option to *skool2ctl.py* and *skool2sft.py* (to preserve the base of decimal and hexadecimal values in DEFB, DEFM, DEFS and DEFW statements)
- Added the `JavaScript` parameter to the *[Game]* section (for specifying JavaScript files to include in every page of a disassembly)
- Fixed the bug that prevents DEFB statements containing only strings and DEFM statements containing only bytes from being restored from a control file or a skool file template
- Added changelog entries to *manic_miner.ref*, *jet_set_willy.ref* and *48.rom.ref*

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

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*

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)

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)

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

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

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*

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*)

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

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

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

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*

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

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

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

7.10.6 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*

7.10.7 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

EIGHT

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, some 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:

- *@assemble*
- *@defb*
- *@defs*
- *@defw*
- *@expand*
- *@if*
- *@keep*
- *@label*
- *@remote*
- *@replace*
- *@rom*
- *@bfix block directives*

- *@isub block directives*
- *@ofix block directives*
- *@rfix block directives*
- *@rsub block directives*
- *@ssub 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*, *@ssub*) 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 will be ignored
- `s` indicates a block containing bytes that are all the same value (typically unused zeroes)
- `t` indicates a block containing text
- `u` indicates an unused block of memory
- `w` indicates a block containing words (two-byte values)

(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 will be ignored
- There is a routine at 24576-24831 titled 'Do stuff'
- There is data at 24832-25087
- There is text at 25088-25343
- Everything from 25344 onwards is unused (but will 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
- Block start comments
- 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, declare each one 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 or exit from the routine at 24576, add one line per register with the `R` directive:

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

See the documentation on *entry headers* for more details on how to format a register description line.

8.2.3 Block start comments

To declare a block start comment that will appear above the instruction at 24576, use the `N` directive thus:

```
N 24576 And so this routine begins.
```

If the start comment consists of two or more paragraphs, declare each one with a separate `N` directive:

```
N 24576 This is the first paragraph of the start comment.
N 24576 This is the second paragraph of the start comment.
```

8.2.4 Mid-block comments

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

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

If the mid-block comment consists of two or more paragraphs, declare each one with a separate `N` directive:

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

8.2.5 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, declare each one 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.
```

8.2.6 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
S 32831,17 Followed by 17 zeroes at 32831
```

The directives (`B`, `T`, `C` and `S`) used here to mark the sub-blocks are the upper case equivalents of the directives used to mark top-level blocks (`b`, `t`, `c` and `s`). 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, 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
```

An `M` directive with no length parameter covers all sub-blocks from the given start address to either the next mid-block comment or the end of the containing block (whichever is closer).

To apply the same instruction-level comment to each instruction in a sub-block or group of sub-blocks, use the `M` directive with a third parameter set to 1:

```
c 32768 A routine with careful timing
M 32768,,1 This instruction at #PC takes #TSTATES(#PC) T-states
```

If a sub-block directive is left blank, then it is assumed to be of the same type as the containing block. So in:

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

the sub-block at 24580 is assumed to be of type `C`.

If the length parameter is omitted from a sub-block directive, then it is assumed to end where the next sub-block starts. So in:

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

the sub-block at 24580 has length 8, because it is implicitly terminated by the following sub-block at 24588.

8.2.7 Sub-block lengths

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 "sublengths" following the main length parameter, and the final sublength 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 ; }
```

Note that even if sublengths are specified, the main length parameter can be omitted (by leaving it blank) if the sub-block is implicitly terminated by the next sub-block. For example:

```
B 24580,,1,2,3 No need to specify the main length parameter here...
B 24592,10 ...because this sub-block implies that it must be 12
```

If the sublength 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
```

Sublengths can be used on `C`, `S`, `T` and `W` directives too (though on `C` directives they are really only useful for specifying *number bases*). For example:

```
S 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 ; }
```

8.2.8 The dot and colon directives

The dot (`.`) directive provides an alternative method of specifying a comment for a top-level or sub-block directive. For example, instead of:

```
c 30000 This is the title of the entry
```

you could write:

```
c 30000
. This is the title of the entry
```

At first glance this does not appear to be an improvement. But one advantage of the dot directive is that a comment can be split over multiple lines, and the line breaks are preserved when restored. This makes it much easier to read and write a long comment, especially if it contains a `#LIST` or `#TABLE` macro. For example:

```
D 30000 #TABLE(default) { =h Header 1 | =h Header 2 } { Cell 1      | Cell 2 } TABLE#
```

can be recast like this:

```
D 30000
. #TABLE(default)
. { =h Header 1 | =h Header 2 }
. { Cell 1      | Cell 2 }
. TABLE#
```

In addition, a sequence of `D`, `N`, `E` or `R` directives at the same address (one for each paragraph or register description) can be reduced to just one of those directives followed by a sequence of dot directives:

```
N 30000
. Paragraph 1.
. .
. Paragraph 2.
```

In fact, the dot directive can be used instead of `D`, `R` and `N` directives when specifying an entry header. For example:

```
c 30000
. This is the title of the entry.
.
. This is the description.
.
.   A Input
. O:B Output
.
. This is the start comment.
```

Note, however, that this works only if the entry header contains no ASM directives.

The dot directive also makes it simpler to preserve `@*sub` and `@*fix` directives that replace part of an instruction-level comment. For example, consider the following skool file snippet:

```
 49155 LD A,(HL)     ; {Increase the sprite's x-coordinate by
@bfix=ADD A,3        ; three}
 49156 ADD A,2       ; two (which is a bug)}
```

When preserved without dot directives, this becomes:

```
@ 49156 bfix=ADD A,3        ; three}
C 49155,3 Increase the sprite's x-coordinate by two (which is a bug)
```

which is restored incorrectly by *sna2skool.py* (using the default line width of 79 characters) as:

```
 49155 LD A,(HL)     ; {Increase the sprite's x-coordinate by two (which is a
@bfix=ADD A,3        ; three}
 49156 ADD A,2       ; bug)}
```

This problem could be addressed by recasting the comment lines in the skool file and adding a `@bfix` directive for 'LD A,(HL)':

```
@bfix=               ; {Increase the sprite's x-coordinate by three
 49155 LD A,(HL)     ; {Increase the sprite's x-coordinate by two (which is a
@bfix=ADD A,3        ; }
 49156 ADD A,2       ; bug)}
```

which would be preserved without dot directives as:

```
@ 49155 bfix=               ; {Increase the sprite's x-coordinate by three
@ 49156 bfix=ADD A,3        ; }
C 49155,3 Increase the sprite's x-coordinate by two (which is a bug)
```

But this solution requires two `@bfix` directives instead of one, repeats the part of the comment that doesn't change, and could still be restored incorrectly if *sna2skool.py* is used with a line width other than the default.

It is much easier and more robust to use dot directives to preserve the original form in a way that will always be restored correctly:

```
@ 49156 bfix=ADD A,3        ; three}
C 49155,3
. Increase the sprite's x-coordinate by
. two (which is a bug)
```

Finally, the colon (`:`) directive can be used alongside the dot directive to force an instruction comment continuation line where there would not otherwise be one. For example:

```
B 31995,2,1
. The first two comment lines
: belong to the first DEFB.
. And this comment line belongs to the second DEFB.
```

would be restored as:

```
b31995 DEFB 0        ; {The first two comment lines
                     ; belong to the first DEFB.
 31996 DEFB 0        ; And this comment line belongs to the second DEFB.}
```

The colon directive is rarely needed, but it is useful in cases like the one above where an `@*sub` or `@*fix` directive is used to replace all or part of the comment of the second instruction only:

```
 49155 LD A,(HL)      ; {Having adjusted the sprite's y-coordinate, we now
                      ; increase its x-coordinate by
@bfix=ADD A,3         ; three}
 49156 ADD A,2        ; two (which is a bug)}
```

This can be preserved as:

```
@ 49156 bfix=ADD A,3         ; three}
C 49155,3
. Having adjusted the sprite's y-coordinate, we now
: increase its x-coordinate by
. two (which is a bug)
```

If a dot directive were used instead of the colon directive here, it would restore incorrectly as:

```
 49155 LD A,(HL)      ; {Having adjusted the sprite's y-coordinate, we now
@bfix=ADD A,3         ; three}
 49156 ADD A,2        ; increase its x-coordinate by
                      ; two (which is a bug)}
```

8.2.9 Loops

Sometimes the instructions and statements in a code or data block follow a repeating pattern. For example:

```
b 30000 Two bytes and one word, times ten
B 30000,2
W 30002
B 30004,2
W 30004
...
B 30036,2
W 30038
```

Repeating patterns like this can be expressed more succinctly as a loop by using the `L` directive, which has the following format:

```
L start,length,count[,blocks]
```

where:

- `start` is the loop start address
- `length` is the length of the loop (the size of the address range to repeat)
- `count` is the number of times to repeat the loop (only values of 2 or more make sense)
- `blocks` is `1` to repeat block-level elements, or `0` to repeat only sub-block elements (default: `0`)

So using the `L` directive, the body of the data block above can be expressed in three lines instead of 20:

```
b 30000 Two bytes and one word, times ten
B 30000,2
W 30002
L 30000,4,10
```

The `L` directive can also be used to repeat entire blocks, by setting the `blocks` argument to `1`. For example:

```
b 40000 A block of five pairs of bytes
B 40000,10,2
L 40000,10,3,1
```

is equivalent to:

```
b 40000 A block of five pairs of bytes
B 40000,10,2
b 40010 A block of five pairs of bytes
B 40010,10,2
b 40020 A block of five pairs of bytes
B 40020,10,2
```

Note that ASM directives in the address range of an `L` directive loop are *not* repeated.

8.2.10 Number bases

Numeric values in instruction operands and DEFB, DEFM, DEFS and DEFW statements are normally rendered in either decimal or hexadecimal, depending on the options passed to *sna2skool.py*. To render a numeric value in a specific base, as a negative number, or as a character, attach a `b` (binary), `c` (character), `d` (decimal), `h` (hexadecimal) or `m` (minus) prefix to the relevant length or sublength parameter on the `B`, `C`, `S`, `T` or `W` directive.

For example:

```
C 30000,b
C 30002,c
```

will result in something like this:

```
30000 LD A,%10001111
30002 LD B,"?"
```

and:

```
B 40000,8,b1:d2:h1,m1,b1,h2
S 40008,8,8:c
```

will result in something like this:

```
40000 DEFB %10101010,23,43,$5F
40004 DEFB -1
40005 DEFB %11110000
40006 DEFB $2B,$80
40008 DEFS 8,"!"
```

Note that attaching a prefix to the main length parameter sets the default base for any sublength parameters that follow. So:

```
B 40000,b,1:d2,1
B 40004,h4,1:b1:d1,1
```

will result in something like this:

```
40000 DEFB %01010101,32,57
40003 DEFB %00001111
```

(continues on next page)

(continued from previous page)

```
40004 DEFB $0F,%11110000,93
40007 DEFB $A0
```

Some instructions have two numeric operands. To specify a different base for each one, use two prefixes:

```
C 30000,hb4
```

which will result in something like this:

```
30000 LD (IX+$0A),%10000001
```

To use the default base for one operand, and a specific base for the other, use the `n` (none) prefix to denote the default base. So if the default base is decimal, then:

```
C 30000,,nb4,hn4
```

will result in something like this:

```
30000 LD (IX+10),%10000001
30004 LD (IX+$0B),130
```

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,,3:n1
B 40004,3,1:c2
```

That is, the length of a string in a DEFB statement is prefixed by `c`, the length of a sequence of bytes in a DEFM statement is prefixed by `n`, 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:n2*2
```

which is equivalent to:

```
T 50000,8,2:n2,2:n2
```

A character code may be 'inverted' (i.e. have bit 7 set), typically to indicate the end of a string:

```
49152 DEFM "Hell","o"+128
```

This can be encoded thus:

```
T 49152,5,4:1
```

and the terminal character will be restored in the same format.

8.2.11 ASM directives

To declare an ASM directive for a block or an individual instruction, use the `@` directive thus:

```
@ address directive[=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:

```
@ 32768 label=LOOP
```

When declaring an *@ignoreua* directive for anything other than an instruction-level comment, a suffix must be appended to the directive to specify the type of comment it applies to:

```
@ address ignoreua:X[=addr1[,addr2...]]
```

where `X` is one of:

- `d` - entry description
- `e` - block end comment
- `i` - instruction-level comment (default)
- `m` - block start comment or mid-block comment
- `r` - register description section
- `t` - entry title

For example, to declare an *@ignoreua* directive for the description of the routine at 49152:

```
@ 49152 ignoreua:d
D 49152 This is the description of the routine at 49152.
```

8.2.12 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).

8.2.13 Non-entry blocks

In addition to regular entries (routines and data blocks), a skool file may also contain blocks of lines that do not match the format of an entry, such as a header comment that appears before the first entry and contains copyright information. Blocks like this can be preserved by the > directive. For example, the copyright header in this skool file:

```
; Copyright 2018 J Smith

; Start
c24576 JP 32768
```

is preserved thus:

```
> 24576 ; Copyright 2018 J Smith
```

Note that the address of the > directive is the address of the next regular entry.

A non-entry block may also appear at the end of the skool file, after the last regular entry:

```
; The end
c65535 RET

; And that was the disassembly.
```

In this case the block is preserved by the > directive with the parameter `1` (indicating a 'footer') following the address of the last entry:

```
> 65535,1 ; And that was the disassembly.
```

8.2.14 Quick reference

Block directives

Every block directive has the format:

```
d address[ title]
```

where `address` is the address of the block, and `title` (optional) is its title. The block directive `d` controls how the contents of the block are disassembled by default, and must be one of the following:

- `b` - data block (DEFB statements)
- `c` - code block (assembly language instructions)
- `g` - game status buffer entry (DEFB statements)
- `i` - block that will be ignored
- `s` - block containing bytes that are all the same value (DEFS statement)
- `t` - block containing text (DEFM statements)
- `u` - unused block of memory (DEFB statements)
- `w` - block containing words (DEFW statements)

B directive

The `B` sub-block directive disassembles an address range as one or more DEFB statements:

```
B address[,length[,sublengths]][ comment]
```

- `address` is the start address
- `length` is the length of the address range; if not given, the range ends where the next declared sub-block starts
- `sublengths` controls the DEFB statement lengths and byte value formats; see *Sub-block lengths* and *Number bases* for more details
- `comment` is the comment applied to the sub-block

C directive

The `C` sub-block directive disassembles an address range as code (assembly language instructions):

```
C address[,length[,sublengths]][ comment]
```

- `address` is the start address
- `length` is the length of the address range; if not given, the range ends where the next declared sub-block starts
- `sublengths` controls the instruction operand value formats; see *Number bases* for more details
- `comment` is the comment applied to the sub-block

D directive

The `D` directive declares a description for a code or data block:

```
D address description
```

- `address` is the address of the block
- `description` is the description

See *Block descriptions* for more details.

E directive

The `E` directive declares a block end comment:

```
E address comment
```

- `address` is the address of the block
- `comment` is the comment

See *Block end comments* for more details.

L directive

The `L` directive defines a control file loop that repeats a sequence of other control directives:

```
L start,length,count[,blocks]
```

See *Loops* for more details.

M directive

The `M` directive applies a comment to a contiguous group of sub-blocks:

```
M address[,length[,repeat]] comment
```

- `address` is the start address of the group of sub-blocks
- `length` is the length of the group; if not given, the directive covers all sub-blocks up to either the next mid-block comment or the end of the containing block (whichever is closer)
- `repeat` is `1` to apply the comment to each instruction line in the group, or `0` to apply it to the group as a whole (default: `0`)
- `comment` is the comment

See *Sub-block syntax* for more details.

N directive

The `N` directive declares a block start comment or mid-block comment:

```
N address comment
```

- `address` is the address of the instruction above which to place the comment
- `comment` is the comment

See *Block start comments* and *Mid-block comments* for more details.

R directive

The `R` directive declares an input or output register value for a code block:

```
R address register
```

- `address` is the address of the code block
- `register` is a description of the register name and value

See *Register values* for more details.

S directive

The `S` sub-block directive disassembles an address range as one or more DEFS statements:

```
S address[,length[,sublengths]][ comment]
```

- `address` is the start address
- `length` is the length of the address range; if not given, the range ends where the next declared sub-block starts
- `sublengths` controls the DEFS statement lengths and byte value formats; see *Sub-block lengths* and *Number bases* for more details
- `comment` is the comment applied to the sub-block

T directive

The `T` sub-block directive disassembles an address range as one or more DEFM statements:

```
T address[,length[,sublengths]][ comment]
```

- `address` is the start address
- `length` is the length of the address range; if not given, the range ends where the next declared sub-block starts
- `sublengths` controls the DEFM statement lengths; see *Sub-block lengths* for more details
- `comment` is the comment applied to the sub-block

W directive

The `W` sub-block directive disassembles an address range as one or more DEFW statements:

```
W address[,length[,sublengths]][ comment]
```

- `address` is the start address
- `length` is the length of the address range; if not given, the range ends where the next declared sub-block starts
- `sublengths` controls the DEFW statement lengths and word value formats; see *Sub-block lengths* and *Number bases* for more details
- `comment` is the comment applied to the sub-block

' ' directive

The ' ' (space) sub-block directive is equivalent to a `B`, `C`, `S`, `T` or `W` directive, according to the default disassembly type of the containing block.

See *Block directives* for more details.

@ directive

The `@` directive declares an ASM directive at a given address:

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

See *ASM directives* for more details.

. and : directives

The `.` and `:` directives provide an alternative method of specifying comments for block and sub-block directives that can be used to preserve line breaks.

See *The dot and colon directives* for more details.

> directive

The > directive declares a line of text that lies outside a regular entry (code or data block).

See *Non-entry blocks* for more details.

8.2.15 Control file comments

A comment may be added to a control file by starting a line with a hash character (`#`), a per cent sign (`%`), or a semicolon (`;`). For example:

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

Control file comments are ignored by *sna2skool.py*, and will not show up in the skool file.

8.2.16 Limitations

Control files cannot preserve ASM block directives that occur inside a regular entry. If your skool file contains any such ASM block directives, they should be replaced before using *skool2ctl.py*.

An ASM block directive that adds, removes or modifies a sequence of instructions and their associated comments can be replaced by one or more plain *@isub*, *@ssub*, *@rsub*, *@ofix*, *@bfix* or *@rfix* directives.

An ASM block directive that modifies part of an entry header, mid-block comment or block end comment can be replaced by an *#IF* macro that checks the relevant substitution mode (`asm`) or fix mode (`fix`). For example:

```
@bfix-begin
; This is a bug.
@bfix+else
; This bug is fixed.
@bfix+end
```

could be replaced by:

```
; This #IF({mode[fix]}<2)(is a bug,bug is fixed).
```

8.2.17 Revision history

Version	Changes
8.7	Added support to the `M` directive for applying its comment to each instruction in its range
7.2	Added the dot (`.`) and colon (`:`) directives for specifying comments over multiple lines
7.1	Added support for specifying that numeric values in instruction operands and DEFB, DEFM, DEFS and DEFW statements be rendered as negative numbers
7.0	Added the `>` directive for preserving non-entry blocks; added support for preserving 'inverted' characters (with bit 7 set); the byte value in an `S` directive may be left blank
4.5	Added support for specifying character values in DEFS statements
4.4	Added support for specifying that numeric values in instruction operands be rendered as characters or in a specific base; added support for specifying character values in DEFW statements
4.3	Added the `@` directive, the `N` directive and support for block start comments
4.2	Added the `L` directive and support for preserving the location of *@ignoreua* directives
3.7	Added support for binary numbers; added support for specifying the base of numeric values in DEFB, DEFM, DEFS and DEFW statements; added the `s` and `S` directives and support for DEFS statements with non-zero byte values
3.6	Added support for preserving blank comments that span two or more instructions
3.1.4	Added support for DEFB and DEFM statements that contain both strings and bytes
2.4	Added support for non-block ASM directives
2.2	Added support for the `*` notation in DEFB, DEFM, DEFS and DEFW statement length lists
2.1.2	Added support for DEFM, DEFS and DEFW statement lengths
2.1.1	Added the `M` directive
2.1	Added support for DEFB statement lengths
2.0.6	Added support for hexadecimal numbers
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

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* and *skool2ctl.py*. The rules are as follows:

- entries (an 'entry' being a routine or data block) must be separated by blank lines, and an entry must not contain any blank lines
- an entry header is a sequence of comment lines broken into four sections; see *Entry header format*
- each line in an entry may start with one of the following characters: `;*` `@bcgistuw`; see *Entry line format*
- tables (grids) have their own markup syntax; see *#TABLE* for details

Entry header format

An entry header is a sequence of comment lines broken into four sections:

- entry title
- entry description (optional)
- registers (optional)
- start comment (optional)

The sections are separated by an empty comment line, and paragraphs within the entry description and start comment 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
;
; This is the start comment above the first instruction in the entry.
```

If a start comment is required but a register section is not, either append the start comment to the entry description, or specify a blank register section by using a dot (`.`) thus:

```
; This entry has a start comment but no register section
;
; This is the entry description.
;
; .
;
; This is the start comment above the first instruction in the entry.
```

Likewise, 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 entry has a register section but no description
;
; .
;
; A An important parameter
; B Another important parameter
```

Register names may be given 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.

If a register description is very long, it may be split over two or more lines by starting the second and subsequent lines with a dot (`.`) thus:

```
; HL The description for this register is quite long, so it is split over two
; .  lines for improved readability
```

Note that by default, the register name is separated from the description by whitespace and must not contain skool macros. If whitespace or skool macros are required in the register name field, then it must be delimited in the same way as an arbitrary *string parameter* of a skool macro. For example:

```
; (Output:B, D) The answers are in these two registers
; /(#R32768)/ The result is placed at this address
```

When a register name is supplied in this format, the delimiter characters must be something other than a letter or digit. In addition, *#LIST* and *#TABLE* macros in a register name field are not expanded in ASM mode.

Entry line format

Each line in an entry may start with one of `;* @bcgistuw`, where:

- `;` begins a comment line
- `*` denotes an entry point in a routine
- `@` begins an *ASM directive*
- `b` denotes the first instruction in a data block
- `c` denotes the first instruction in a code block (routine)
- `g` denotes the first instruction in a game status buffer entry
- `i` denotes an ignored entry
- `s` denotes the first instruction in a data block containing bytes that are all the same value (typically unused zeroes)
- `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)
- a space begins a line that does not require any of the markers listed above

The format of a line containing an instruction is:

```
C##### INSTRUCTION[ ; comment]
```

where:

- `C` is one of the characters listed above: `* bcdgirstuw`
- `#####` 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*).

8.3.2 ASM directives

To write an ASM directive in a skool file, start a line with `@`; for example:

```
; Start the game
@label=START
c24576 XOR A
```

See *ASM modes and directives* for more details.

8.3.3 Escaping characters

Backslash (\) and double quote (") characters in string and character operands must be escaped by preceding them with a backslash. For example:

```
c32768 LD A,"\""     ; LD A,34
 32770 LD B,"\\"     ; LD B,92
```

This ensures that SkoolKit or an assembler can parse such operands correctly.

8.3.4 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
```

8.3.5 Non-entry blocks

In addition to regular entries (routines and data blocks), a skool file may also contain blocks of lines that do not match the format of an entry, such as a header comment that appears before the first entry and contains copyright information. For example:

```
; Copyright 2018 J Smith

; Start
c24576 JP 32768
```

Non-entry blocks such as this copyright comment are reproduced by *skool2asm.py*, ignored by *skool2html.py*, and preserved verbatim by *skool2ctl.py*.

To qualify as a regular entry, a block must contain at least one line that starts with `b`, `c`, `g`, `i`, `s`, `t`, `u` or `w` when parsed in the relevant *subtitution mode* or *bugfix mode* (which depends on the command being run).

So, for example:

```
@isub-begin
c24573 JP 32768
@isub-end
```

is seen as a regular entry (without the `@isub` block directives) by *skool2ctl.py* and *skool2html.py*, but is invisible to *skool2asm.py*. And:

```
@isub+begin
c24573 JP 32768
@isub+end
```

is seen as a non-entry block (with the `@isub` block directives retained) by *skool2ctl.py* and *skool2html.py*, but as a regular entry (without the `@isub` block directives) by *skool2asm.py*.

8.3.6 Revision history

Version	Changes
8.1	Register name fields may contain whitespace and *skool macros*
4.3	Added support for the start comment in entry headers; an ASM directive can be declared by starting a line with `@`
4.2	Added support for splitting register descriptions over multiple lines
3.7	Added support for binary numbers; added the `s` block type
3.1.2	Added support for 'Input' and 'Output' prefixes in register sections
2.4	Added the ability to separate paragraphs and specify a blank entry description by using a dot (`.`) on a line of its own
2.1	Added support for hexadecimal numbers

8.4 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).

8.4.1 Syntax

Skool macros have the following general form:

```
#MACROri1,ri2,...[,oi1,oi2,...](rs1,rs2,...[,os1,os2,...])
```

where:

- `MACRO` is the macro name
- `ri1`, `ri2` etc. are required integer parameters
- `oi1`, `oi2` etc. are optional integer parameters
- `rs1`, `rs2` etc. are required string parameters
- `os1`, `os2` etc. are optional string parameters

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

```
#UDG39144
```

is equivalent to:

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

and:

```
#UDG30115,,2
```

is equivalent to:

```
#UDG30115,56,2
```

8.4.2 Numeric parameters

Numeric parameters may be written in decimal notation:

```
#UDG51673,17
```

or in hexadecimal notation (prefixed by `$`):

```
#UDG$C9D9,$11
```

Wherever a sequence of numeric parameters appears in a macro, that sequence may optionally be enclosed in parentheses: `(` and `)`. Parentheses are generally recommended - especially when there are two or more parameters - in order to unambiguously separate the numeric parameters from any content that follows them. Parentheses are *required* if any numeric parameter is written as an expression containing arithmetic operations, skool macros or replacement fields:

```
#UDG(51672+{offset},#PEEK51672)
```

The following operators are permitted in an arithmetic expression:

- arithmetic operators: `+`, `-`, `*`, `/`, `%` (modulo), `**` (power)
- bitwise operators: `&` (AND), `|` (OR), `^` (XOR)
- bit shift operators: `>>`, `<<`
- Boolean operators: `&&` (and), `||` (or)
- comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

Parentheses and spaces are also permitted in an arithmetic expression:

```
#IF(1 == 2 || (1 <= 2 && 2 < 3))(Yes,No)
```

8.4.3 String parameters

Where a macro requires a single string parameter consisting of arbitrary text, it must be enclosed in parentheses, square brackets or braces:

```
(text)
[text]
{text}
```

If `text` contains unbalanced brackets, a non-whitespace character that is not present in `text` may be used as an alternative delimiter. For example:

```
/text/
|text|
```

Where a macro requires multiple string parameters consisting of arbitrary text, they must be enclosed in parentheses, square brackets or braces and be separated by commas:

```
(string1,string2)
[string1,string2]
{string1,string2}
```

When a comma-separated sequence of string parameters is split, any commas that appear between parentheses are retained. For example, the string parameters of the outer `#FOR` macro in:

```
#FOR0,1(n,#FOR(0,1)(m,(n,m),;),;)
```

are split into `n`, `#FOR(0,1)(m,(n,m),;)` and `;`, and the string parameters of the inner `#FOR` macro are split into `m`, `(n,m)`, and `;`.

Alternatively, an arbitrary delimiter - `d`, which cannot be whitespace - and separator - `s`, which can be whitespace - may be used. (They can be the same character.) The string parameters must open with `ds`, be separated by `s`, and close with `sd`. For example:

```
//same/delimiter/and/separator//
| different delimiter and separator |
```

Note that if an alternative delimiter or separator is used, it must not be '&', '<' or '>'.

Changed in version 6.4: When a comma-separated sequence of string parameters is split, any commas that appear between parentheses are retained.

8.4.4 Replacement fields

The following replacement fields are available for use in the integer parameters of the *@if* directive and every skool macro (including macros defined by *#DEF* or *#DEFINE*), and also in the string parameters of some macros:

- `asm` - 1 if in *@isub mode*, 2 if in *@ssub mode*, 3 if in *@rsub mode*, or 0 otherwise
- `base` - 10 if the `--decimal` option is used with *skool2asm.py* or *skool2html.py*, 16 if the `--hex` option is used, or 0 if neither option is used
- `case` - 1 if the `--lower` option is used with *skool2asm.py* or *skool2html.py*, 2 if the `--upper` option is used, or 0 if neither option is used
- `fix` - 1 if in *@ofix mode*, 2 if in *@bfix mode*, 3 if in *@rfix mode*, or 0 otherwise
- `html` - 1 if in HTML mode, 0 otherwise
- `mode` - a dictionary containing a copy of the `asm`, `base`, `case`, `fix` and `html` fields
- `sim` - a dictionary of register values populated by the *#SIM* macro
- `vars` - a dictionary of variables defined by the `--var` option of *skool2asm.py* or *skool2html.py*; accessing an undefined variable in this dictionary yields the integer value '0'

Replacement fields for the variables defined by the *#LET* macro are also available. Note that the `#LET` macro can change the values of the `asm`, `base`, `case`, `fix` and `html` fields, but their original values are always available in the `mode` dictionary.

For example:

```
#IF({mode[case]}==1)(hl,HL)
```

expands to `hl` if in lower case mode, or `HL` otherwise.

Note that if a replacement field is used, the parameter string must be enclosed in parentheses.

Changed in version 8.7: Added the `sim` dictionary.

Changed in version 8.2: Added the `mode` dictionary.

Changed in version 6.4: The `asm` replacement field indicates the exact ASM mode; added the `fix` and `vars` replacement fields.

8.4.5 SMPL macros

The macros described in this section constitute the Skool Macro Programming Language (SMPL). They can be used to programmatically specify values in the parameter string of any macro.

#()

The `#()` macro expands the skool macros in its sole string parameter.

```
#(text)
```

It takes effect only when it immediately follows the opening token of another skool macro, and is expanded *before* that macro. For example:

```
#UDGARRAY#(2(#FOR37159,37168,9||n|(n+1),#PEEKn|;||))(item)
```

This instance of the `#()` macro expands the `#FOR` macro first, giving:

```
2((37159+1),#PEEK37159;(37168+1),#PEEK37168)
```

It then expands the `#PEEK` macros, ultimately forming the parameters of the `#UDGARRAY` macro.

See *String parameters* for details on alternative ways to supply the `text` parameter. Note that if an alternative delimiter is used, it must not be an alphanumeric character (A-Z, a-z, 0-9).

#DEF

The `#DEF` macro defines a new skool macro.

```
#DEF[flags](#MACRO[(ia[=i0],ib[=i1]...)[(sa[=s0],sb[=s1]...)]] body)
```

- `flags` controls various options (see below)
- `MACRO` is the macro name (which must be all upper case letters)
- `ia[=i0]`, `ib[=i1]` etc. are the integer parameter names and optional default values; the parameter names must consist of lower case letters only
- `sa[=s0]`, `sb[=s1]` etc. are the string parameter names and optional default values
- `body` is the body of the macro definition, which may contain placeholders (`$var`, `${var}` - when `flags` is 0) or replacement fields (`{var}` - when `flags` is 1) representing the integer and string argument values

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 - use replacement fields (e.g. `{var}`) instead of $-placeholders (`$var`, `${var}`) to represent the defined macro's argument values
- 2 - strip leading and trailing whitespace from the output of the defined macro whenever it is expanded

For example:

```
#DEF(#MIN(a,b) #IF($a<$b)($a,$b))
```

This defines a `#MIN` macro that accepts two integer arguments and expands to the value of the smaller argument.

Default values for the defined macro's optional integer parameters can be specified in the macro's signature. For example:

```
#DEF(#PROD(a,b=1,c=1) #EVAL($a*$b*$c))
```

This defines a `#PROD` macro that accepts one, two or three integer arguments, the second and third of which default to 1, and expands to the product of all three arguments.

Default values for the defined macro's optional string parameters can also be specified in the macro's signature, and their default values may refer to the integer argument values. For example:

```
#DEF(#NUM(a)(s=$a) $s)
```

This defines a `#NUM` macro that accepts one integer argument and an optional string argument. It expands either to the integer argument, or to the string argument if provided. So `#NUM15` expands to '15', and `#NUM15($0F)` expands to '$0F'.

If `flags` is odd (bit 0 set), replacement fields are used instead of $-placeholders to represent the defined macro's argument values. The main advantage of using replacement fields is that Python string formatting options can be used on the argument values. For example:

```
#DEF1(#HEX(n) {n:04X})
```

This defines a `#HEX` macro that formats its sole integer argument as a 4-digit upper case hexadecimal number.

However, when using replacement fields, care must be taken to escape any field that doesn't represent an argument value. For example:

```
#LET(count=0)
#DEF1(#ADD(amount) #LET(count={{count}}+{amount}))
```

This defines a variable named `count`, and an `#ADD` macro that increases its value by a given amount. Note how the replacement field for the `count` variable in the body of the macro definition is escaped: `{{count}}`.

If bit 1 of `flags` is set, the defined macro will be expanded, in isolation from any surrounding content, as soon as it is encountered. For that to work, the macro definition must be entirely self-contained, i.e. it must not depend on any surrounding content in order to be syntactically correct. For example, if the `#IFZERO` macro is defined thus:

```
#DEF2(#IFZERO(n) #IF($n==0))
```

then any attempt to expand an `#IFZERO` macro will lead to an error message about the `#IF` macro having no output strings. To fix this, either reset bit 1 of `flags`, or redefine `#IFZERO` with the output strings included in the definition:

```
#DEF2(#IFZERO(n)(a,b) #IF($n==0)($a,$b))
```

For more examples, see *Defining macros with #DEF*.

Note that if a string parameter of a defined macro is optional, that argument will take its default value only if it is omitted; if instead it is left blank, it takes the value of the empty string.

In general, the string arguments of a defined macro may be supplied between alternative delimiters (see *String parameters*) if desired. However, if every string parameter of the defined macro is optional, the string arguments must be either omitted entirely or provided between parentheses (and therefore separated by commas). This allows a macro with all of its optional string arguments omitted to be immediately followed by some character other than an opening parenthesis without that character being interpreted as an alternative delimiter.

To define a macro that will be available for use immediately anywhere in the skool file or ref files, consider using the *@expand* directive, or the `Expand` parameter in the *[Config]* section.

The `flags` parameter of the `#DEF` macro may contain *replacement fields*.

The integer parameters of a macro defined by `#DEF` may contain *replacement fields*, and may also be supplied via keyword arguments.

Version	Changes
8.6	Added the `flags` parameter, the ability to use replacement fields to represent the defined macro's argument values, and the ability to strip whitespace from the defined macro's output
8.5	New

#DEFINE

The #DEFINE macro defines a new skool macro.

```
#DEFINEiparams[,sparams](name,value)
```

- iparams is the number of integer parameters the macro expects
- sparams is the number of string parameters the macro expects (default: 0)
- name is the macro name (which must be all upper case letters)
- value is the macro's output value (a standard Python format string containing replacement fields for the integer and string arguments)

For example:

```
#DEFINE2(MIN,#IF({0}<{1})({0},{1}))
```

This defines a #MIN macro that accepts two integer arguments and expands to the value of the smaller argument.

To define a macro that will be available for use immediately anywhere in the skool file or ref files, consider using the *@expand* directive, or the Expand parameter in the *[Config]* section.

The integer parameters of a macro defined by #DEFINE may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the name and value parameters.

Note: The #DEFINE macro is deprecated since version 8.5. Use the more powerful *#DEF* macro instead.

Version	Changes
8.2	New

#EVAL

The #EVAL macro expands to the value of an arithmetic expression.

```
#EVALexpr[,base,width]
```

- expr is the arithmetic expression
- base is the number base in which the value is expressed: 2, 10 (the default) or 16
- width is the minimum number of digits in the output (default: 1); the value will be padded with leading zeroes if necessary

For example:

```
; The following mask byte is #EVAL(#PEEK29435,2,8).
 29435 DEFB 62
```

This instance of the #EVAL macro expands to '00111110' (62 in binary).

The parameter string of the #EVAL macro may contain *replacement fields*.

Version	Changes
8.0	Added support for replacement fields in the parameter string
6.0	Hexadecimal values are rendered in lower case when the `--lower` option is used
5.1	New

#FOR

The `#FOR` macro expands to a sequence of strings based on a range of integers.

```
#FORstart,stop[,step,flags](var,string[,sep,fsep])
```

- `start` is first integer in the range
- `stop` is the final integer in the range
- `step` is the gap between each integer in the range (default: 1)
- `flags` controls whether to affix commas to or replace variable names in each separator (see below)
- `var` is the variable name; for each integer in the range, it evaluates to that integer
- `string` is the output string that is evaluated for each integer in the range; wherever the variable name (`var`) appears, its value is substituted
- `sep` is the separator placed between each output string (default: the empty string); this may be modified depending on the value of `flags`
- `fsep` is the separator placed between the final two output strings (default: `sep`)

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 - prefix each separator (`sep`) with a comma
- 2 - suffix each separator (`sep`) with a comma
- 4 - replace any variable name (`var`) in each separator (`sep`) with the variable value

For example:

```
; The next three bytes (#FOR31734,31736,,1(n,#PEEKn, , and )) define the
; item locations.
 31734 DEFB 24,17,156
```

This instance of the `#FOR` macro expands to '24, 17 and 156'.

The integer parameters of the `#FOR` macro (`start`, `stop`, `step`, `flags`) may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `var`, `string`, `sep` and `fsep` parameters.

Version	Changes
8.7	Added the `flags` parameter
8.2	Added support for replacement fields in the integer parameters
5.1	New

#FOREACH

The `#FOREACH` macro expands to a sequence of output strings based on a sequence of input strings.

```
#FOREACH([s1,s2,...])(var,string[,sep,fsep])
```

or:

```
#FOREACH(svar)(var,string[,sep,fsep])
```

- `s1`, `s2` etc. are the input strings
- `svar` is a special variable that expands to a specific sequence of input strings (see below)
- `var` is the variable name; for each input string, it evaluates to that string
- `string` is the output string that is evaluated for each input string; wherever the variable name (`var`) appears, its value is substituted
- `sep` is the separator placed between each output string (default: the empty string)
- `fsep` is the separator placed between the final two output strings (default: `sep`)

For example:

```
; The next three bytes (#FOREACH(31734,31735,31736)||n|#PEEKn|, | and ||)
; define the item locations.
 31734 DEFB 24,17,156
```

This instance of the `#FOREACH` macro expands to '24, 17 and 156'.

The `#FOREACH` macro recognises certain special variables, each one of which expands to a specific sequence of strings. The special variables are:

- `ENTRY[types]` - the addresses of every entry of the specified type(s) in the memory map; if `types` is not given, every type is included
- `EREFaddr` - the addresses of the routines that jump to or call a given instruction (at `addr`)
- `REFaddr` - the addresses of the routines that jump to or call a given routine (at `addr`), or jump to or call any entry point within that routine

For example:

```
; The messages can be found at #FOREACH(ENTRYt)||n|n|, | and ||.
```

This instance of the `#FOREACH` macro expands to a list of the addresses of the entries of type `t` (text).

See *String parameters* for details on alternative ways to supply the `s1,s2,...` and `var,string[,sep,fsep]` parameter strings.

Version	Changes
5.1	New

#FORMAT

The `#FORMAT` macro performs a Python-style string formatting operation on its string argument.

```
#FORMAT[case](text)
```

- `case` is 1 to convert the formatted string to lower case, 2 to convert it to upper case, or 0 to leave it alone (the default)
- `text` is the string to format

For example:

```
#FORMAT(0x{count:04X})
```

This instance of the `#FORMAT` macro formats the value of the `count` variable (assuming it has already been defined by the *#LET* macro) as a 4-digit upper case hexadecimal number prefixed by '0x'.

Note that if `text` could be read as an integer parameter, `case` should be explicitly specified in order to prevent `text` from being interpreted as the `case` parameter. For example:

```
#FORMAT0({count})
```

Alternatively, the *#EVAL* macro may be a better option for formatting a pure numeric value.

The parameters of the `#FORMAT` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `text` parameter.

Version	Changes
8.5	Added the `case` parameter
8.2	New

#IF

The `#IF` macro expands to an arbitrary string based on the truth value of an arithmetic expression.

```
#IFexpr(true[,false])
```

- `expr` is the arithmetic expression, which may contain *replacement fields*
- `true` is the output string when `expr` is true
- `false` is the output string when `expr` is false (default: the empty string)

For example:

```
; #FOR0,7(n,#IF(#PEEK47134 & 2**(7-n))(X,O))
 47134 DEFB 170
```

This instance of the `#IF` macro is used (in combination with a `#FOR` macro and a `#PEEK` macro) to display the contents of the address 47134 in the memory snapshot in binary format with 'X' for one and 'O' for zero: XOXOXOXO.

See *String parameters* for details on alternative ways to supply the `true` and `false` output strings.

Version	Changes
6.0	Added support for replacement fields in the `expr` parameter
5.1	New

#LET

The `#LET` macro defines an integer, string or dictionary variable.

The syntax for defining an integer or string variable is:

```
#LET(name=value)
```

- `name` is the variable name
- `value` is the value to assign; this may contain skool macros (which are expanded immediately) and *replacement fields* (which are replaced after any skool macros have been expanded)

If `name` ends with a dollar sign (`$`), `value` is interpreted as a string. Otherwise `value` is evaluated as an arithmetic expression.

For example:

```
#LET(count=2*2)
#LET(count$=2*2)
```

These `#LET` macros assign the integer value '4' to the variable `count` and the string value '2*2' to the variable `count$`. The variables are then accessible to other macros via the replacement fields `{count}` and `{count$}`.

The syntax for defining a dictionary variable is:

```
#LET(name[]=(default[,k1[:v1],k2[:v2]...]))
```

- `name` is the dictionary variable name
- `default` is the default value (used when a key is not found in the dictionary)
- `k1:v1`, `k2:v2` etc. are the key-value pairs in the dictionary

The keys in a dictionary are integers, and the associated values are strings if `name` ends with a dollar sign, or integers otherwise. If the value part of a key-value pair is omitted, it defaults to the key.

For example:

```
#LET(n[]=(0,1:10,2:20))
#LET(d$[]=(?,1:a,2:b))
```

The first `#LET` macro defines the dictionary variable `n` with default integer value 0, and keys '1' and '2' mapping to the integer values 10 and 20. The values in this dictionary are accessible to other macros via the replacement fields `{n[1]}` and `{n[2]}`.

The second `#LET` macro defines the dictionary variable `d$` with default string value '?', and keys '1' and '2' mapping to the string values 'a' and 'b'. The values in this dictionary are accessible to other macros via the replacement fields `{d$[1]}` and `{d$[2]}`.

To define a variable that will be available for use immediately anywhere in the skool file or ref files, consider using the *@expand* directive.

See *String parameters* for details on alternative ways to supply the entire `name=value` parameter string, or the part after the equals sign when defining a dictionary variable.

Version	Changes
8.6	Added the ability to define dictionary variables
8.2	New

#MAP

The #MAP macro expands to a value from a map of key-value pairs whose keys are integers.

```
#MAPkey(default[,k1:v1,k2:v2...])
```

- key is the integer to look up in the map; this parameter may contain *replacement fields*
- default is the default output string (used when key is not found in the map)
- k1:v1, k2:v2 etc. are the key-value pairs in the map

For example:

```
; The next three bytes specify the directions that are available from here:
; #FOR56112,56114,,1(q,#MAP(#PEEKq)(?,0:left,1:right,2:up,3:down), , and ).
 56112 DEFB 0,1,3
```

This instance of the #MAP macro is used (in combination with a #FOR macro and a #PEEK macro) to display a list of directions available based on the contents of addresses 56112-56114: 'left, right and down'.

Note that the keys (k1, k2 etc.) may be expressed using arithmetic operations. They may also be expressed using skool macros, but in that case the *entire* parameter string of the #MAP macro must be enclosed by a *#()* macro.

See *String parameters* for details on alternative ways to supply the default output string and the key-value pairs.

Version	Changes
6.0	Added support for replacement fields in the key parameter
5.1	New

#PC

The #PC macro expands to the address of the closest instruction in the current entry.

```
#PC
```

For example:

```
c32768 XOR A ; This instruction is at #PC.
```

This instance of the #PC macro expands to '32768'.

In an entry header (i.e. title, description, register description or start comment), the #PC macro expands to the address of the first instruction in the entry. In a mid-block comment, the #PC macro expands to the address of the following instruction. In an instruction-level comment, the #PC macro expands to the address of the instruction. In a block end comment, the #PC macro expands to the address of the last instruction in the entry.

Version	Changes
8.0	New

#PEEK

The `#PEEK` macro expands to the contents of an address in the internal memory snapshot constructed from the contents of the skool file.

```
#PEEKaddr
```

- `addr` is the address, which may contain *replacement fields*

For example:

```
; At the start of the game, the number of lives remaining is #PEEK33879.
```

This instance of the `#PEEK` macro expands to the contents of the address 33879 in the internal memory snapshot.

Note that, by default, the internal memory snapshot constructed by *skool2asm.py* is entirely blank (all zeroes), and the snapshot constructed by *skool2html.py* is populated only by `DEFB`, `DEFM`, `DEFS` and `DEFW` statements, and by *@defb*, *@defs* and *@defw* directives. To change this behaviour, use the *@assemble* directive.

See also *#POKES*.

Version	Changes
8.2	Added support for replacement fields in the `addr` parameter
5.1	New

#STR

The `#STR` macro expands to the text string at a given address in the memory snapshot.

```
#STRaddr[,flags,length][(end)]
```

- `addr` is the address of the first character in the string
- `flags` indicates operations to be performed on the string (default: 0)
- `length` is the number of characters in the string; if -1 (the default), the string ends immediately before the first zero byte, or on the first byte that has bit 7 set (bit 7 of that byte will be reset before converting it to a character), or when `end` evaluates to true
- `end` is an arithmetic expression that identifies the end marker byte for the string (when bit 3 of `flags` is set)

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 - strip trailing whitespace from the string
- 2 - strip leading whitespace from the string
- 4 - replace each sequence of N>=2 spaces in the string with `#SPACE(N)` (see *#SPACE*)
- 8 - use the `end` parameter to determine where the string ends

When bit 3 of `flags` is set, `end` is evaluated for each byte encountered, and if the result is true, the string is terminated. `end` may contain the placeholder `$b` for the current byte value.

For example:

```
; The messages here are '#STR47154', '#STR47158' and '#STR47161,8($b==255)'.
 47154 DEFM "One",0
 47158 DEFM "Tw","o"+128
 47161 DEFM "Three",255
```

These instances of the #STR macro expand to 'One', 'Two' and 'Three'.

The parameters of the #STR macro may contain *replacement fields*.

Version	Changes
8.6	New

#WHILE

The #WHILE macro repeatedly expands macros while a conditional expression is true.

```
#WHILE(expr)(body)
```

- expr is the conditional expression
- body is the text to repeatedly expand; leading and trailing whitespace are stripped from the expanded value

For example:

```
#LET(a=3)
#WHILE({a}>0)(
  #EVAL({a})
  #LET(a={a}-1)
)
```

This instance of the #WHILE macro expands to '321'.

The expr parameter of the #WHILE macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the body parameter.

Version	Changes
8.6	New

8.4.6 General macros

#AUDIO

In HTML mode, the #AUDIO macro expands to an HTML5 <audio> element.

```
#AUDIO[flags,offset](fname)[(delays)]
```

Or, when bit 2 of flags is set:

```
#AUDIO[flags,offset](fname)(start,stop)
```

- flags controls various options (see below)
- offset is the initial offset in T-states from the start of a frame (default: 0); this value affects when contention and interrupt delays (if enabled) first take effect
- fname is the name of the audio file
- delays is a comma-separated list of delays (in T-states) between speaker state changes; this parameter may contain skool macros (which are expanded first) and *replacement fields* (which are replaced after any skool macros have been expanded)

- `start` is the address at which to start executing code in a simulator (when bit 2 of `flags` is set)
- `stop` is the address at which to stop executing code in a simulator (when bit 2 of `flags` is set)

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 (bit 0) - modify delays to approximate the effect of running in contended memory; this increases any delays that occur during the contended period of a frame by a given factor (as specified by the `ContentionBegin`, `ContentionEnd` and `ContentionFactor` parameters in the *[AudioWriter]* section)
- 2 (bit 1) - modify delays as if interrupts were enabled; this increases any delays that occur over a frame boundary by a given number of T-states (as specified by the `InterruptDelay` parameter in the *[AudioWriter]* section)
- 4 (bit 2) - execute instructions from `start` to `stop` in a simulator to obtain the delays between speaker state changes

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 audio directory (as defined by the `AudioPath` parameter in the *[Paths]* section).

If `delays` is specified and `fname` ends with '.wav', a corresponding audio file in WAV format is created. Each element in `delays` can be an integer, a list or tuple of integers, or a list/tuple of lists/tuples of integers etc. nested to arbitrary depth, expressed as Python literals. For example:

```
1000, [1500]*100, [(800, 1200)*2, 900]*200
```

This would be flattened into a list of integers, as follows:

- a single instance of '1000'
- 100 instances of '1500'
- 200 instances of the sequence '800, 1200, 800, 1200, 900'

The sum of this list of integers being 1131000, this would result in an audio file of duration 1131000 / 3500000 = 0.323s (assuming that no memory contention is simulated and interrupts are disabled, i.e. bits 0 and 1 of `flags` are reset).

The characters allowed in the `delays` parameter are ' ' (space), newline, the digits 0-9, and any of `,*+-%()[]`.

An alternative to supplying the delay values manually is to execute the code that produces the sound effect in a simulator, and let the simulator compute the delays. This can be done by setting bit 2 of `flags` and specifying the code to execute via the `start` and `stop` address parameters. For example:

```
; #AUDIO4(beep.wav)(32768,32782)
@assemble=2
c32768 LD L,0
*32771 OUT (254),A
 32773 XOR 16
 32775 LD B,200
 32777 DJNZ 32777
 32779 DEC L
 32780 JR NZ,32771
 32782 RET
```

Note: The simulator does not simulate memory contention, I/O contention, or interrupts. Use bits 0 and 1 of `flags` to approximate memory contention effects and interrupt delays if desired.

Note also that, by default, the internal memory snapshot constructed by *skool2asm.py* is entirely blank (all zeroes), and the snapshot constructed by *skool2html.py* is populated only by `DEFB`, `DEFM`, `DEFS` and `DEFW` statements, and

by *@defb*, *@defs* and *@defw* directives. To make sure that the internal memory snapshot actually contains the code to be executed, use the *@assemble* directive (as shown in the example above).

If `delays` or `start` and `stop` parameters are specified, but `fname` does not end with '.wav', no audio file is written. This enables the parameters to be kept in place as a reminder of how an original WAV file was created by the `#AUDIO` macro before it was converted to another format.

If neither `delays` nor `start` and `stop` parameters are specified, or `fname` does not end with '.wav', the named audio file must already exist in the specified location, otherwise the `<audio>` element controls will not work. To make sure that a pre-built audio file is copied into the desired location when *skool2html.py* is run, it can be declared in the *[Resources]* section.

By default, if `fname` ends with '.wav', but a '.flac', '.mp3' or '.ogg' file with the same basename already exists, that file is used and no WAV file is written. This enables an original WAV file to be replaced by an alternative (compressed) version without having to modify the `fname` parameter of the `#AUDIO` macro. The alternative audio file types that the `#AUDIO` macro looks for before writing a WAV file are specified by the `AudioFormats` parameter in the *[Game]* section.

The `flags`, `offset`, `start` and `stop` parameters of the `#AUDIO` macro may contain *replacement fields*.

The *audio* template is used to format the `<audio>` element.

Audio file creation can be configured via the *[AudioWriter]* section.

Version	Changes
8.7	New

#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, which may contain *replacement fields*

Each argument can be expressed either as a plain value (e.g. `32768`) or as a keyword argument (e.g. `address=32768`).

For example:

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

This instance of the `#CALL` macro expands to the return value of the *word* 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
8.3	Added support for replacement fields in the `args` parameter
8.1	Added support for keyword arguments
5.1	Added support for arithmetic expressions and skool macros in the `args` parameter
3.1	Added support for ASM mode
2.1	New

#CHR

In HTML mode, the `#CHR` macro expands either to a numeric character reference, or to a unicode character in the UTF-8 encoding. In ASM mode, it always expands to a unicode character in the UTF-8 encoding.

```
#CHRnum[,flags]
```

- `num` is the character code
- `flags` enables options that control the output (default: 0)

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 - produce a character in the UTF-8 encoding instead of a numeric character reference in HTML mode
- 2 - map character code 94 to 8593 ('↑'), 96 to 163 ('£'), and 127 to 169 ('©'), in accordance with the ZX Spectrum character set

For example:

```
26751 DEFB 127   ; This is the copyright symbol: #CHR169
26572 DEFB 127   ; This is also the copyright symbol: #CHR127,2
```

In HTML mode, these instances of the `#CHR` macro expand to '©'. In ASM mode, they both expand to '©'.

The parameter string of the `#CHR` macro may contain *replacement fields*.

Version	Changes
8.6	Added the `flags` parameter, the ability to use UTF-8 encoding in HTML mode, and support for mapping character codes 94, 96 and 127 to '↑', '£' and '©'
8.3	Added support for replacement fields in the parameter string
5.1	Added support for arithmetic expressions and skool macros in the parameter string
3.1	New

#D

The `#D` 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, which may contain *replacement fields*

For example:

```
; Now we make an indirect jump to one of the following routines:
; .
; #TABLE(default,centre)
```

(continues on next page)

(continued from previous page)

```
; { =h Address | =h Description }
; { #R27126    | #D27126 }
```

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

Version	Changes
8.3	Added support for replacement fields in the `addr` parameter
5.1	Added support for arithmetic expressions and skool macros in the `addr` parameter

#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>.)
```

`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 *String parameters* for details on alternative ways to supply the `text` parameter. Note that if an alternative delimiter is used, it must not be an upper case letter.

See also *#UDGTABLE*.

Version	Changes
3.1.2	New

#INCLUDE

In HTML mode, the `#INCLUDE` macro expands to the contents of one or more ref file sections. In ASM mode, it expands to an empty string.

```
#INCLUDE[paragraphs](pattern)
```

- `paragraphs` specifies how to format the contents of the ref file sections: verbatim (0 - the default), or into paragraphs (1)
- `pattern` is a regular expression pattern that identifies the names of the ref file sections to include

The `#INCLUDE` macro can be used to insert the contents of one ref file section into another. For example:

```
[MemoryMap:RoutinesMap]
Intro=#INCLUDE(RoutinesMapIntro)

[RoutinesMapIntro]
This is the intro to the 'Routines' map page.
```

If `pattern` identifies multiple ref file sections, they are concatenated in the order in which they appear in the ref file.

The `paragraphs` parameter of the `#INCLUDE` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `pattern` parameter.

Version	Changes
8.6	Added the ability to combine multiple ref file sections
8.3	Added support for replacement fields in the `paragraphs` parameter
5.3	New

#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, it defaults either to the title of the entry being linked to (if the page is a *box page* and contains an entry with the specified anchor), or to the page's link text.

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'.

To create a hyperlink to an entry on a memory map page, use the address of the entry as the anchor. For example:

```
; Now we update the #LINK:GameStatusBuffer#40000(number of lives).
```

In HTML mode, the anchor of this `#LINK` macro (40000) is converted to the format specified by the `AddressAnchor` parameter in the *[Game]* section.

Version	Changes
5.4	When linking to an entry on a *box page*, the link text, if left blank, defaults to the title of the entry (in HTML mode)
5.2	An entry address anchor in a link to a memory map page is converted to the format specified by the `AddressAnchor` parameter
3.1.3	If left blank, the link text defaults to the page's link text in HTML mode
2.1	New

#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[,bullet])][<flag>][items]LIST#
```

- `class` is the CSS class to use for the `<ul>` element
- `bullet` is the bullet character to use in ASM mode
- `flag` is the wrap flag (see below)

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

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 for all lists by using a *@set* directive to set the `bullet` property, or it can be changed for a specific list by setting the `bullet` parameter.

The wrap flag (`flag`), if present, determines how *sna2skool.py* will write list items when reading from a control file. Supported values are:

- `nowrap` - write each list item on a single line
- `wrapalign` - wrap each list item with an indent at the start of the second and subsequent lines to maintain text alignment with the first line

By default, each list item is wrapped over multiple lines with no indent.

Version	Changes
7.2	`#LIST` can be used in register descriptions in ASM mode
7.0	Added the `nowrap` and `wrapalign` flags
6.4	In ASM mode: `#LIST` can be used in an instruction-level comment and as a parameter of another macro; if the bullet character is an empty string, list items are no longer indented by one space; added the `bullet` parameter
3.2	New

#N

The `#N` macro renders a numeric value in either decimal or hexadecimal format depending on the options used with *skool2asm.py* or *skool2html.py*. A hexadecimal number is rendered in lower case when the `--lower` option is used, or in upper case otherwise.

```
#Nvalue[,hwidth,dwidth,affix,hex][(prefix[,suffix])]
```

- `value` is the numeric value
- `hwidth` is the minimum number of digits printed in hexadecimal output (default: 2 for values < 256, or 4 otherwise)

- `dwidth` is the minimum number of digits printed in decimal output (default: 1)
- `affix` is 1 if `prefix` or `suffix` is specified, 0 if not (default: 0)
- `hex` is 1 to render the value in hexadecimal format unless the `--decimal` option is used, or 0 to render it in decimal format unless the `--hex` option is used (default: 0)
- `prefix` is the prefix for a hexadecimal number (default: empty string)
- `suffix` is the suffix for a hexadecimal number (default: empty string)

For example:

```
#N15,4,5,1(0x)
```

This instance of the `#N` macro expands to one of the following:

- `00015` (when `--hex` is not used)
- `0x000F` (when `--hex` is used without `--lower`)
- `0x000f` (when both `--hex` and `--lower` are used)

The integer parameters of the `#N` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `prefix` and `suffix` parameters.

Version	Changes
8.3	Added support for replacement fields in the integer parameters
6.2	Added the `hex` parameter
5.2	New

#R

In HTML mode, the `#R` 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), which may contain *replacement fields*
- `code` is the ID of the disassembly that contains the routine or data block (if not given, the current disassembly is assumed; otherwise this must be either an ID defined in an *[OtherCode:*]* section of the ref file, or `main` to identify the main disassembly)
- `#name` is the named anchor of an item on the disassembly page
- `link text` is the link text to use

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

If `link_text` is not provided, it defaults to the label for `addr` if one is defined, or to the address formatted according to the `Address` parameter in the *[Game]* section.

In ASM mode, the `#R` macro expands to the link text if it is specified, or to the label for `addr`, or to `addr` (converted to decimal or hexadecimal as appropriate) 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).

To create a hyperlink to the first instruction in a routine or data block, use an anchor that evaluates to the address of that instruction. For example:

```
; See the #R40000#40000(first item) in the data table at 40000.
```

In HTML mode, the anchor of this `#R` macro (40000) is converted to the format specified by the `AddressAnchor` parameter in the *[Game]* section.

Version	Changes
8.4	In HTML mode, the link text defaults to the address formatted according to the `Address` parameter
8.3	Added support for replacement fields in the `addr` parameter
6.1	In ASM mode, `addr` is converted to decimal or hexadecimal as appropriate even when it refers to an unavailable instruction
5.1	An anchor that matches the entry address is converted to the format specified by the `AddressAnchor` parameter; added support for arithmetic expressions and skool macros in the `addr` parameter
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
2.0	Added support for the `@code` notation

#RAW

The `#RAW` macro expands to the exact value of its sole string argument, leaving any other macros (or macro-like tokens) it contains unexpanded.

```
#RAW(text)
```

For example:

```
; See the routine at #RAW(#BEEF).
```

This instance of the `#RAW` macro expands to '#BEEF'.

See *String parameters* for details on alternative ways to supply the `text` parameter. Note that if an alternative delimiter is used, it must not be an upper case letter.

Version	Changes
6.4	New

#REG

In HTML mode, the `#REG` macro expands to a styled `<span>` element containing a register name or arbitrary text (with case adjusted as appropriate).

```
#REGreg
```

where `reg` is the name of the register, or:

```
#REG(text)
```

where `text` is arbitrary text (e.g. `hlh'l'`).

See *String parameters* for details on alternative ways to supply the `text` parameter. Note that if an alternative delimiter is used, it must not be a letter.

In ASM mode, the `#REG` macro expands to either `reg` or `text` (with case adjusted as appropriate).

The register name (`reg`) must be one of the following:

```
a b c d e f h l
a' b' c' d' e' f' h' l'
af bc de hl
af' bc' de' hl'
ix iy ixh iyh ixl iyl
i r sp pc
```

For example:

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

Version	Changes
5.4	Added support for an arbitrary text parameter
5.3	Added support for the F and F' registers
5.1	The `reg` parameter must be a valid register name

#SIM

The `#SIM` macro simulates the execution of machine code in the internal memory snapshot constructed from the contents of the skool file.

```
#SIMstop[,start,clear,a,f,bc,de,hl,xa,xf,xbc,xde,xhl,ix,iy,i,r,sp]
```

- `stop` is the address at which to stop execution
- `start` is the address at which to start execution (default: `stop` from the previous invocation of the `#SIM` macro, or 0 if this is the first run)
- `clear` is 0 to use the register values that resulted from the previous invocation of the `#SIM` macro (the default), or 1 to reset them to their default values (shown below)
- `a` sets the value of the A (accumulator) register (default: 0)
- `f` sets the value of the F (flags) register (default: 0)
- `bc` sets the value of the BC register pair (default: 0)
- `de` sets the value of the DE register pair (default: 0)
- `hl` sets the value of the HL register pair (default: 0)
- `xa` sets the value of the A' (shadow accumulator) register (default: 0)
- `xf` sets the value of the F' (shadow flags) register (default: 0)
- `xbc` sets the value of the shadow BC register pair (default: 0)
- `xde` sets the value of the shadow DE register pair (default: 0)

- `xhl` sets the value of the shadow HL register pair (default: 0)
- `ix` sets the value of the IX register (default: 0)
- `iy` sets the value of the IY register (default: 23610)
- `i` sets the value of the I register (default: 63)
- `r` sets the value of the R register (default: 0)
- `sp` sets the value of the stack pointer (default: 23552)

The parameters of the `#SIM` macro may contain *replacement fields* and may also be given as keyword arguments.

When execution stops, the simulator's register and clock values are copied to the `sim` dictionary, where they are accessible via replacement fields with the following names:

- `sim[A]`
- `sim[F]`
- `sim[BC]`
- `sim[DE]`
- `sim[HL]`
- `sim[^A]` - the shadow A register
- `sim[^F]` - the shadow flags register
- `sim[^BC]` - the shadow BC register pair
- `sim[^DE]` - the shadow DE register pair
- `sim[^HL]` - the shadow HL register pair
- `sim[IX]`
- `sim[IY]`
- `sim[I]`
- `sim[R]`
- `sim[SP]`
- `sim[PC]` - the program counter (equal to `stop`); this is used as the default value of `start` for the next invocation of the `#SIM` macro
- `sim[tstates]` - the number of T-states elapsed

For example:

```
@assemble=2,2
; #SIM(start=32768,stop=32772,bc=13256,de=672)
 32768 LD HL,443
 32771 ADD HL,BC
; At this point HL=#EVAL({sim[HL]}).
; #SIM(32773)
 32772 ADD HL,DE
; And now HL=#EVAL({sim[HL]}).
 32773 RET
```

The first `#SIM` macro initialises the BC and DE register pairs to 13256 and 672 respectively, starts executing code at 32768, and stops when it reaches the 'ADD HL,DE' instruction at 32772. The second `#SIM` macro picks up execution where the first left off, and stops when it reaches the 'RET' instruction at 32773.

After the #EVAL macros have been expanded, the second mid-block comment here is rendered as 'At this point HL=13699', and the third is rendered as 'And now HL=14371'.

Note: The simulator does not simulate memory contention, I/O contention, or interrupts. This means that sim[tstates] may not be accurate if the code being simulated runs in or accesses contended memory, or performs I/O operations, or runs while interrupts are enabled.

Note that, by default, the internal memory snapshot constructed by *skool2asm.py* is entirely blank (all zeroes), and the snapshot constructed by *skool2html.py* is populated only by DEFB, DEFM, DEFS and DEFW statements, and by *@defb*, *@defs* and *@defw* directives. To make sure that the internal memory snapshot actually contains the code to be executed, use the *@assemble* directive (as shown in the example above).

Note also that code executed by the #SIM macro operates directly on the internal memory snapshot, and therefore can modify it. To avoid that, use the *#PUSHS* and *#POPS* macros to operate on a copy of the snapshot.

Version	Changes
8.7	New

#SPACE

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

```
#SPACE[num]
```

- num is the number of spaces required (default: 1), which may contain *replacement fields*

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
8.3	Added support for replacement fields in the num parameter
5.1	Added support for arithmetic expressions and skool macros in the num parameter
2.4.1	Added support for the #SPACE([num]) syntax

#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]...]]])][<flag>][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.
- `flag` is the wrap flag (see below)

Each row in a table must start with `{` followed by a space, and end with `}` preceded by a space. The cells in a row must be separated by `|` with a space 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 render a cell as a `<th>` element, 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 makes a cell transparent (i.e. gives it the same background colour as the page body).

If a cell requires more than one indicator, separate the indicators 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 wide }
; TABLE#
```

The wrap flag (`flag`), if present, determines how *sna2skool.py* will write table rows when reading from a control file. Supported values are:

- `nowrap` - write each table row on a single line
- `wrapalign` - wrap each table row with an indent at the start of the second and subsequent lines to maintain text alignment with the rightmost column on the first line

By default, each table row is wrapped over multiple lines with no indent.

See also *#UDGTABLE*.

Version	Changes
7.2	`#TABLE` can be used in register descriptions in ASM mode
7.0	Added the `nowrap` and `wrapalign` flags
6.4	In ASM mode, `#TABLE` can be used in an instruction-level comment and as a parameter of another macro

#TSTATES

The `#TSTATES` macro expands to the time taken, in T-states, to execute one or more instructions.

```
#TSTATESstart[,stop,flags(text)]
```

- `start` is the address of the instruction at which to start the clock
- `stop` is the address of the instruction at which to stop the clock
- `flags` controls various options (see below)
- `text` is the text to expand to (when bit 1 of `flags` is set); this may contain the placeholders `$min` and `$max` for the sums of the smaller and larger timing values of the instructions in the given range, or `$tstates` for the actual timing value when bit 2 of `flags` is set

`flags` is the sum of the following values, chosen according to the desired outcome:

- 1 (bit 0) - use the larger timing value for an instruction whose timing is variable
- 2 (bit 1) - expand to `text`
- 4 (bit 2) - execute instructions in a simulator to get the actual timing

For example:

```
c30000 LD A,1   ; This instruction takes #TSTATES30000 T-states
```

This instance of the `#TSTATES` macros expands to '7'.

For any instruction in the range `start` to `stop` whose timing is variable (e.g. a conditional call, return or relative jump), the smaller timing value is used by default:

```
c40000 RET Z    ; This instruction takes at least #TSTATES40000 T-states
```

This instance of the `#TSTATES` macros expands to '5'.

To use the larger timing values, set bit 0 of `flags`. If both smaller and larger timing values are required, set bit 1 of `flags` and use the `text` parameter:

```
c50000 LD B,100    ; Set the delay parameter
 50002 DJNZ 50002  ; Delay for #TSTATES50002,,2(#EVAL(99*$max+$min)) T-states
```

This instance of the `#TSTATES` macro expands to `#EVAL(99*13+8)`, which in turn expands to '1295'.

Note that an instruction's timing can be determined only if it has been assembled. To make sure that it is assembled, use the *@assemble* directive. In addition, unless bit 2 of `flags` is set, only true instructions (i.e. not `DEFB`, `DEFM`, `DEFS` and `DEFW` statements) can be timed.

If bit 2 of `flags` is set, the `stop` address must be specified, and the instructions are executed in a simulator to determine their actual timing. This is useful for computing the time taken by conditional operations and operations that are repeated in a loop. For example:

```
c32768 LD DE,0     ; {This creates a delay of #TSTATES(32768,32776,4)
*32771 DEC DE      ; T-states
 32772 LD A,D      ;
 32773 OR E        ;
 32774 JR NZ,32771 ; }
 32776 RET
```

This instance of the `#TSTATES` macro expands to '1703941'.

Note: The simulator does not simulate memory contention, I/O contention, or interrupts. This means that `#TSTATES` may not provide accurate timing if the code being timed runs in or accesses contended memory, or performs I/O operations, or runs while interrupts are enabled.

The integer parameters of the `#TSTATES` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `text` parameter.

See also the *#SIM* macro, which can not only time instructions, but also track changes to register values as code is executed.

See also the `Timings` configuration parameter for *sna2skool.py*, which can be used to show instruction timings in comment fields when disassembling a snapshot.

Version	Changes
8.7	New

#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 will be rendered in HTML mode only.

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

#VERSION

The `#VERSION` macro expands to the version of SkoolKit.

```
#VERSION
```

Version	Changes
6.0	New

8.4.7 Image macros

The *#COPY*, *#FONT*, *#OVER*, *#PLOT*, *#SCR*, *#UDG*, *#UDGARRAY* and *#UDGS* macros (described in the following sections) may be used to create images based on graphic data in the memory snapshot. They are not supported in ASM mode.

Some of these macros have several numeric parameters, most of which are optional. This can give rise to a long sequence of commas in a macro parameter string, making it hard to read (and write); for example:

```
#UDG32768,,,,,,1
```

To alleviate this problem, the image macros accept keyword arguments at any position in the parameter string; the `#UDG` macro above could be rewritten as follows:

```
#UDG32768,rotate=1
```

#COPY

In HTML mode, the `#COPY` macro copies all or part of an existing frame into a new frame.

```
#COPY[x,y,width,height,scale,mask,tindex,alpha][{CROP}](old,new)
```

- `x` and `y` are the coordinates of the top left tile of the existing frame to include in the new frame (default: (0, 0))
- `width` and `height` are the width and height (in tiles) of the portion of the existing frame to copy (by default, the portion extends to the right and bottom edges of the existing frame)
- `scale` is the scale of the new frame; if omitted, the scale of the existing frame is used
- `mask` is the mask type of the new frame (see *Masks*); if omitted, the mask type of the existing frame is used
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour for the new frame (see *Palette*); if omitted, the transparency index of the existing frame is used
- `alpha` is the alpha value (0-255) to use for the transparent colour in the new frame; if omitted, the alpha value of the existing frame is used
- `CROP` is the cropping specification for the new frame (see *Cropping*); if omitted, the cropping specification of the existing frame is used

- `old` is the name of the existing frame
- `new` is the name of the new frame

For example:

```
; #UDGARRAY4(30000-30120-8)(*original)
; #COPY1,1,2,2(original,centre)
; #UDGARRAY*centre(img)
```

This instance of the `#COPY` macro creates a new frame from a copy of the central 2x2 portion of the 4x4 frame created by the `#UDGARRAY` macro. The `#UDGARRAY*` macro then creates an image of the new frame.

The integer parameters and the cropping specification of the `#COPY` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `old` and `new` parameters.

Version	Changes
8.6	Added the `tindex` and `alpha` parameters
8.5	New

#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,tindex,alpha][{CROP}][(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 scale of the image (default: 2)
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour (default: 0; see *Palette*)
- `alpha` is the alpha value (0-255) to use for the transparent colour (default: the value of the `PNGAlpha` parameter in the *[ImageWriter]* section)
- `CROP` is the cropping specification (see *Cropping*)
- `fname` is the name of the image file (see *Filenames*; default: '*font*')

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 8x8 font whose graphic data starts at 49152.

The integer parameters and the cropping specification of the `#FONT` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `text` parameter.

Version	Changes
8.3	Added support for replacement fields in the integer parameters and the cropping specification
8.2	Added the `tindex` and `alpha` parameters
6.3	Added support for image path ID replacement fields in the `fname` parameter
5.1	Added support for arithmetic expressions and skool macros in the numeric parameters
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the `<img>` element
4.0	Added support for keyword arguments
3.6	Added the `text` parameter, and made the `chars` parameter optional
3.0	Added image-cropping capabilities
2.0.5	Added the `fname` parameter and support for regular 8x8 fonts

#OVER

In HTML mode, the `#OVER` macro superimposes one frame (the foreground frame) on another (the background frame).

```
#OVERx,y[,xoffset,yoffset,rmode][(attr)][(byte)](bg,fg)
```

- `x` and `y` are the tile coordinates on the background frame at which to superimpose the foreground frame; negative coordinates are allowed
- `xoffset` and `yoffset` are the pixel offsets by which to shift the foreground frame from the given tile coordinates (default: (0, 0))
- `rmode` is the attribute and graphic byte replacement mode (see below)
- `attr` is the replacement attribute byte for any background UDG over which a foreground UDG is superimposed (when `rmode` is 1 or 3)
- `byte` is the replacement graphic byte for any background UDG over which a foreground UDG is superimposed (when `rmode` is 2 or 3)
- `bg` is the name of the background frame
- `fg` is the name of the foreground frame

`rmode` specifies whether and how to replace the attribute and graphic bytes of each background UDG over which a foreground UDG is superimposed:

- 0 - leave the attribute byte unchanged and apply the foreground frame's mask
- 1 - replace the attribute byte with the value of `attr` and apply the foreground frame's mask
- 2 - leave the attribute byte unchanged and replace the graphic bytes with the value of `byte`
- 3 - replace the attribute byte with the value of `attr` and replace the graphic bytes with the value of `byte`

`attr` is an expression that is evaluated once for each background UDG over which a foreground UDG is superimposed. `attr` may contain skool macros, and recognises the following placeholders:

- `$b` - the background UDG attribute value
- `$f` - the foreground UDG attribute value

`byte` is an expression that is evaluated once for each of the 8 graphic bytes in a background UDG over which a foreground UDG is superimposed. `byte` may contain skool macros, and recognises the following placeholders:

- `$b` - the background UDG graphic byte value
- `$f` - the foreground UDG graphic byte value

- `$m` - the foreground UDG mask byte value (or 0 if the foreground UDG has no mask)

If the foreground frame has no mask, its contents are combined with those of the background frame by OR operations.

For example:

```
; #UDGARRAY2(30000-30024-8)(*background)
; #UDG30032:30040(*object)
; #OVER0,1(background,object)
; #UDGARRAY*background(image)
```

This instance of the `#OVER` macro superimposes the frame created by the `#UDG` macro at tile coordinates (0, 1) on the background frame created by the `#UDGARRAY` macro. The `#UDGARRAY*` macro then creates an image of the modified background frame.

The integer parameters of the `#OVER` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `bg` and `fg` parameters.

Version	Changes
8.5	New

#PLOT

In HTML mode, the `#PLOT` macro sets, resets or flips a pixel in a frame already created by one of the other image macros.

```
#PLOTx,y[,value](frame)
```

- `x` and `y` are the coordinates of the pixel, relative to the top-left corner of the frame
- `value` is 0 to reset the pixel, 1 to set it (the default), or 2 to flip it
- `frame` is the name of the frame

For example:

```
; #UDG30000(*tile)
; #PLOT1,2(tile)
; #UDGARRAY*tile(tile)
```

This instance of the `#PLOT` macro sets the second pixel from the left in the third row from the top in the frame created by the `#UDG` macro. The `#UDGARRAY*` macro then creates an image of the modified frame.

The integer parameters of the `#PLOT` macro may contain *replacement fields*.

Version	Changes
8.3	New

#SCR

In HTML mode, the `#SCR` 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,df,af,tindex,alpha][{CROP}][(fname)]
```

- `scale` is the 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)
- `df` is the base address of the display file (default: 16384)
- `af` is the base address of the attribute file (default: 22528)
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour (default: 0; see *Palette*)
- `alpha` is the alpha value (0-255) to use for the transparent colour (default: the value of the `PNGAlpha` parameter in the *[ImageWriter]* section)
- `CROP` is the cropping specification (see *Cropping*)
- `fname` is the name of the image file (see *Filenames*; default: '*scr*')

For example:

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

The integer parameters and the cropping specification of the `#SCR` macro may contain *replacement fields*.

Version	Changes
8.3	Added support for replacement fields in the integer parameters and the cropping specification
8.2	Added the `tindex` and `alpha` parameters
6.3	Added support for image path ID replacement fields in the `fname` parameter
5.1	Added support for arithmetic expressions and skool macros in the numeric parameters
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the `<img>` element
4.0	Added support for keyword arguments
3.0	Added image-cropping capabilities and the `df` and `af` parameters
2.0.5	Added the `scale`, `x`, `y`, `w`, `h` and `fname` parameters

#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,mask,tindex,alpha][:MASK][{CROP}][(fname)]
```

- `addr` is the base address of the UDG bytes
- `attr` is the attribute byte to use (default: 56)
- `scale` is the scale of the image (default: 4)
- `step` is the interval between successive bytes of the UDG (default: 1)
- `inc` is 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)
- `mask` is the type of mask to apply (see *Masks*)
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour (default: 0; see *Palette*)
- `alpha` is the alpha value (0-255) to use for the transparent colour (default: the value of the `PNGAlpha` parameter in the *[ImageWriter]* section)
- `MASK` is the mask specification (see below)
- `CROP` is the cropping specification (see *Cropping*)
- `fname` is the name of the image file (see *Filenames*); if not given, a name specified by the `UDGFilename` parameter in the *[Paths]* section will be used

The mask specification (`MASK`) takes the form:

```
addr[,step]
```

- `addr` is the base address of the mask bytes to use for the UDG
- `step` is the interval between successive mask bytes (defaults to the value of `step` for the UDG)

Note that if any of the parameters in the mask specification is expressed using arithmetic operations or skool macros, then the entire specification must be enclosed in parentheses.

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*), with attribute byte 6 (INK 6: PAPER 0).

The integer parameters, mask specification and cropping specification of the `#UDG` macro may contain *replacement fields*.

Version	Changes
8.3	Added support for replacement fields in the integer parameters, mask specification and cropping specification
8.2	Added the `tindex` and `alpha` parameters
6.3	Added support for image path ID replacement fields in the `fname` parameter
5.1	Added support for arithmetic expressions and skool macros in the numeric parameters
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the `<img>` element
4.0	Added the `mask` parameter and support for AND-OR masking; added support for keyword arguments
3.1.2	Made the `attr` parameter optional
3.0	Added image-cropping capabilities
2.4	Added the `rotate` parameter
2.3.1	Added the `flip` parameter
2.1	Added support for masks
2.0.5	Added the `fname` parameter

#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,mask,tindex,alpha](SPEC1[;SPEC2;...
↪])[@ATTRS1[;ATTRS2;...]][{CROP}](fname)
```

- `width` is the width of the image (in UDGs)
- `attr` is the default attribute byte of each UDG (default: 56)
- `scale` is the 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)
- `mask` is the type of mask to apply (see *Masks*)
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour (default: 0; see *Palette*)
- `alpha` is the alpha value (0-255) to use for the transparent colour (default: the value of the `PNGAlpha` parameter in the *[ImageWriter]* section)
- `CROP` is the cropping specification (see *Cropping*)
- `fname` is the name of the image file (see *Filenames*)

`SPEC1`, `SPEC2` etc. are UDG specifications for the sets of UDGs that make up the array. The parentheses around them are optional, but recommended. If the parentheses are omitted, `SPEC1` must be prefixed by a semicolon to separate it from the main parameters. Each UDG specification has the form:

```
addr[,attr,step,inc][:MASK]
```

- addr is the address range specification for the set of UDGs (see below)
- attr is the attribute byte of each UDG in the set (defaults to the value of attr for the UDG array)
- step is the interval between successive bytes of each UDG in the set (defaults to the value of step for the UDG array)
- inc is added to each byte of every UDG in the set before constructing the image (defaults to the value of inc for the UDG array)
- MASK is the mask specification

The mask specification (MASK) takes the form:

```
addr[,step]
```

- addr is the address range specification for the set of mask UDGs (see below)
- step is the interval between successive bytes of each mask UDG in the set (defaults to the value of step for the set of UDGs)

ATTRS1, ATTRS2 etc. are attribute address range specifications (see below). If supplied, attribute values are taken from the specified addresses instead of the attr parameter values.

Address range specifications (for both UDGs and attributes) 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 items in each row as 1, and the step between the base addresses of adjacent items 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.

Note that, like the main parameters of a #UDGARRAY macro (up to but not including the first semicolon), if any of the following parts of the parameter string is expressed using arithmetic operations or skool macros, then that part must be enclosed in parentheses:

- any of the 1-5 parts of a UDG, mask or attribute address range specification (separated by - and x)
- the part of a UDG or mask specification after the comma that follows the address range

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*.

The integer parameters, UDG specifications, attribute address range specification and cropping specification of the `#UDGARRAY` macro may contain *replacement fields*.

See also *#UDGS*.

Version	Changes
8.6	The UDG specifications may be enclosed in parentheses
8.3	Added support for replacement fields in the integer parameters and the UDG, attribute address range and cropping specifications
8.2	Added the `tindex` and `alpha` parameters
7.1	Added the ability to specify attribute addresses
6.3	Added support for image path ID replacement fields in the `fname` parameter
5.1	Added support for arithmetic expressions and skool macros in the numeric parameters
4.2	Added the ability to specify alt text for the `<img>` element
4.0	Added the `mask` parameter and support for AND-OR masking; added support for keyword arguments
3.6	Added support for creating an animated image from an arbitrary sequence of frames
3.1.1	Added support for UDG address ranges with horizontal and vertical steps
3.0	Added image-cropping capabilities
2.4	Added the `rotate` parameter
2.3.1	Added the `flip` parameter
2.2.5	Added support for masks
2.0.5	New

#UDGS

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

```
#UDGSwidth,height[,scale,flip,rotate,mask,tindex,alpha][{CROP}](fname)(uframe)
```

- `width` is the width of the array
- `height` is the height of the array
- `scale` is the scale of the image
- `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)
- `mask` is the type of mask to apply (see *Masks*)
- `tindex` is the index (0-15) of the entry in the palette to use as the transparent colour (see *Palette*)
- `alpha` is the alpha value (0-255) to use for the transparent colour
- `CROP` is the cropping specification (see *Cropping*)
- `fname` is the name of the image file (see *Filenames*)
- `uframe` is expanded once for each slot in the array, and may contain the placeholders `$x` and `$y` for the coordinates of the slot; the resulting text is interpreted as the name of the frame whose top-left UDG should be placed into the slot

For example:

```
#UDGS2,2(sprite)(
  #LET(a=30000+9*(2*$y+$x))
  #UDG({a}+1,#PEEK({a}))(*udg)
  udg
)
```

This instance of the `#UDGS` macro expands to an `<img>` element for the image of the 2x2 sprite formed by the four 9-byte UDG definitions (where the first byte is the attribute value, and the next eight bytes are the graphic data) with base addresses 30000, 30009, 30018 and 30027. The image file is named *sprite.png*.

Unless specified by macro arguments, the scale, mask type, transparency index and alpha value of the image created by the `#UDGS` macro are copied from the last frame used to populate the array (corresponding to the bottom-right UDG).

The integer parameters and the `uframe` parameter of the `#UDGS` macro may contain *replacement fields*.

See *String parameters* for details on alternative ways to supply the `uframe` parameter.

See also *#UDGARRAY*.

Version	Changes
8.6	New

Filenames

The `fname` parameter of the *#FONT*, *#SCR*, *#UDG*, *#UDGARRAY* and *#UDGS* macros can be used to specify not only an image filename, but also its exact location, the `alt` attribute of the `<img>` element, and a frame name (see *Animation*).

If `fname` contains an image path ID replacement field (e.g. `{ScreenshotImagePath}/udgs`), the corresponding parameter value from the *[Paths]* section will be substituted.

If `fname` starts with a '/', the filename is taken to be relative to the root of the HTML disassembly.

If `fname` contains no image path ID replacement fields and does not start with a '/', the filename is taken to be relative to the directory defined by one of the following parameters in the *[Paths]* section, depending on the macro being used:

- `FontImagePath` - *#FONT*
- `ScreenshotImagePath` - *#SCR*
- `UdgImagePath` - *#UDG*, *#UDGARRAY* and *#UDGS*

If `fname` does not end with '*.png*', that suffix will be appended.

If an image with the given filename doesn't already exist, it will be created.

The value of the `alt` attribute in the `<img>` element can be specified by appending a `|` character and the required text to the filename. For example:

```
#SCR(screenshot1|Screenshot 1)
```

This `#SCR` macro creates an image named *screenshot1.png* with alt text 'Screenshot 1'.

Animation

The image macros may be used to create the frames of an animated image. 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 creates the animated image from a set of frames:

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

`FRAME1`, `FRAME2` etc. are frame specifications. The parentheses around them are optional, but recommended. Each frame specification has the form:

```
name[,delay,x,y]
```

- `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)
- `x` and `y` are the coordinates at which to render the frame, relative to the top-left corner of the first frame (default: (0,0))

For example:

```
; #UDGTABLE {
; #FONT:(hello)$3D00(hello*) |
; #FONT:(there)$3D00(there*) |
; #FONT:(peeps)$3D00(peeps*) |
; #UDGARRAY*(hello,50;there;peeps)(hello_there_peeps)
; } TABLE#
```

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

The integer parameters of a frame specification may contain *replacement fields*.

Note that the first frame of an animated image determines the size of the image as a whole. Therefore, the region defined by the width, height and coordinates of any subsequent frame must fall entirely inside the first frame.

Version	Changes
8.6	The frame specifications may be enclosed in parentheses
8.3	Added the `x` and `y` parameters to the frame specification; added support for replacement fields in the integer parameters of a frame specification
3.6	New

Cropping

The *#COPY*, *#FONT*, *#SCR*, *#UDG*, *#UDGARRAY* and *#UDGS* macros accept a cropping specification (`CROP`) which takes the form:

```
x,y,width,height
```

- `x` is the x-coordinate of the leftmost pixel column of the constructed image to include in the final image (default: 0); 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 (default: 0); if greater than 0, the image will be cropped on the top
- `width` is the width of the final image in pixels (default: width of the constructed image)
- `height` is the height of the final image in pixels (default: height of the constructed image)

For example:

```
#UDG40000,scale=2{2,2,12,12}
```

This `#UDG` macro creates an image of the UDG at 40000, at scale 2, with the top two rows and bottom two rows of pixels removed, and the leftmost two columns and rightmost two columns of pixels removed.

The parameters of the cropping specification may contain *replacement fields*.

Masks

The *#COPY*, *#UDG*, *#UDGARRAY* and *#UDGS* macros accept a `mask` parameter that determines what kind of mask to apply to each UDG. The supported values are:

- 0 - no mask
- 1 - OR-AND mask (this is the default)
- 2 - AND-OR mask

Given a 'background' bit (B), a UDG bit (U), and a mask bit (M), the OR-AND mask works as follows:

- OR the UDG bit (U) onto the background bit (B)
- AND the mask bit (M) onto the result

U	M	Result
0	0	0 (paper)
0	1	B (transparent)
1	0	0 (paper)
1	1	1 (ink)

The AND-OR mask works as follows:

- AND the mask bit (M) onto the background bit (B)
- OR the UDG bit (U) onto the result

U	M	Result
0	0	0 (paper)
0	1	B (transparent)
1	0	1 (ink)
1	1	1 (ink)

By default, transparent bits in masked images are rendered in bright green (#00fe00); this colour can be changed by modifying the `TRANSPARENT` parameter in the *[Colours]* section. To make the transparent bits in masked images actually transparent, set `PNGAlpha=0` in the *[ImageWriter]* section.

Palette

Images created by the image macros use colours drawn from a palette of 16 entries:

- 0 - transparent
- 1 - black
- 2 - blue
- 3 - red
- 4 - magenta
- 5 - green
- 6 - cyan
- 7 - yellow
- 8 - white
- 9 - bright blue
- 10 - bright red
- 11 - bright magenta
- 12 - bright green
- 13 - bright cyan
- 14 - bright yellow
- 15 - bright white

The RGB values for these colours are defined in the *[Colours]* section.

The index values (0-15) may be used by an image macro's `tindex` parameter to specify a transparent colour to use other than the default (0). The palette entry specified by `tindex`, if not 0, will be used as the transparent colour only if the image does not already contain any transparent bits produced by a *mask*. In an animated image, the `tindex` and `alpha` values on the first frame take effect; any `tindex` and `alpha` values on the second or subsequent frames are ignored.

For example:

```
#UDG30000,attr=2,tindex=1,alpha=0
```

This `#UDG` macro creates an image of the UDG at 30000 with red INK and black PAPER (`attr=2`), black as the transparent colour (`tindex=1`), and full transparency (`alpha=0`).

8.4.8 Snapshot macros

The *#POKES*, *#POPS* and *#PUSHS* macros (described in the following sections) may be used to manipulate the memory snapshot that is built from the skool file. Each macro expands to an empty string.

#POKES

The `#POKES` 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 parameter string of the `#POKES` macro may contain *replacement fields*.

See also *#PEEK*.

Version	Changes
8.3	Added support for replacement fields in the parameter string
5.1	Added support for arithmetic expressions and skool macros in the parameter string
3.1	Added support for ASM mode
2.3.1	Added support for multiple addresses

#POPS

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

```
#POPS
```

Version	Changes
3.1	Added support for ASM mode

#PUSHS

The #PUSHS macro saves the current internal memory 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. The name can be retrieved by using the *get_snapshot_name()* method on HtmlWriter.

Version	Changes
3.1	Added support for ASM mode

8.4.9 Defining macros with #DEF

By using the *#DEF* macro, it is possible to define new macros based on existing ones without writing any Python code. Some examples are given below.

#ASM

There is the *#HTML* macro for inserting content in HTML mode only, but there is no corresponding macro for inserting content in ASM mode only. The following #DEF macro defines an #ASM macro to fill that gap:

```
#DEF(#ASM #IF({mode[asm]}))
```

For example:

```
#ASM(This text appears only in ASM mode.)
```

#ASMUDG

The *#UDG* macro is not supported in ASM mode, but #DEF can define an #ASMUDG macro (based on the #ASM macro defined above) that is:

```
#DEF(#ASMUDG(a) #ASM(#LIST(,) #FOR($a,$a+7)(u,{ |#FOR(7,0,-1)(n,#IF(#PEEKu&2**n)(*,␣
↪))| }) LIST#))
```

For example:

```
; #ASMUDG30000
 30000 DEFB 48,72,136,144,104,4,10,4
```

If conversion of DEFB statements has been switched on in ASM mode by the *@assemble* directive (e.g. @assemble=,1), this #ASMUDG macro produces the following output:

```
; |  **    |
; | *  *   |
; |*   *   |
; |*  *    |
; | ** *   |
```

(continues on next page)

(continued from previous page)

```
; |      *   |
; |    * * |
; |      *   |
```

#TILE, #TILES

Suppose the game you're disassembling arranges tiles in groups of nine bytes: the attribute byte first, followed by the eight graphic bytes. If there is a tile at 32768, then:

```
#UDG(32769,#PEEK32768)
```

will create an image of it. If you want to create several tile images, this syntax can get cumbersome; it would be easier if you could supply just the address of the attribute byte. The following `#DEF` macro defines a `#TILE` macro that creates a tile image given an attribute byte address:

```
#DEF(#TILE(a) #UDG($a+1,#PEEK$a))
```

Now you can create an image of the tile at 32768 like this:

```
#TILE32768
```

If you have several nine-byte tiles arranged one after the other, you might want to create images of all of them in a single row of a `#UDGTABLE`. The following `#DEF` macro defines a `#TILES` macro (based on the `#TILE` macro already defined) for this purpose:

```
#DEF(#TILES(a,m) #FOR($a,$a+9*($m-1),9)(n,#TILEn, | ))
```

Now you can create a `#UDGTABLE` of images of a series of 10 tiles starting at 32768 like this:

```
#UDGTABLE { #TILES32768,10 } TABLE#
```

8.5 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.

8.5.1 [AudioWriter]

The `AudioWriter` section contains configuration parameters that control audio file creation by the *#AUDIO* macro. The parameters are in the format:

```
name=value
```

Recognised parameters are:

- `ClockSpeed` - Z80 clock speed in cycles per second (default: 3500000)
- `ContentionBegin` - when memory contention begins, in T-states from the start of a frame (default: 14334)
- `ContentionEnd` - when memory contention ends, in T-states from the start of a frame (default: 57248)
- `ContentionFactor` - percentage slowdown when memory contention is in effect (default: 51)
- `FrameDuration` - length of a frame in T-states (default: 69888)
- `InterruptDelay` - delay in T-states caused by an interrupt routine (default: 942)
- `SampleRate` - sample rate in Hz (default: 44100)

Version	Changes
8.8	Removed the `MaxAmplitude` parameter
8.7	New

8.5.2 [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
3.4	Added support for hexadecimal RGB triplets
2.0.5	New

8.5.3 [Config]

The Config section contains configuration parameters in the format:

```
name=value
```

Recognised parameters are:

- Expand - arbitrary text (intended to consist of one or more *SMPL macros*) that is expanded before any skool macros are expanded elsewhere in the ref file or in skool file annotations; this is similar to the *@expand* directive, but can be used to expand macros that are required only in HTML mode
- GameDir - the root directory of the game's HTML disassembly; if not specified, the base name of the skool 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
- InitModule - the name of a Python module to import before the HTML writer class is imported; the module's location may be specified in the same way as for HtmlWriterClass (see above)
- RefFiles - a semicolon-separated list of extra ref files to use (after any that are automatically read by virtue of having the same filename prefix as the skool file, and before any others named on the *skool2html.py* command line)

The Expand parameter can be used along with the *#INCLUDE* macro to place all the text to be expanded in its own section. This is particularly useful when that text cannot easily be placed on a single line. For example:

```
[Config]
Expand=#INCLUDE(Expand)

[Expand]
; Multiple lines of text to be expanded here
...
```

For information on using the `HtmlWriterClass` parameter and creating your own Python class for writing an HTML disassembly, see the documentation on *extending SkoolKit*.

The `InitModule` parameter can be used to modify the built-in `HtmlWriter` class before it is imported by *skool2html.py*. This is an alternative to creating your own `HtmlWriter` class that is suitable when the required modification is small.

Note that the `Config` section must appear in a ref file that is read automatically by *skool2html.py* by virtue of having the same filename root as the skool file given on the command line (i.e. *game*.ref* if the skool file is *game.skool*).

Version	Changes
8.6	Added the `Expand` parameter
8.5	Added the `InitModule` parameter
5.0	Added the `RefFiles` parameter
3.3.1	Added support to the `HtmlWriterClass` parameter for specifying a module outside the module search path
2.2.3	Added the `HtmlWriterClass` parameter
2.0	New

8.5.4 [EntryGroups]

The `EntryGroups` section defines groups of entries (routines and data blocks) whose disassembly pages can then be given custom titles and headers via the *[Titles]* and *[PageHeaders]* sections. Each line in this section has the form:

```
name=ADDR1[,ADDR2...]
```

where:

- `name` is the entry group name
- `ADDR1`, `ADDR2` etc. are address specifications that identify the entries in the group

An address specification can be either a single address (e.g. `30000`), or an address range (e.g. `30000-30010`).

For example:

```
SpriteVariables=32768,32770-32772
```

This defines an entry group named 'SpriteVariables' that consists of the entries at 32768 and any others between addresses 32770 and 32772 inclusive. The titles and headers of the disassembly pages for these entries can then be specified like this:

```
[Titles]
Asm-SpriteVariables=Sprite variable at {entry[address]}

[PageHeaders]
Asm-SpriteVariables=Sprite variables
```

Entry group names may also be used in the `Includes` parameter of a *[MemoryMap:*]* section. For example:

```
[MemoryMap:SpriteVariables]
Includes=SpriteVariables
```

This defines a memory map page named 'SpriteVariables' consisting of only the entries that belong to the group of the same name.

Version	Changes
8.5	Added support for identifying entries by address ranges
8.4	New

8.5.5 [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:

- `Address` - the format of the address fields on disassembly pages and memory map pages, and of the default link text for the *#R* macro when the target address has no label (default: ''); this format string recognises the replacement field `address`; if the format string is blank, the address is formatted exactly as it appears in the skool file (without any `$` prefix)
- `AddressAnchor` - the format of the anchors attached to instructions on disassembly pages and entries on memory map pages (default: `{address}`)
- `AsmSinglePage` - `1` to write the disassembly on a single page, or `0` to write a separate page for each routine and data block (default: `0`)
- `AudioFormats` - a comma separated list of audio file formats that the *#AUDIO* macro looks for before creating a WAV file (default: `flac,mp3,ogg`)
- `Bytes` - the format specification for the `bytes` attribute of instruction objects in the *asm* and *asm_single_page* templates (default: ''); if not blank, assembled instruction byte values are displayed on disassembly pages
- `Copyright` - the copyright message that appears in the footer of every page (default: '')
- `Created` - the message indicating the software used to create the disassembly that appears in the footer of every page (default: 'Created using SkoolKit #VERSION.')
- `DisassemblyTableNumCols` - the number of columns in the disassembly table on disassembly pages (default: 5); this value is used by the *asm* and *asm_single_page* templates
- `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
- `InputRegisterTableHeader` - the text displayed in the header of input register tables on routine disassembly pages (default: 'Input')
- `JavaScript` - the base name of the JavaScript file to include in every page (default: None); multiple JavaScript files can be declared by separating their names with semicolons

- `Length` - the format of the `length` attribute of entry objects in *HTML templates*, which is used in the Length column on *memory map pages* (default: `{size}`); this format string recognises the replacement field `size`, equal to the size of the entry in bytes
- `LinkInternalOperands` - `1` to hyperlink instruction operands that refer to an address in the same entry as the instruction, or `0` to leave them unlinked (default: `0`)
- `LinkOperands` - a comma-separated list of instruction types whose operands will 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 displayed in the header of output register tables on routine disassembly pages (default: 'Output')
- `Release` - the message indicating the release name and version number of the disassembly that appears in the footer of every page (default: '')
- `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

Every parameter in this section may contain *skool macros*.

The `AddressAnchor` parameter contains a standard Python format string that specifies the format of the anchors attached to instructions on disassembly pages and entries on memory map pages. The default format string is `{address}`, which produces decimal addresses (e.g. `#65280`). To produce 4-digit, lower case hexadecimal addresses instead (e.g. `#ff00`), change `AddressAnchor` to `{address:04x}`. Or to produce 4-digit, upper case hexadecimal addresses if the `--hex` option is used with *skool2html.py*, and decimal addresses otherwise: `{address#IF({mode[base]}==16)(:04X)}`.

Version	Changes
8.7	Added the `AudioFormats` parameter
8.4	Added the `Address` and `Length` parameters
8.0	Added the `AsmSinglePage` parameter
7.2	Added the `Bytes` and `DisassemblyTableNumCols` parameters
6.0	Every parameter (not just `Logo`) may contain *skool macros*
4.3	Added the `AddressAnchor` parameter
4.1	Added the `LinkInternalOperands` parameter
4.0	Set default values for the `InputRegisterTableHeader` and `OutputRegisterTableHeader` parameters; added the `Copyright`, `Created` and `Release` parameters (which used to live in the `[Info]` section in SkoolKit 3)
3.7	Added the `JavaScript` 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`)
3.4	Added the `LinkOperands` parameter
3.1.2	Added the `InputRegisterTableHeader` and `OutputRegisterTableHeader` parameters
2.0.5	Added the `Logo` parameter

8.5.6 [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:

- `PNGAlpha` - the default alpha value (0-255) to use for the transparent colour in a PNG image, where 0 means fully transparent, and 255 means fully opaque (default: `255`)
- `PNGCompressionLevel` - the compression level (0-9) to use for PNG image data, 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`)

Version	Changes
3.0.1	Added the `PNGAlpha` and `PNGEnableAnimation` parameters
3.0	New

8.5.7 [Index]

The `Index` section contains a list of link group IDs in the order in which the link groups will appear on the disassembly index page. The link groups themselves - with the exception of `OtherCode` - are defined in *[Index:*:*]* sections. `OtherCode` is a special built-in link group that contains links to the index pages of secondary disassemblies defined by *[OtherCode:*]* sections.

To see the default `Index` section, run the following command:

```
$ skool2html.py -r Index$
```

Version	Changes
2.0.5	New

8.5.8 [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

To see the default link groups and their contents, run the following command:

```
$ skool2html.py -r Index:
```

Version	Changes
2.0.5	New

8.5.9 [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:

```
PageID=text
```

where:

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

Recognised page IDs are:

- `AsmSinglePage` - the disassembly page (when writing a single-page disassembly)
- `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
- `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 header defined in the *[PageHeaders]* section, except where indicated above.

The link text for a page defined by a *[MemoryMap:*]*, *[OtherCode:*]* or *[Page:*]* section also defaults to the page header text, but can be overridden in this section.

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
5.3	Added the `AsmSinglePage` page ID
2.5	Added the `UnusedMap` page ID
2.2.5	Added the `Changelog` page ID
2.0.5	New

8.5.10 [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.

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

```
name=value
```

Recognised parameters and their default values are:

- `EntryDescriptions` - `1` to display entry descriptions, or `0` not to (default: `0`)
- `EntryTypes` - the types of entries to show in the map (by default, no types are shown); entry types are identified as follows:
  - `b` - DEFB blocks
  - `c` - routines
  - `g` - game status buffer entries
  - `s` - blocks containing bytes that are all the same value
  - `t` - messages
  - `u` - unused addresses
  - `w` - DEFW blocks
- `Includes` - a comma-separated list of entries to include on the memory map page in addition to those specified by the `EntryTypes` parameter; each item in the list may be a single address, an address range (e.g. `30000-30010`), or the name of an entry group defined in the *[EntryGroups]* section
- `Intro` - the text (which may contain HTML markup) displayed at the top of the memory map page (default: '')
- `LabelColumn` - `1` to display the 'Label' column if any entries have ASM labels defined, or `0` not to (default: `0`)
- `LengthColumn` - `1` to display the 'Length' column, or `0` not to (default: `0`); see also the `Length` parameter in the *[Game]* section
- `PageByteColumns` - `1` to display 'Page' and 'Byte' columns, or `0` not to (default: `0`)
- `Write` - `1` to write the memory map page, or `0` not to (default: `1`)

Every parameter in this section may contain *skool macros*.

To see the default memory map pages and their properties, run the following command:

```
$ skool2html.py -r MemoryMap
```

A custom memory map page can be defined by creating a `MemoryMap:*` section for it. By default, the page will be written to *maps/PageID.html*; to change this, add a line to the *[Paths]* section. The title, page header and link text for the custom memory map page can be defined in the *[Titles]*, *[PageHeaders]* and *[Links]* sections.

Every memory map page is built using the *HTML template* whose name matches the page ID, if one exists; otherwise, the stock *Layout* template is used.

Version	Changes
8.5	Added support for address ranges in the `Includes` parameter
8.4	The `EntryTypes` parameter defaults to an empty string
8.1	Added the `LabelColumn` parameter
6.2	Added the `Includes` parameter
6.0	Every parameter (not just `Intro`) may contain *skool macros*
4.0	Added the `EntryDescriptions` and `LengthColumn` parameters
2.5	New

8.5.11 [OtherCode:*]

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

```
[OtherCode:CodeID]
```

where `CodeID` 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.

An `OtherCode:*` section may either be empty or contain a single parameter named `Source` in the form:

```
Source=fname
```

where `fname` is the path to the skool file from which to generate the secondary disassembly. If the `Source` parameter is not provided, its value defaults to *CodeID.skool*.

When a secondary disassembly named `CodeID` is defined, the following page and directory IDs become available for use in the *[Paths]*, *[Titles]*, *[PageHeaders]* and *[Links]* sections:

- `CodeID-Index` - the ID of the index page
- `CodeID-Asm-*` - the IDs of the disassembly pages (`*` is one of `bcgstuw`, depending on the entry type)
- `CodeID-CodePath` - the ID of the directory in which the disassembly pages are written
- `CodeID-AsmSinglePage` - the ID of the disassembly page (when writing a single-page disassembly)

By default, the index page is written to *CodeID/CodeID.html*, and the disassembly pages are written in a directory named *CodeID*; if a single-page template is used, the disassembly page is written to *CodeID/asm.html*.

Note that the index page is a memory map page, and as such can be configured by creating a *[MemoryMap:*]* section (`MemoryMap:CodeID-Index`) for it.

Version	Changes
5.0	Made the `Source` parameter optional
2.0	New

8.5.12 [Page:*]

A `Page:*` section either declares a page that already exists, or defines a custom page in the HTML disassembly. The section name takes 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.

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

```
name=value
```

Recognised parameters are:

- `Content` - the path (directory and filename) of a page that already exists; when this parameter is supplied, no others are required
- `JavaScript` - the base name of the JavaScript file to use in addition to any declared by the `JavaScript` parameter in the *[Game]* section (default: None); multiple JavaScript files can be declared by separating their names with semicolons
- `PageContent` - the HTML source of the body of the page; the *#INCLUDE* macro may be used here to include the contents of a separate ref file section
- `SectionPrefix` - the prefix of the names of the ref file sections from which to build the entries on a *box page*
- `SectionType` - how to parse and render *box page* entry sections (when `SectionPrefix` is defined): as single-line list items with indentation (`ListItems`), as multi-line list items prefixed by '-' (`BulletPoints`), or as paragraphs (the default)

Every parameter in this section may contain *skool macros*.

Note that the `Content`, `SectionPrefix` and `PageContent` parameters are mutually exclusive (and that is their order of precedence); one of them must be present.

By default, the custom page is written to a file named *PageID.html* in the root directory of the disassembly; to change this, add a line to the *[Paths]* section. The title, page header and link text for the custom page default to 'PageID', but can be overridden in the *[Titles]*, *[PageHeaders]* and *[Links]* sections.

Every custom page is built using the *HTML template* whose name matches the page ID, if one exists; otherwise, the *Layout* template is used.

Version	Changes
6.0	Added support for `SectionType=BulletPoints`; every parameter (not just `PageContent`) may contain *skool macros*
5.4	Added the `SectionType` parameter
5.3	Added the `SectionPrefix` parameter
3.5	The `JavaScript` parameter specifies the JavaScript file(s) to use
2.1	New

8.5.13 [PageHeaders]

The `PageHeaders` section defines the header text for every page in the HTML disassembly. Each line has the form:

```
PageID=[prefix<>]suffix
```

where:

- `PageID` is the ID of the page
- `prefix` is the page header prefix (displayed to the left of the game logo); if present, this must be separated from the suffix by <>
- `suffix` is the page header suffix (displayed to the right of the game logo)

Recognised page IDs are:

- `Asm-b` - disassembly pages for 'b' blocks (default: 'Data')
- `Asm-c` - disassembly pages for 'c' blocks (default: 'Routines')
- `Asm-g` - disassembly pages for 'g' blocks (default: 'Game status buffer')
- `Asm-s` - disassembly pages for 's' blocks (default: 'Unused')
- `Asm-t` - disassembly pages for 't' blocks (default: 'Messages')
- `Asm-u` - disassembly pages for 'u' blocks (default: 'Unused')
- `Asm-w` - disassembly pages for 'w' blocks (default: 'Data')
- `AsmSinglePage` - the disassembly page (when writing a single-page disassembly)
- `Bugs` - the 'Bugs' page
- `Changelog` - the 'Changelog' page
- `DataMap` - the 'Data' memory map page
- `Facts` - the 'Trivia' page
- `GameIndex` - the disassembly index page (default: 'The complete<>RAM disassembly')
- `GameStatusBuffer` - the 'Game status buffer' page
- `Glossary` - the 'Glossary' page
- `GraphicGlitches` - the 'Graphic glitches' page
- `MemoryMap` - the 'Everything' memory map page
- `MessagesMap` - the 'Messages' memory map page
- `Pokes` - the 'Pokes' page
- `RoutinesMap` - the 'Routines' memory map page
- `UnusedMap` - the 'Unused addresses' memory map page

Every parameter in this section may contain *skool macros*.

The default header text for a page is the same as the title defined in the *[Titles]* section, except where indicated above.

The `Asm-*` parameters are formatted with an `entry` dictionary identical to the one that is available in the *asm* template.

The header text for a page defined by a *[MemoryMap:*]*, *[OtherCode:*]* or *[Page:*]* section defaults to the page's title, but can be overridden in this section.

The header text of each disassembly page for the entries belonging to a group defined in the *[EntryGroups]* section also defaults to the page's title, but can be overridden in this section.

Version	Changes
8.0	Added support for specifying a prefix and suffix; an `entry` dictionary is available when formatting `Asm-*` parameters; added the `GameIndex` page ID
6.0	The default header for `Asm-t` pages is 'Messages'; page headers may contain *skool macros*
5.3	Added the `AsmSinglePage` page ID
4.0	New

8.5.14 [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:

- `AsmSinglePage` - the disassembly page (when writing a single-page disassembly; default: `asm.html`)
- `Bugs` - the 'Bugs' page (default: `reference/bugs.html`)
- `Changelog` - the 'Changelog' page (default: `reference/changelog.html`)
- `CodeFiles` - the format of the disassembly page filenames (default: `{address}.html`)
- `DataMap` - the 'Data' memory map page (default: `maps/data.html`)
- `Facts` - the 'Trivia' page (default: `reference/facts.html`)
- `GameIndex` - the 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`)
- `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`)
- `UDGFilename` - the format of the default filename for images created by the *#UDG* macro (default: `udg{addr}_{attr}x{scale}`); this is a standard Python format string that recognises the macro parameters `addr`, `attr` and `scale`
- `UnusedMap` - the 'Unused addresses' memory map page (default: `maps/unused.html`)

Recognised directory IDs and their default paths are:

- `AudioPath` - the directory in which audio files are assumed to be by the *#AUDIO* macro (default: `audio`)
- `CodePath` - the directory in which the disassembly pages are written (default: `asm`)

- `FontImagePath` - the directory in which font images (created by the *#FONT* macro) are placed (default: `{ImagePath}/font`)
- `FontPath` - the directory in which font files specified by the `Font` parameter in the *[Game]* section are placed (default: `.`)
- `ImagePath` - the base directory in which images are placed (default: `images`)
- `JavaScriptPath` - the directory in which JavaScript files specified by the `JavaScript` parameter in the *[Game]* section and *[Page:*]* sections are placed (default: `.`)
- `ScreenshotImagePath` - the directory in which screenshot images (created by the *#SCR* macro) are placed (default: `{ImagePath}/scr`)
- `StyleSheetPath` - the directory in which CSS files specified by the `StyleSheet` parameter in the *[Game]* section are placed (default: `.`)
- `UDGImagePath` - the directory in which UDG images (created by the *#UDG* or *#UDGARRAY* macro) are placed (default: `{ImagePath}/udgs`)

Every parameter in this section may contain *skool macros*.

The `CodeFiles` parameter contains a standard Python format string that specifies the format of a disassembly page filename based on the address of the routine or data block. The default format string is `{address}.html`, which produces decimal addresses (e.g. `65280.html`). To produce 4-digit, upper case hexadecimal addresses instead (e.g. `FF00.html`), change `CodeFiles` to `{address:04X}.html`. Or to produce 4-digit, upper case hexadecimal addresses if the `--hex` option is used with *skool2html.py*, and decimal addresses otherwise: `{address#IF({mode[base]}==16)(:04X)}.html`.

Version	Changes
8.7	Added the `AudioPath` directory ID
6.3	Added the `ImagePath` directory ID and the ability to define one image path ID in terms of another
6.0	Paths may contain *skool macros*; added the `UDGFilename` parameter (which used to live in the *[Game]* section)
5.3	Added the `AsmSinglePage` file ID
4.3	Added the `CodeFiles` file ID
3.1.1	Added the `FontPath` directory ID
2.5	Added the `UnusedMap` file ID
2.2.5	Added the `Changelog` file ID
2.1.1	Added the `CodePath` directory ID
2.0.5	Added the `FontImagePath` directory ID
2.0	New

8.5.15 [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
```

`fname` may contain the special wildcard characters `*`, `?` and `[]`, which are expanded as follows:

- `*` - matches any number of characters
- `**` - matches any files and zero or more directories and subdirectories
- `?` - matches any single character
- `[seq]` - matches any character in `seq`; `seq` may be a simple sequence of characters (e.g. `abcde`) or a range (e.g. `a-e`)
- `[!seq]` - matches any character not in `seq`

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
8.0	Added support for the `**` pattern
6.3	Added support for pathname pattern expansion using wildcard characters
3.6	New

8.5.16 [Template:*]

Each `Template:*` section defines a template used to build an HTML page (or part of one).

To see the contents of the default templates, run the following command:

```
$ skool2html.py -r Template:
```

For more information, see *HTML templates*.

Version	Changes
4.0	New

8.5.17 [Titles]

The `Titles` section defines the title (i.e. text used to compose the `<title>` element) for every page in the HTML disassembly. Each line has the form:

```
PageID=title
```

where:

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

Recognised page IDs and their default titles are:

- `Asm-b` - disassembly pages for 'b' blocks (default: 'Data at {entry[address]}')
- `Asm-c` - disassembly pages for 'c' blocks (default: 'Routine at {entry[address]}')

- `Asm-g` - disassembly pages for 'g' blocks (default: 'Game status buffer entry at {entry[address]}')
- `Asm-s` - disassembly pages for 's' blocks (default: 'Unused RAM at {entry[address]}')
- `Asm-t` - disassembly pages for 't' blocks (default: 'Text at {entry[address]}')
- `Asm-u` - disassembly pages for 'u' blocks (default: 'Unused RAM at {entry[address]}')
- `Asm-w` - disassembly pages for 'w' blocks (default: 'Data at {entry[address]}')
- `AsmSinglePage` - the disassembly page (when writing a single-page disassembly; default: 'Disassembly')
- `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')
- `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')

Every parameter in this section may contain *skool macros*.

The `Asm-*` parameters are formatted with an `entry` dictionary identical to the one that is available in the *asm* template.

The title of a page defined by a *[MemoryMap:*]*, *[OtherCode:*]* or *[Page:*]* section defaults to the page ID, but can be overridden in this section.

The title of each disassembly page for the entries belonging to a group defined in the *[EntryGroups]* section defaults to the title for that page's entry type, but can be overridden in this section.

Version	Changes
8.0	An `entry` dictionary is available when formatting `Asm-*` parameters; the default title for each `Asm-*` page includes the entry address as a replacement field
6.0	The default title for `Asm-t` pages is 'Text at'; titles may contain *skool macros*
5.3	Added the `AsmSinglePage` page ID
4.0	Added the `Asm-*` page IDs
2.5	Added the `UnusedMap` page ID
2.2.5	Added the `Changelog` page ID
2.0.5	New

8.5.18 Box pages

A 'box page' is an HTML page that contains entries (blocks of arbitrary text) distinguished by alternating background colours, and a table of contents (links to each entry). It is defined by a *[Page:*]* section that contains a `SectionPrefix` parameter, which determines the prefix of the ref file sections from which the entries are built.

SkoolKit defines some box pages by default. Their names and the ref file sections that can be used to define their entries are as follows:

- `Bugs` - `[Bug:title]` or `[Bug:anchor:title]`
- `Changelog` - `[Changelog:title]` or `[Changelog:anchor:title]`
- `Facts` - `[Fact:title]` or `[Fact:anchor:title]`
- `Glossary` - `[Glossary:title]` or `[Glossary:anchor:title]`
- `GraphicGlitches` - `[GraphicGlitch:title]` or `[GraphicGlitch:anchor:title]`
- `Pokes` - `[Poke:title]` or `[Poke:anchor:title]`

To see the contents of the default *[Page:*]* sections, run the following command:

```
$ skool2html.py -r Page:
```

If `anchor` is omitted from an entry section name, it defaults to the title converted to lower case with parentheses and whitespace characters replaced by underscores.

By default, a box page entry section is parsed as a sequence of paragraphs separated by blank lines. For example:

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

Second paragraph.

...
```

However, if the `SectionType` parameter in the *[Page:*]* section is set to `ListItems`, each entry section is parsed as a sequence of single-line list items with indentation. For example:

```
[Changelog:title]
Intro text.

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

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

The intro text and the first top-level item must be separated by a blank line. Lower-level items are created by using indentation, as shown. Blank lines between items are optional and are ignored. If the intro text is a single hyphen (`-`), it is not included in the final HTML rendering.

If your list items are long, you might prefer to set the `SectionType` parameter to `BulletPoints`; in that case, each entry section is parsed as a sequence of multi-line list items prefixed by '-'. For example:

```
[Changes:title]
Intro text.
```

(continues on next page)

(continued from previous page)

```
- First top-level item,
  split over two lines.
  - First subitem, also
    split over two lines.
  - Second subitem, on one line this time.
    - First subsubitem,
      this time split
      over three lines.

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

An entry section's `anchor`, `title` and contents may contain HTML markup and *skool macros*.

Changed in version 6.0: Added support for parsing an entry section as a sequence of multi-line list items prefixed by '-' (`SectionType=BulletPoints`). The `anchor` and `title` of an entry section name may contain *skool macros*.

Changed in version 5.4: The `anchor` part of an entry section name is optional.

8.5.19 Appending content

Content may be appended to an existing ref file section defined elsewhere by adding a '+' suffix to the section name. For example, to add a line to the *[Game]* section:

```
[Game+]
AddressAnchor={address:04x}
```

New in version 7.0.

8.5.20 Ref file comments

A comment may be added to a ref file by starting a line with a semicolon. For example:

```
; This is a comment
```

If a non-comment line in a ref file section needs to start with a semicolon, it can be escaped by doubling it:

```
[Glossary:term]
<code>
;; This is not a ref file comment
</code>
```

The content of this section will be rendered thus:

```
<code>
; This is not a ref file comment
</code>
```

8.5.21 Square brackets

If a ref file section needs to contain a line that looks like a section header (i.e. like `[SectionName]`), then to prevent that line from being parsed as a section header it can be escaped by doubling the opening square bracket:

```
[Glossary:term]
<code>
[[This is not a section header]
</code>
```

The content of this section will be rendered thus:

```
<code>
[This is not a section header]
</code>
```

In fact, any line that starts with two opening square brackets will be rendered with the first one removed.

New in version 4.0.

8.6 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.

8.6.1 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*.

8.6.2 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
@rfix+begin
       LD A,H
       OR L
@rfix+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*.

8.6.3 ASM directives

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

@assemble

The `@assemble` directive controls whether assembly language instructions, `DEFB`, `DEFM`, `DEFS` and `DEFW` statements, and *@defb*, *@defs* and *@defw* directives are converted into byte values for the purpose of populating the memory snapshot.

```
@assemble=H,A
```

`H` is an integer value that determines what is converted in HTML mode, and `A` is an integer value that determines what is converted in ASM mode:

- `0` - do not convert anything (this is the default in ASM mode)

- `1` - convert `DEFB`, `DEFM`, `DEFS` and `DEFW` statements and `@defb`, `@defs` and `@defw` directives only (this is the default in HTML mode)
- `2` - convert assembly language instructions as well

If `H` or `A` is blank or omitted, its value is left unchanged.

For example:

```
; The eight bytes of code in this routine are also used as UDG data.
; .
; #HTML(#UDG44919)
@assemble=2
c44919 LD DE,46572   ;
 44922 CP 200        ;
 44924 JP 45429      ;
@assemble=1
```

The `@assemble=2` directive 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 image created by the `#UDG` macro would be blank.

Version	Changes
7.0	The accepted values are 0, 1 and 2 (previously -1, 0 and 1)
6.3	Added support for specifying what's converted in HTML mode and ASM mode separately, and for switching off conversion entirely
6.1	Added the ability to assemble instructions whose operands contain arithmetic expressions
5.0	New

@bfix

The `@bfix` directive replaces, inserts or removes a label, instruction and comment in *@bfix mode*.

```
@bfix=[>][|][+][/][LABEL:][INSTRUCTION][; comment]
```

or, when removing instructions:

```
@bfix=!addr1[-addr2]
```

- `>` - if this marker is present, `INSTRUCTION` is inserted before the current instruction instead of replacing it
- `|` - if this marker is present, `INSTRUCTION` overwrites any overlapping instructions instead of pushing them aside
- `+` - if this marker is present, `INSTRUCTION` is inserted after the current instruction instead of replacing it
- `/` - if this marker is present, any remaining comment lines are removed
- `LABEL` is the replacement label; if not given, any existing label is left unchanged
- `INSTRUCTION` is the replacement instruction; if not given, the existing instruction is left unchanged
- `comment` is the replacement comment; if not given, the existing comment is left unchanged
- `addr1` is the address of the first instruction to remove
- `addr2`, if given, is the address of the last instruction to remove

For example:

```
@label=CMASK
@bfix=BMASK: AND B ; Apply the mask
 29713 AND C       ; This should be 'AND B'
```

This `@bfix` directive replaces the instruction `AND C` with `AND B`, replaces the label `CMASK` with `BMASK`, and also replaces the comment.

Comment continuation lines can be replaced, removed or added by using additional `@bfix` directives. For example, to replace both comment lines of an instruction that has two:

```
@bfix=AND B  ; This directive replaces the first comment line
@bfix=       ; and this directive replaces the second comment line
 29713 AND C ; Both of these comment lines
             ; will be replaced
```

To add a second comment line to an instruction that has only one:

```
@bfix=AND B  ; This directive replaces the first comment line
@bfix=       ; and this directive adds a second comment line
 29713 AND C ; This comment line will be replaced
```

To replace two comment lines with one:

```
@bfix=/AND B ; The '/' in this directive effectively terminates the comment
 29713 AND C ; This comment line will be replaced
             ; and this one will be removed
```

A single instruction can be replaced with two or more by using the `|` (overwrite) marker. For example, to replace `LD HL,0` with `LD L,0` and `LD H,L`:

```
@bfix=|LD L,0  ; Clear L
@bfix=|LD H,L  ; Clear H
 36671 LD HL,0 ; Clear HL
```

Two or more instructions can also be replaced with a single instruction. For example, to replace `XOR A` and `INC A` with `LD A,1`:

```
@bfix=|LD A,1
 49912 XOR A
 49913 INC A
```

A sequence of instructions can be replaced by chaining `@bfix` directives. For example, to swap two `XOR` instructions:

```
@bfix=|XOR C
@bfix=|XOR B
 51121 XOR B
 51122 XOR C
```

This is equivalent to:

```
@bfix=XOR C
 51121 XOR B
@bfix=XOR B
 51122 XOR C
```

Note that when `@bfix` directives are chained like this, the second and subsequent directives replace instruction comments in their entirety, instead of line by line. For example:

```
@bfix=|LD A,D ; Set A=D
@bfix=|XOR B  ; Flip the bits
 51121 LD A,B ; Set A=B
 51122 XOR C  ; XOR the contents of the accumulator with the contents of the
              ; C register
```

replaces both comment lines of the instruction at 51122 with 'Flip the bits'.

A sequence of instructions can be inserted before the current instruction by using the > marker. For example:

```
 47191 EX DE,HL
; A mid-block comment.
@bfix=>LD (HL),C
@bfix=>INC HL
 47192 LD (HL),B
```

This will insert `LD (HL),C` and `INC HL` between `EX DE,HL` and `LD (HL),B`. The mid-block comment that was above `LD (HL),B` will now be above `LD (HL),C`.

A sequence of instructions can be inserted after the current instruction (without first specifying a replacement for it) by using the + marker. For example:

```
@bfix=+LD (HL),C
@bfix=INC HL
 47191 EX DE,HL
; A mid-block comment.
 47192 LD (HL),B
```

This will insert `LD (HL),C` and `INC HL` between `EX DE,HL` and `LD (HL),B`. In this case, the mid-block comment above `LD (HL),B` will remain there.

The current instruction can be replaced and a sequence of instructions inserted after it by chaining `@bfix` directives. For example:

```
@bfix=LD (HL),B  ; {Save B and C here
@bfix=INC HL     ;
@bfix=LD (HL),C  ; }
 61125 LD (HL),A ; Save A here
 61126 RET
```

This will replace `LD (HL),A` with `LD (HL),B` and insert `INC HL` and `LD (HL),C` before the `RET` instruction.

An instruction can be removed by using the `!` notation. For example:

```
 51184 XOR A
@bfix=!51185
 51185 AND A ; This instruction is redundant
 51186 RET
```

This removes the redundant instruction at 51185.

An entire entry can be removed by specifying an address range that covers every instruction in the entry:

```
; Unused
@bfix=!40000-40001
c40000 NOP
 40001 RET
```

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment

@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 will 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
```

@defb

The `@defb` directive makes *skool2asm.py* and *skool2html.py* insert byte values into the memory snapshot at a given address.

```
@defb=[address:]value1[,value2...]
```

- `address` is the address
- `value1`, `value2` etc. are the byte values (as might appear in a `DEFB` statement)

If `address` is omitted, it defaults to the address immediately after the last byte of the previous `@defb`, *@defs* or *@defw* directive preceding the same instruction (if one exists), or to the address of the next instruction otherwise.

The sequence of comma-separated values may be followed by a semicolon (`;`) and arbitrary text, which will be ignored.

For example:

```
@defb=30000:5,"Hello" ; Welcome message
```

This will insert the value 5 followed by the ASCII codes of the characters in "Hello" into the memory snapshot at address 30000.

`@defb` directives are also processed by *sna2skool.py* when it is run on a control file; thus the `@defb` directive can be used to override the contents of the snapshot that is read by *sna2skool.py*.

`@defb` directives are also processed by *skool2bin.py* when the `--data` option is used.

Version	Changes
8.1	The `address` parameter is optional
6.3	New

@defs

The `@defs` directive makes *skool2asm.py* and *skool2html.py* insert a sequence of byte values into the memory snapshot at a given address.

```
@defs=[address:]length[,value]
```

- `address` is the address
- `length` is the length of the sequence
- `value` is the byte value (default: 0)

If `address` is omitted, it defaults to the address immediately after the last byte of the previous *@defb*, `@defs` or *@defw* directive preceding the same instruction (if one exists), or to the address of the next instruction otherwise.

The directive may be followed by a semicolon (`;`) and arbitrary text, which will be ignored.

For example:

```
@defs=30000:5,$FF ; Five 255s
```

This will insert the value 255 into the memory snapshot at addresses 30000-30004.

`@defs` directives are also processed by *sna2skool.py* when it is run on a control file; thus the `@defs` directive can be used to override the contents of the snapshot that is read by *sna2skool.py*.

`@defs` directives are also processed by *skool2bin.py* when the `--data` option is used.

Version	Changes
8.1	The `address` parameter is optional
6.3	New

@defw

The `@defw` directive makes *skool2asm.py* and *skool2html.py* insert word values into the memory snapshot at a given address.

```
@defw=[address:]value1[,value2...]
```

- `address` is the address
- `value1`, `value2` etc. are the word values (as might appear in a `DEFW` statement)

If `address` is omitted, it defaults to the address immediately after the last byte of the previous *@defb*, *@defs* or `@defw` directive preceding the same instruction (if one exists), or to the address of the next instruction otherwise.

The sequence of comma-separated values may be followed by a semicolon (`;`) and arbitrary text, which will be ignored.

For example:

```
@defw=30000:32768,32775 ; Message addresses
```

This will insert the word values 32768 and 32775 into the memory snapshot at addresses 30000 and 30002.

`@defw` directives are also processed by *sna2skool.py* when it is run on a control file; thus the `@defw` directive can be used to override the contents of the snapshot that is read by *sna2skool.py*.

`@defw` directives are also processed by *skool2bin.py* when the `--data` option is used.

Version	Changes
8.1	The `address` parameter is optional
6.3	New

@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 by *skool2asm.py*.

See also *@start*.

Version	Changes
2.2.2	New

@equ

The `@equ` directive defines an EQU directive that will appear in the ASM output.

```
@equ=label=value
```

- `label` is the label
- `value` is the value assigned to the label

For example:

```
@equ=ATTRS=22528
c32768 LD HL,22528
```

This will produce an EQU directive (`ATTRS EQU 22528`) in the ASM output, and replace the operand of the instruction at 32768 with a label: `LD HL,ATTRS`.

Version	Changes
5.4	New

@expand

The `@expand` directive specifies an arbitrary piece of text - intended to consist of one or more *SMPL macros* - that will be expanded by the ASM writer or HTML writer during initialisation (before any skool macros that appear in skool file annotations or ref file sections are expanded).

```
@expand=text
```

- `text` is the text to expand

For example:

```
@expand=#DEF(#MAX(a,b) #IF($a>$b)($a,$b))
```

This `@expand` directive passes the given *#DEF* macro to the ASM writer or HTML writer for expansion during initialisation; this has the effect of making the user-defined `#MAX` macro available for use immediately anywhere in the skool file (and any secondary skool files if the directive appears in the main skool file) or ref files.

If `text` begins with +, it is appended to the text of the previous `@expand` directive (with the + removed); this enables long macro definitions to be split over multiple lines. For example:

```
@expand=#DEF(#OLIST()(items)
@expand=+  #LET(n=1)
@expand=+  #LIST
@expand=+  #FOREACH($items)(item,{ #EVAL({n}). item } #LET(n={n}+1))
@expand=+  LIST#
@expand=+)
```

These `@expand` directives make the `#OLIST` macro available, which can then be used to create a numbered list of items:

```
#OLIST/a,b,c/
```

See also the `Expand` parameter in the *[Config]* section, which may be used instead of the `@expand` directive if there is no need to expand `text` in ASM mode.

Version	Changes
8.4	Added support for the + notation
8.2	New

@if

The `@if` directive conditionally processes other ASM directives based on the value of an arithmetic expression.

```
@if(expr)(true[,false])
```

- `expr` is the arithmetic expression, which may contain *replacement fields*
- `true` is processed when `expr` is true
- `false` (if given) is processed when `expr` is false

See *Numeric parameters* for details on the operators that may be used in the `expr` parameter.

For example:

```
@if({mode[case]}==1))(replace=/#hl/hl,replace=/#hl/HL)
```

would process `replace=/#hl/hl` if in lower case mode, or `replace=/#hl/HL` otherwise.

The `true` and `false` parameters may be supplied in the same way as they are for the *#IF* macro. See *String parameters* for more details.

Version	Changes
6.4	New

@ignoreua

The `@ignoreua` directive suppresses warnings that would otherwise be printed (during the rendering phase) concerning addresses not converted to labels in the comment that follows. The comment may be an entry title, an entry description, a register description section, a block start comment, a mid-block comment, a block end comment, or an instruction-level comment.

```
@ignoreua[=addr1[,addr2...]]
```

- `addr1`, `addr2` etc. are the addresses to suppress warnings for; if none are specified, warnings for all addresses are suppressed

Although specifying a list of addresses is optional, doing so has the advantage that if another unconvertible address is added to the comment later on, a warning will appear for it, at which point you can decide whether to fix it (in case it was added by mistake) or add it to the list.

To apply the directive to an entry title:

```
@ignoreua=32768
; Prepare data at 32768
c32768 LD A,(HL)
```

If the `@ignoreua` directive were not present, a warning would be printed 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.
c49152 LD A,(HL)
```

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

To apply the directive to a register description section:

```
; Prepare data in page 128
;
; This routine operates on the data in page 128.
;
@ignoreua
; HL 32768
c49152 LD A,(HL)
```

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

To apply the directive to a block start comment:

```
; Prepare data in page 128
;
; This routine operates on the data in page 128.
;
; HL 128*256
;
@ignoreua
; First pick up the byte at 32768.
c49152 LD A,(HL)
```

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

Version	Changes
8.1	Added the ability to specify the addresses for which to suppress warnings
4.2	Added support for register description sections
2.4.1	Added support for entry titles, entry descriptions, mid-block comments and block end comments

@isub

The `@isub` directive replaces, inserts or removes a label, instruction and comment in *@isub mode*.

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

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment

@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 labels for numeric values in the operand of the next instruction:

```
@keep[=val1[,val2...]]
```

- `val1`, `val2` etc. are the values to keep; if none are specified, all values are kept

In HTML mode, the `@keep` directive also prevents the operand from being hyperlinked.

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.

Version	Changes
6.2	Added the ability to specify the values to keep; the `@keep` directive is applied to instructions that have been replaced by an *@isub*, *@ssub* or *@rsub* directive

@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`.

If `LABEL` is blank (`@label=`), the next instruction will have its entry point marker removed (if it has one), and be prevented from having a label automatically generated.

If `LABEL` starts with `*` (e.g. `@label=*LOOP`), the next instruction will be marked as an entry point (as if the instruction line in the skool file started with `*`), in addition to having its label set.

If `LABEL` is just `*` (`@label=*`), the next instruction will be marked as an entry point, and have a label automatically generated.

skool2asm.py automatically uses labels defined by the `@label` directive. *skool2html.py* includes them in its output if the `--asm-labels` option is used.

`@label` directive values are also checked by *sna2skool.py* while reading a control file. They can be used to prevent an entry point marker from being added to an instruction where it otherwise would be (`@label=`), or force one to be added where it otherwise wouldn't (`@label=*`).

Version	Changes
7.0	An entry point marker (`*`) can be added to or removed from the next instruction
6.3	`LABEL` may be blank (to prevent the next instruction from having a label automatically generated)

@nowarn

The `@nowarn` directive suppresses any warnings that would otherwise be reported (during the parsing phase) for the next instruction concerning:

- an address in a `LD` instruction operand being replaced with a label (if the instruction has not been replaced by a `@*sub` or `@*fix` directive)
- an address in an instruction operand not being replaced with a label (because the address has no label defined)

```
@nowarn[=addr1[,addr2...]]
```

- `addr1`, `addr2` etc. are the addresses to suppress warnings for; if none are specified, warnings for all addresses are suppressed

For example:

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

If this `@nowarn` directive were not present, a warning would be printed 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 (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).

Version	Changes
8.1	Added the ability to specify the addresses for which to suppress warnings

@ofix

The `@ofix` directive replaces, inserts or removes a label, instruction and comment in *@ofix mode*.

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

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment

@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 makes *skool2asm.py* insert an `ORG` assembler directive.

```
@org[=address]
```

- `address` is the `ORG` address; if not specified, it defaults to the address of the next instruction

Note that the `@org` directive works only on the first instruction in an entry.

The `@org` directive also forces *skool2bin.py* to place the next instruction at the given address.

Version	Changes
6.3	The `address` parameter is optional

@refs

The `@refs` directive manages the addresses of the referrers of (i.e. the routines that jump to or call) the next instruction.

```
@refs=[addr1[,addr2...]][:raddr1[,raddr2...]]
```

- `addr1`, `addr2` etc. are addresses to add to the list of referrers
- `raddr1`, `raddr2` etc. are addresses to remove from the list of referrers

This directive can be used to declare one or more additional referrers for an instruction that would not otherwise be identified by the *instruction utility* or *snapshot reference calculator* (e.g. because the instruction is jumped to indirectly via `JP (HL)` or `RET`). As a result:

- *sna2skool.py* will attach an entry point marker (`*`) to the instruction when reading a control file, and include the additional referrers in any comment generated for the entry point (when the `ListRefs` *configuration parameter* is `1` or `2`)
- *snapinfo.py*, when generating a *call graph*, will add an edge between a node representing an additional referrer and the node representing the routine that contains the instruction
- the addresses of the additional referrers become available to the special `EREF` and `REF` variables of the *#FOREACH* macro

`@refs` can also be used to remove one or more referrer addresses that have been added automatically (because the instruction is jumped to or called directly). As a result:

- *sna2skool.py* will remove the referrers from any comment generated for the entry point (when the `ListRefs` *configuration parameter* is `1` or `2`), and remove any entry point marker (`*`) from the instruction if all the referrers have been removed
- *snapinfo.py*, when generating a *call graph*, will not place an edge between a node representing a removed referrer and the node representing the routine that contains the instruction
- the addresses of the removed referrers will not be available to the special `EREF` and `REF` variables of the *#FOREACH* macro

For example:

```
@ 40000 refs=32768:49152
```

This `@refs` directive (in a control file) declares that the routine at 32768 uses the entry point at 40000, and the routine at 49152 does not.

Version	Changes
8.2	New

@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

@remote

The `@remote` directive creates a remote entry in a skool file. A remote entry enables `JR`, `JP` and `CALL` instructions to be hyperlinked to an entry defined in another skool file.

```
@remote=code:address[,address2...]
```

- `code` is the ID of the disassembly defined in the other skool file
- `address` is the address of the remote entry
- `address2` etc. are addresses of other entry points in the remote entry

For example:

```
@remote=main:29012,29015
```

This directive, if it appeared in a secondary skool file, would enable references to the routine at 29012 and its entry point at 29015 in the main disassembly. It would also enable the *#R* macro to create a hyperlink to a remote entry point using the form:

```
#R29015@main
```

Version	Changes
6.3	New

@replace

The `@replace` directive replaces strings that match a regular expression in skool file annotations and ref file section names and contents.

```
@replace=/pattern/repl
```

or:

```
@replace=/pattern/repl/
```

- `pattern` is the regular expression
- `repl` is the replacement string

(If the second form is used, any text appearing after the terminating `/` is ignored.)

For example:

```
@replace=/#copy/#CHR(169)
```

This `@replace` directive replaces all instances of `#copy` with `#CHR(169)`.

If `/` appears anywhere in `pattern` or `repl`, then an alternative separator should be used; for example:

```
@replace=|n/a|not applicable
```

As a convenience for dealing with decimal and hexadecimal numbers, wherever `\i` appears in `pattern`, it is replaced by a regular expression group that matches a decimal number or a hexadecimal number preceded by `$`. For example:

```
@replace=/#udg\i,\i/#UDG(\1,#PEEK\2)
```

This `@replace` directive would replace `#udg$a001,40960` with `#UDG($a001,#PEEK40960)`.

Note that string replacements specified by `@replace` directives are made before skool macros are expanded, and in the order in which the directives appear in the skool file. For example, if we have:

```
@replace=/#foo\i/#bar\1
@replace=/#bar\i/#EVAL\1,16
```

then `#foo31` would be replaced by `#EVAL31,16`, but if these directives were reversed:

```
@replace=/#bar\i/#EVAL\1,16
@replace=/#foo\i/#bar\1
```

then `#foo31` would be replaced by `#bar31`.

See also the *#DEF* macro, which is more flexible than `@replace` for defining new macros.

Version	Changes
6.0	Replaces strings in ref file section names
5.1	New

@rfix

The `@rfix` directive replaces, inserts or removes a label, instruction and comment in *@rfix mode*.

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

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment
5.2	New

@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*.

@rom

The `@rom` directive inserts a copy of the 48K ZX Spectrum ROM into the internal memory snapshot constructed from the contents of the skool file.

```
@rom
```

Some reasons why you might want to do this are:

- to simulate the execution of ROM code (whether called by game code or otherwise) with the *#SIM* macro
- to create a WAV file of the ROM's 'BEEPER' subroutine in action with the *#AUDIO* macro
- to gain access to the Spectrum character set at 0x3D00 for the purpose of creating images of text

Version	Changes
8.7	New

@rsub

The `@rsub` directive replaces, inserts or removes a label, instruction and comment in *@rsub mode*.

The syntax is equivalent to that for the *@rfix* directive.

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment

@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 must 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`)
- `table-border-horizontal` - the character to use for the horizontal borders of a table defined by a *#TABLE* macro (default: `-`); if two characters are specified, the first is used for the external borders and the second is used for the internal borders
- `table-border-join` - the character to use for the horizontal and vertical border joins of a table defined by a *#TABLE* macro (default: `+`)
- `table-border-vertical` - the character to use for the vertical borders of a table defined by a *#TABLE* macro (default: `|`)
- `table-row-separator` - the character used to separate non-header cells in adjacent rows of a table defined by a *#TABLE* macro; by default, such cells are not separated
- `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
8.1	Added the `table-row-separator` property
8.0	Added the `table-border-horizontal`, `table-border-join` and `table-border-vertical` properties
3.4	Added the `handle-unsupported-macros` and `wrap-column-width-min` properties
3.3.1	Added the `comment-width-min`, `indent`, `instruction-width`, `label-colons`, `line-width` and `warnings` properties
3.2	New

@ssub

The `@ssub` directive replaces, inserts or removes a label, instruction and comment in *@ssub mode*.

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

Version	Changes
7.1	Added support for the + marker (to insert an instruction after the current one)
7.0	Added support for specifying the replacement comment over multiple lines, replacing the label, and inserting, overwriting and removing instructions
6.4	Added support for replacing the comment

@ssub block directives

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

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

Version	Changes
4.4	New

@start

The `@start` directive indicates where to start parsing the skool file for the purpose of generating ASM output. Everything before the `@start` directive is ignored by *skool2asm.py*.

See also *@end*.

@writer

The `@writer` directive specifies the name of the Python class to use to generate ASM output. It must 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
3.3.1	Added support for specifying a module outside the module search path
3.1	New

8.7 ASM templates

Each line of output produced by *skool2asm.py* is built from a template. A template contains 'replacement fields' - identifiers enclosed by braces (`{` and `}`) - that are replaced by appropriate content (such as a label or register name) when the template is formatted.

The default templates can be overridden by custom templates read from a file by setting the `Templates` configuration parameter of *skool2asm.py*. To define a custom template, specify its name in square brackets on a line of its own, and follow it with the content of the template. For example:

```
[org]
.{org} {address}
```

8.7.1 comment

The `comment` template is used to format a line in an entry title, entry description, block start comment, mid-block comment, or block end comment.

```
; {text}
```

The following identifier is available:

- `text` - the text of the comment line

This template is also used to used to format lines between paragraphs in comments, with `text` set to an empty string.

8.7.2 equ

The `equ` template is used to format an EQU directive produced by *@equ*.

```
{label} {equ} {value}
```

The following identifiers are available:

- `equ` - 'EQU' or 'equ' (depending on the case)
- `label` - the label
- `value` - the value

8.7.3 instruction

The `instruction` template is used to format an instruction line or instruction comment continuation line.

```
{indent}{operation:{width}} {sep} {text}
```

The following identifiers are available:

- `indent` - the instruction indent (as defined by the `indent` property)
- `operation` - either the operation (e.g. 'XOR A'), or an empty string (if formatting a comment continuation line)
- `sep` - the comment separator (';' if there is a comment, an empty string otherwise)
- `text` - the text of the comment line

- `width` - the width of the instruction field (as defined by the `instruction-width` property)

The `indent` and `instruction-width` properties can be set by either the *@set* directive, or the `Set-indent` and `Set-instruction-width` configuration parameters of *skool2asm.py*.

8.7.4 label

The `label` template is used to format an instruction label.

```
{label}{suffix}
```

The following identifiers are available:

- `label` - the instruction label
- `suffix` - ':' or an empty string (as defined by the `label-colons` property)

The `label-colons` property can be set by either the *@set* directive, or the `Set-label-colons` configuration parameter of *skool2asm.py*.

8.7.5 org

The `org` template is used to format an ORG directive produced by *@org*.

```
{indent}{org} {address}
```

The following identifiers are available:

- `address` - the ORG address (as a string)
- `indent` - the instruction indent (as defined by the `indent` property)
- `org` - 'ORG' or 'org' (depending on the case)

The `indent` property can be set by either the *@set* directive, or the `Set-indent` configuration parameter of *skool2asm.py*.

8.7.6 register

The `register` template is used to format lines in the register section of an entry header.

```
; {prefix:>{prefix_len}}{reg:<{reg_len}} {text}
```

The following identifiers are available:

- `max_reg_len` - the maximum length of all register names in the register section
- `prefix` - the register prefix (e.g. 'In:' or 'O:'), or an empty string (if formatting a register description continuation line)
- `prefix_len` - the maximum length of all register prefixes in the register section
- `reg` - the register name (e.g. 'HL'), or an empty string (if formatting a register description continuation line)
- `reg_len` - the length of the register name
- `text` - the text of a line of the register description

Changed in version 8.7: Added the '<' alignment specifier to the `reg` field (to handle the case where `reg_len` is 0).

Changed in version 8.0: Added the `max_reg_len` identifier.

8.8 HTML templates

Every page in an HTML disassembly is built from the *Layout* template and zero or more subtemplates defined by *[Template:*]* sections in the ref file.

A template may contain 'replacement fields' - identifiers enclosed by braces (`{` and `}`) - that are replaced by appropriate content (typically derived from the skool file or a ref file section) when the template is formatted. The following 'universal' identifiers are available in every template:

- `Game` - a dictionary of the parameters in the *[Game]* section
- `SkoolKit` - a dictionary of parameters relevant to the page currently being built

The parameters in the `SkoolKit` dictionary are:

- `include` - the name of the subtemplate used to format the content between the page header and footer
- `index_href` - the relative path to the disassembly index page
- `javascripts` - a list of javascript objects; each one has a single attribute, `src`, which holds the relative path to the JavaScript file
- `page_header` - a two-element list containing the page header prefix and suffix (as defined in the *[PageHeaders]* section)
- `page_id` - the page ID (e.g. `GameIndex`, `MemoryMap`)
- `path` - the page's filename, including the full path relative to the root of the disassembly
- `stylesheets` - a list of stylesheet objects; each one has a single attribute, `href`, which holds the relative path to the CSS file
- `title` - the title of the page (as defined in the *[Titles]* section)

The parameters in a dictionary are accessed using the `[param]` notation; for example, wherever `{Game[Copyright]}` appears in a template, it is replaced by the value of the `Copyright` parameter in the *[Game]* section when the template is formatted.

Changed in version 8.0: `SkoolKit[page_header]` is a two-element list containing the page header prefix and suffix. Added `SkoolKit[include]`, `SkoolKit[javascripts]` and `SkoolKit[stylesheets]`.

Changed in version 6.4: Added `SkoolKit[path]`.

8.8.1 Layout

The `Layout` template is used to format every HTML page.

In any page defined by a *[Page:*]* section, the following identifier is available (in addition to the universal identifiers):

- `Page` - a dictionary of the parameters in the corresponding *[Page:*]* section

In any page defined by a *[MemoryMap:*]* section, the following identifier is available (in addition to the universal identifiers):

- `MemoryMap` - a dictionary of the parameters in the corresponding *[MemoryMap:*]* section

To see the default `Layout` template, run the following command:

```
$ skool2html.py -r Template:Layout
```

New in version 8.0.

8.8.2 asm

The `asm` template is used to format the content between the header and footer of a disassembly page.

The following identifiers are available (in addition to the universal identifiers):

- `entry` - an object representing the current memory map entry (see below)
- `next_entry` - an object representing the next memory map entry (see below)
- `prev_entry` - an object representing the previous memory map entry (see below)

The attributes in the `prev_entry` and `next_entry` objects are:

- `address` - the address of the entry (may be in decimal or hexadecimal format, depending on how it appears in the skool file, and the options passed to *skool2html.py*)
- `anchor` - the anchor for the entry, formatted according to the value of the `AddressAnchor` parameter in the *[Game]* section
- `byte` - the LSB of the entry address
- `description` - a list of paragraphs comprising the entry description
- `exists` - '1' if the entry exists, '0' otherwise
- `href` - the relative path to the disassembly page for the entry
- `label` - the ASM label of the first instruction in the entry
- `length` - the size of the entry in bytes, as a string formatted according to the value of the `Length` parameter in the *[Game]* section
- `location` - the address of the entry as a decimal number
- `map_href` - the relative path to the entry on the 'Memory Map' page
- `page` - the MSB of the entry address
- `size` - the size of the entry in bytes
- `title` - the title of the entry
- `type` - the block type of the entry ('b', 'c', 'g', 's', 't', 'u' or 'w')

The `entry` object also has these attributes, and the following additional ones:

- `annotated` - '1' if any instructions in the entry have a non-empty comment field, '0' otherwise
- `end_comment` - a list of paragraphs comprising the entry's end comment
- `input_registers` - a list of input register objects
- `instructions` - a list of instruction objects
- `labels` - '1' if any instructions in the entry have an ASM label, '0' otherwise
- `output_registers` - a list of output register objects
- `show_bytes` - '1' if the entry contains at least one assembled instruction with byte values and the `Bytes` parameter in the *[Game]* section is not blank, '0' otherwise

Each input and output register object has the following attributes:

- `description` - the register's description (as it appears in the register section for the entry in the skool file)
- `name` - the register's name (e.g. 'HL')

Each instruction object has the following attributes:

- `address` - the address of the instruction (may be in decimal or hexadecimal format, depending on how it appears in the skool file, and the options passed to *skool2html.py*)
- `anchor` - the anchor for the instruction, formatted according to the value of the `AddressAnchor` parameter in the *[Game]* section
- `block_comment` - a list of paragraphs comprising the instruction's mid-block comment
- `bytes` - the byte values of the assembled instruction (see below)
- `called` - '2' if the instruction is an entry point, '1' otherwise
- `comment` - the text of the instruction's comment field
- `comment_rowspan` - the number of instructions to which the comment field applies; this will be '0' if the instruction has no comment field
- `label` - the instruction's ASM label
- `location` - the address of the instruction as a decimal number
- `operation` - the assembly language operation (e.g. 'LD A,B'), with operand hyperlinked if appropriate

The `bytes` attribute can be used to render the byte values of an instruction. The format specifier for this attribute has the following form:

```
bfmt
```

or:

```
/bfmt/sep[/fmt]
```

- `bfmt` is the format specifier applied to each byte value
- `sep` is the separator string inserted between byte values; by default it is blank
- `fmt` is the format specifier applied to the entire string of byte values; by default it is blank

The delimiter used here (`/`) to separate the `bfmt`, `sep` and `fmt` parameters is arbitrary; it could be any character that doesn't appear in `bfmt` itself.

For example:

```
{$instruction[bytes]:02X}
```

would produce the string `3E01` for the instruction 'LD A,1'. And:

```
{$instruction[bytes]:/02X/ />11}
```

would render byte values as 2-digit upper case hexadecimal numbers separated by spaces, and right align the entire field to a width of 11 characters.

By default, the `Bytes` parameter in the *[Game]* section is used as the byte format specification:

```
{$instruction[bytes]:{Game[Bytes]}}
```

If you define a custom template that replaces `{Game[Bytes]}` with a hard-coded byte format specification, it's a good idea to also remove the `if({entry[show_bytes]})` directive (and the corresponding `endif`), to ensure that the byte values are displayed.

Note that byte values are available only for regular assembly language instructions (not DEFB, DEFM, DEFS or DEFW statements), and only if they have actually been assembled by using *@assemble=2*. When no byte values are available, or the format specifier is blank, the `bytes` identifier produces an empty string.

To see the default `asm` template, run the following command:

```
$ skool2html.py -r Template:asm$
```

Changed in version 8.4: Added the `length` attribute to entry objects.

Changed in version 8.1: Added the `fmt` parameter to the format specifier for the `bytes` attribute of instruction objects.

New in version 8.0.

8.8.3 asm_single_page

The `asm_single_page` template is used to format the content between the header and footer of a single-page disassembly.

The following identifier is available (in addition to the universal identifiers):

- `entries` - a list of memory map entry objects

The attributes of each memory map entry object are the same as those of the `entry` object in the *asm* template.

To see the default `asm_single_page` template, run the following command:

```
$ skool2html.py -r Template:asm_single_page
```

New in version 8.0.

8.8.4 audio

The `audio` template is used to format `<audio>` elements created by the *#AUDIO* macro.

The following identifier is available (in addition to the universal identifiers):

- `src` - the relative path to the audio file

To see the default `audio` template, run the following command:

```
$ skool2html.py -r Template:audio
```

New in version 8.7.

8.8.5 box_entries

The `box_entries` template is used to format the content between the header and footer of a *box page* with a default `SectionType`.

The following identifier is available (in addition to the universal identifiers):

- `entries` - a list of entry objects

Each entry object has the following attributes:

- `anchor` - the anchor for the entry
- `contents` - a list of paragraphs comprising the contents of the entry
- `order` - '1' or '2', depending on the order of the entry on the page
- `title` - the entry title

To see the default `box_entries` template, run the following command:

```
$ skool2html.py -r Template:box_entries
```

New in version 8.0.

8.8.6 box_list_entries

The `box_list_entries` template is used to format the content between the header and footer of a *box page* whose `SectionType` is `BulletPoints` or `ListItems`.

The following identifier is available (in addition to the universal identifiers):

- `entries` - a list of entry objects

Each entry object has the following attributes:

- `anchor` - the anchor for the entry
- `intro` - the entry intro text
- `item_list` - replaced by a copy of the *item_list* subtemplate
- `order` - '1' or '2', depending on the order of the entry on the page
- `title` - the entry title

To see the default `box_list_entries` template, run the following command:

```
$ skool2html.py -r Template:box_list_entries
```

New in version 8.0.

8.8.7 footer

The `footer` template is the subtemplate included in the *Layout* template to format the `<footer>` element of a page.

To see the default `footer` template, run the following command:

```
$ skool2html.py -r Template:footer
```

New in version 5.0.

8.8.8 home

The `home` template is used to format the content between the header and footer of the disassembly home page.

The following identifier is available (in addition to the universal identifiers):

- `sections` - a list of section objects

Each section object represents a group of links and has the following attributes:

- `header` - the header text for the group of links (as defined in the name of the *[Index:*:*]* section)
- `items` - a list of items in the group

Each item represents a link to a page and has the following attributes:

- `href` - the relative path to the page being linked to

- `link_text` - the link text for the page (as defined in the *[Links]* section)
- `other_text` - the supplementary text displayed alongside the link (as defined in the *[Links]* section)

To see the default `home` template, run the following command:

```
$ skool2html.py -r Template:home
```

New in version 8.0.

8.8.9 img

The `img` template is used to format `<img>` elements for the *image macros* and for the game logo image (if defined) in the header of every page.

The following identifiers are available (in addition to the universal identifiers):

- `alt` - the 'alt' text for the image
- `src` - the relative path to the image file

To see the default `img` template, run the following command:

```
$ skool2html.py -r Template:img
```

8.8.10 item_list

The `item_list` template is the subtemplate used by the *box_list_entries* template to format a list of items (or subitems, or subsubitems etc.) in an entry on a *box page* whose `SectionType` is `BulletPoints` or `ListItems`.

The following identifiers are available (in addition to the universal identifiers):

- `indent` - the indentation level of the item list: '' (blank string) for the list of top-level items, '1' for a list of subitems, '2' for a list of subsubitems etc.
- `items` - a list of item objects

Each item object has the following attributes:

- `subitems` - a preformatted list of subitems (may be blank)
- `text` - the text of the item

Note that the `item_list` template is used to format the `subitems` attribute of each item (this template is recursive).

To see the default `item_list` template, run the following command:

```
$ skool2html.py -r Template:item_list
```

New in version 8.0.

8.8.11 link

The `link` template is the subtemplate used to format the hyperlinks created by the *#LINK* and *#R* macros, and the hyperlinks in instruction operands on disassembly pages.

The following identifiers are available (in addition to the universal identifiers):

- `href` - the relative path to the page being linked to
- `link_text` - the link text for the page

To see the default `link` template, run the following command:

```
$ skool2html.py -r Template:link
```

8.8.12 list

The `list` template is used by the *#LIST* macro to format a list.

The following identifiers are available (in addition to the universal identifiers):

- `class` - the CSS class name for the list
- `items` - the list items

To see the default `list` template, run the following command:

```
$ skool2html.py -r Template:list
```

Changed in version 8.0: Replaced the `m_list_item` identifier with the `items` identifier.

New in version 4.2.

8.8.13 memory_map

The `memory_map` template is used to format the content between the header and footer of memory map pages and the 'Game status buffer' page.

The following identifier is available (in addition to the universal identifiers):

- `entries` - a list of memory map entry objects

The attributes of each memory map entry object are the same as those of the `prev_entry` and `next_entry` objects in the *asm* template.

To see the default `memory_map` template, run the following command:

```
$ skool2html.py -r Template:memory_map
```

New in version 8.0.

8.8.14 page

The `page` template is used to format the content between the header and footer of a non-box page defined by a *[Page:*]* section.

To see the default `page` template, run the following command:

```
$ skool2html.py -r Template:page
```

New in version 8.0.

8.8.15 reg

The `reg` template is the subtemplate used by the *#REG* macro to format a register name.

The following identifier is available (in addition to the universal identifiers):

- `reg` - the register name (e.g. 'HL')

To see the default `reg` template, run the following command:

```
$ skool2html.py -r Template:reg
```

8.8.16 section

The `section` template is used to format the paragraphs in a ref file section processed by the *#INCLUDE* macro.

The following identifier is available (in addition to the universal identifiers):

- `section` - a list of paragraphs

To see the default `section` template, run the following command:

```
$ skool2html.py -r Template:section
```

New in version 8.0.

8.8.17 table

The `table` template is used by the *#TABLE* macro to format a table.

The following identifiers are available (in addition to the universal identifiers):

- `class` - the CSS class name for the table
- `rows` - a list of row objects

Each row object has a `cells` attribute, which is a list of cell objects for that row. Each cell object has the following attributes:

- `class` - the CSS class name for the cell
- `colspan` - the number of columns spanned by the cell
- `contents` - the contents of the cell
- `header` - 1 if the cell is a header cell, 0 otherwise
- `rowspan` - the number of rows spanned by the cell

To see the default `table` template, run the following command:

```
$ skool2html.py -r Template:table
```

Changed in version 8.0: Replaced the `m_table_row` identifier with the `rows` identifier.

New in version 4.2.

8.8.18 Template directives

HTML templates may contain directives enclosed by `<#` and `#>` to conditionally include or repeat content. To take effect, a directive must appear on a line of its own.

foreach

The `foreach` directive repeats the content between it and the corresponding `endfor` directive, once for each object in a list.

```
<# foreach(var,list) #>
content
<# endfor #>
```

- `var` is the loop variable, representing each object in the list
- `list` is the list of objects to iterate over

Wherever the string `var` appears in `content`, it is replaced by `list[0]`, `list[1]`, etc. Care should be taken to name the loop variable such that no unwanted replacements are made.

For example, if `names` contains the strings 'Alice', 'Bob' and 'Carol', then:

```
<# foreach(name,names) #>
{name}
<# endfor #>
```

would produce the following output:

```
Alice
Bob
Carol
```

if

The `if` directive includes the content between it and the corresponding `else` directive (optional) or `endif` directive (required) if a given expression is true, and excludes it otherwise.

```
<# if(expr) #>
content
<# else #>
alternative content
<# endif #>
```

`expr` may be any syntactically valid Python expression, and may contain the names of any fields that are available in the template.

The `if` directive follows the same rules as Python when determining the truth of an expression: `None`, `False`, zero, and any empty string or collection is false; everything else is true.

Note that any replacement fields in `expr` are replaced with their string representations before the expression is evaluated. For example, if the value of the field 'val' is the string '0', then `val` evaluates to '0' (which is true, because it's a non-empty string); but `{val}` evaluates to 0 (which is false).

include

The `include` directive includes content from another template.

```
<# include(template) #>
```

`template` is the name of the template to include; it may contain replacement fields.

For example, if there is a template named `title` that contains `<title>{title}</title>`, and the `title` field holds the string 'My Page', then:

```
<head>
<# include(title) #>
</head>
```

would produce the following output:

```
<head>
<title>My Page</title>
</head>
```

8.8.19 Page-specific templates

When SkoolKit builds an HTML page, it uses the template whose name matches the page ID (`PageID`) if it exists, or the stock *Layout* template otherwise. For example, when building the `RoutinesMap` memory map page, SkoolKit will use the `RoutinesMap` template if it exists.

Wherever `Asm-*` appears in the tables below, it means one of `Asm-b`, `Asm-c`, `Asm-g`, `Asm-s`, `Asm-t`, `Asm-u` or `Asm-w`, depending on the type of memory map entry.

Page type	Preferred template(s)
Home (index)	`GameIndex`
Other code index	`CodeID-Index`
Routine/data block	`Asm-*`, `Asm`
Other code routine/data block	`CodeID-Asm-*`, `CodeID-Asm`
Disassembly (single page)	`AsmSinglePage`
Other code disassembly (single page)	`CodeID-AsmSinglePage`
Memory map	`PageID`
Box page	`PageID`
Custom page (non-box)	`PageID`

When SkoolKit builds the content of an HTML page between the page header and footer, it uses the subtemplate whose name starts with `PageID-` if it exists, or the appropriate stock subtemplate otherwise. For example, when building the entries on the `Changelog` page, SkoolKit uses the `Changelog-box_list_entries` template if it exists, or the stock *box_list_entries* template otherwise.

Page type	Preferred template(s)	Stock template
Routine/data block	`Asm-*-asm`, `Asm-asm`	*asm*
Other code routine/data block	`CodeID-Asm-*-asm`, `CodeID-Asm-asm`	*asm*
Disassembly (single page)	`AsmSinglePage-asm_single_page`	*asm_single_page*
Other code disassembly (single page)	`CodeID-AsmSinglePage-asm_single_page`	*asm_single_page*
Box page with regular entries	`PageID-box_entries`	*box_entries*
Box page with list entries	`PageID-box_list_entries`	*box_list_entries*
Home (index)	`GameIndex-home`	*home*
Memory map	`PageID-memory_map`	*memory_map*
Other code index	`CodeID-Index-memory_map`	*memory_map*
Custom page (non-box)	`PageID-page`	*page*

When SkoolKit builds an element of an HTML page whose format is defined by a subtemplate, it uses the subtemplate whose name starts with `PageID-` if it exists, or one of the stock subtemplates otherwise. For example, when building the footer of the `Changelog` page, SkoolKit uses the `Changelog-footer` template if it exists, or the stock *footer* template otherwise.

Element type	Preferred template	Stock template
`<audio>` element	`PageID-audio`	*audio*
Page footer	`PageID-footer`	*footer*
`<img>` element	`PageID-img`	*img*
Box page list entry	`PageID-item_list`	*item_list*
Hyperlink	`PageID-link`	*link*
List created by the *#LIST* macro	`PageID-list`	*list*
Register name rendered by the *#REG* macro	`PageID-reg`	*reg*
Table created by the *#TABLE* macro	`PageID-table`	*table*

CHAPTER

NINE

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 or defining a new sprite-building macro with the *#DEFINE* macro would be too tedious or impractical. Or you might want to insert a timestamp somewhere in 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 writer and ASM writer 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:

```
from skoolkit.skoolhtml import HtmlWriter
from skoolkit.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/python3/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 named *game.py* in *~/.skoolkit*, you should set the `HtmlWriterClass` parameter thus:

```
HtmlWriterClass=~/.skoolkit:game.GameHtmlWriter
```

and the `@writer` directive thus:

```
@writer=~/.skoolkit:game.GameAsmWriter
```

The HTML writer or ASM writer class can also be specified on the command line by using the `-W`/`--writer` option of *skool2html.py* or *skool2asm.py*. For example:

```
$ skool2html.py -W ~/.skoolkit:game.GameHtmlWriter game.skool
```

Specifying the writer class this way will override any `HtmlWriterClass` parameter in the ref file or `@writer` directive in the skool file.

Note that if the writer class is specified with a blank module path (e.g. `:game.GameHtmlWriter`), SkoolKit will search for the module in both the current working directory and the directory containing the skool file named on the command line.

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 skoolkit.graphics import Frame
from skoolkit.skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def sprite(self, cwd, sprite_id, fname):
        udgs = self.build_sprite(sprite_id)
        return self.handle_image(Frame(udgs), fname, cwd)
```

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 skoolkit.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()
```

Note that if the return value of a #CALL method contains skool macros, then they will be expanded.

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*)
- 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 will be expanded. Note that if string itself contains skool macros, then they will be expanded.

The *expand_sprite* method on GameHtmlWriter may therefore look something like this:

```
from skoolkit.graphics import Frame
from skoolkit.skoolhtml import HtmlWriter
from skoolkit.skoolmacro import parse_image_macro

class GameHtmlWriter(HtmlWriter):
    # #SPRITEid[{x,y,width,height}](fname)
    def expand_sprite(self, text, index, cwd):
```

(continues on next page)

(continued from previous page)

```
        end, crop_rect, fname, frame, alt, (sprite_id,) = parse_image_macro(text,␣
↪index, names=['id'])
        udgs = self.build_sprite(sprite_id)
        frame = Frame(udgs, 2, 0, *crop_rect, name=frame)
        return end, self.handle_image(frame, fname, cwd, alt)
```

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 skoolkit.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=0*, *num=0*, *defaults=()*, *names=()*, *fields=None*)

Parse a sequence of comma-separated integer parameters, optionally enclosed in parentheses. If parentheses are used, the parameters may be expressed using arithmetic operators and skool macros. See *Numeric parameters* for more details.

Parameters

- **`text`** – The text to parse.
- **`index`** – The index at which to start parsing.
- **`num`** – The maximum number of parameters to parse; this is set to the number of elements in *names* if that list is not empty.
- **`defaults`** – The default values of the optional parameters.
- **`names`** – The names of the parameters; if not empty, keyword arguments are parsed. Parameter names are restricted to lower case letters (a-z).
- **`fields`** – A dictionary of replacement field names and values. The fields named in this dictionary are replaced by their values wherever they appear in the parameter string.

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

Changed in version 6.0: Added the *fields* parameter.

Changed in version 5.1: Added support for parameters expressed using arithmetic operators and skool macros.

Changed in version 4.0: Added the *names* parameter and support for keyword arguments; *index* defaults to 0.

skoolkit.skoolmacro.**parse_strings**(*text*, *index=0*, *num=0*, *defaults=()*)

Parse a sequence of comma-separated string parameters. The sequence must be enclosed in parentheses, square brackets or braces. If the sequence itself contains commas or unmatched brackets, then an alternative delimiter and separator may be used; see *String parameters* for more details.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **num** – The maximum number of parameters to parse. If 0, all parameters are parsed; if 1, the entire parameter string is parsed as a single parameter, regardless of commas.
- **defaults** – The default values of the optional parameters.

Returns

A tuple of the form `(end, result)`, where:

- `end` is the index at which parsing terminated
- `result` is either the single parameter itself (when *num* is 1), or a list of the parameters

New in version 5.1.

skoolkit.skoolmacro.**parse_brackets**(*text*, *index=0*, *default=None*, *opening='('*, *closing=')'*)

Parse a single string parameter enclosed either in parentheses or by an arbitrary pair of delimiters.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **default** – The default value if no string parameter is found.
- **opening** – The opening delimiter.
- **closing** – The closing delimiter.

Returns

A tuple of the form `(end, param)`, where:

- `end` is the index at which parsing terminated
- `param` is the string parameter (or *default* if none is found)

New in version 5.1.

skoolkit.skoolmacro.**parse_image_macro**(*text*, *index=0*, *defaults=()*, *names=()*, *fname=''*, *fields=None*)

Parse a string of the form:

```
[params][{x,y,width,height}][(fname[*frame][|alt])]
```

The parameter string `params` may contain comma-separated integer values, and may optionally be enclosed in parentheses. Parentheses are *required* if any parameter is expressed using arithmetic operations or skool macros.

Parameters

- **text** – The text to parse.

- **index** – The index at which to start parsing.
- **defaults** – The default values of the optional parameters.
- **names** – The names of the parameters.
- **fname** – The default base name of the image file.
- **fields** – A dictionary of replacement field names and values. The fields named in this dictionary are replaced by their values wherever they appear in `params` or `{x,y,width,height}`.

Returns

A tuple of the form `(end, crop_rect, fname, frame, alt, values)`, where:

- `end` is the index at which parsing terminated
- `crop_rect` is `(x, y, width, height)`
- `fname` is the base name of the image file
- `frame` is the frame name (*None* if no frame is specified)
- `alt` is the alt text (*None* if no alt text is specified)
- `values` is a list of the parameter values

Changed in version 8.3: Added the *fields* parameter.

New in version 5.1.

9.1.5 Expanding skool macros

Both AsmWriter and HtmlWriter provide methods for expanding skool macros. These are useful for immediately expanding macros in a `#CALL` method or custom macro method.

`AsmWriter.`**`expand`**(*text*)
Return *text* with skool macros expanded.

`HtmlWriter.`**`expand`**(*text*, *cwd=None*)
Return *text* with skool macros expanded. *cwd* is the current working directory, which is required by macros that create images or hyperlinks.

Changed in version 5.1: The *cwd* parameter is optional.

9.1.6 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*, *trim=True*)
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.
- **trim** – If *True*, remove leading whitespace from each line.

Changed in version 5.3: Added the *trim* parameter.

`HtmlWriter.`**`get_sections`**(*section_type*, *paragraphs=False*, *lines=False*, *trim=True*)
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.
- **`trim`** – If *True*, remove leading whitespace from each line.

Changed in version 5.3: Added the *trim* parameter.

`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`.

9.1.7 Formatting templates

HtmlWriter provides a method for formatting a template defined by a *[Template:*]* section.

`HtmlWriter.`**`format_template`**(*name*, *fields*)
Format a template with a set of replacement fields.

Parameters

- **`name`** – The name of the template.
- **`fields`** – A dictionary of replacement field names and values.

Returns The formatted string.

Changed in version 8.0: Removed the *default* parameter.

New in version 4.0.

Note that if *name* is 'Layout', the template whose name matches the current page ID will be used, if it exists; if no such template exists, the *Layout* template will be used. If *name* is not 'Layout', the template named `PageID-name` (where `PageID` is the current page ID) will be used, if it exists; if no such template exists, the `name` template will be used. This is in accordance with SkoolKit's rules for preferring *page-specific templates*.

9.1.8 Base, case and fields

The *base* and *case* attributes on AsmWriter and HtmlWriter can be inspected to determine the mode in which *skool2asm.py* or *skool2html.py* is running.

The *base* attribute has one of the following values:

- 0 - default (neither `--decimal` nor `--hex`)
- 10 - decimal (`--decimal`)
- 16 - hexadecimal (`--hex`)

New in version 6.1.

The *case* attribute has one of the following values:

- 0 - default (neither `--lower` nor `--upper`)
- 1 - lower case (`--lower`)
- 2 - upper case (`--upper`)

New in version 6.1.

The *fields* attribute on AsmWriter and HtmlWriter is a dictionary of replacement field names and values (see *Replacement fields*). It can be used with the *parse_ints()* and *parse_image_macro()* functions.

New in version 6.0.

9.1.9 Memory snapshots

The *snapshot* attribute on HtmlWriter and AsmWriter is a 65536-element list that represents the 64K of the Spectrum's memory; it is populated when the skool file is being parsed.

HtmlWriter and AsmWriter also provide methods for saving and restoring memory snapshots, which can be useful for temporarily changing graphic data or the contents of data tables.

`HtmlWriter.`**`push_snapshot`**`(`*name=''*`)`
Save a copy of the current memory snapshot for later retrieval (by *pop_snapshot()*).

Parameters **name** – An optional name for the snapshot.

`HtmlWriter.`**`pop_snapshot`**`()`
Replace the current memory snapshot with the one most recently saved by *push_snapshot()*.

In addition, HtmlWriter (but not AsmWriter) provides a method for retrieving the snapshot name.

`HtmlWriter.`**`get_snapshot_name`**`()`
Return the name of the current memory snapshot.

9.1.10 Graphics

If you are going to implement a custom image-creating `#CALL` method or skool macro, you will need to make use of the skoolkit.graphics.Udg and skoolkit.graphics.Frame classes.

The Udg class represents an 8x8 graphic (8 bytes) with a single attribute byte, and an optional mask.

`class` `skoolkit.graphics.`**`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).

Changed in version 5.4: The Udg class moved from skoolkit.skoolhtml to skoolkit.graphics.

An `#INVERSE` macro that creates an inverse image of a UDG with scale 2 might be implemented like this:

```
from skoolkit.graphics import Frame, Udg
from skoolkit.skoolhtml import HtmlWriter
from skoolkit.skoolmacro import parse_ints

class GameHtmlWriter(HtmlWriter):
    # #INVERSEaddress,attr
    def expand_inverse(self, text, index, cwd):
        end, address, attr = parse_ints(text, index, 2)
        udg_data = [b ^ 255 for b in self.snapshot[address:address + 8]]
        frame = Frame([[Udg(attr, udg_data)]], 2)
        fname = 'inverse{}_{}'.format(address, attr)
        return end, self.handle_image(frame, fname, cwd)
```

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.

The Udg class also provides a method for creating a copy of a UDG.

Udg.**copy**()
Return a deep copy of the UDG.

The Frame class represents a single frame of a still or animated image.

class skoolkit.graphics.**Frame**(*udgs*, *scale=1*, *mask=0*, *x=0*, *y=0*, *width=None*, *height=None*, *delay=32*, *name=''*, *tindex=0*, *alpha=- 1*, *x_offset=0*, *y_offset=0*)
Create a frame of a still or animated image.

Parameters

- **udgs** – The two-dimensional array of tiles (instances of *Udg*) from which to build the frame, or a function that returns the array of tiles.
- **scale** – The scale of the frame.
- **mask** – The type of mask to apply to the tiles in the frame: 0 (no mask), 1 (OR-AND mask), or 2 (AND-OR mask).
- **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 the width of the array of tiles) is used.
- **height** – The height of the frame; if *None*, the maximum height (derived from *y* and the height of the array of tiles) is used.

- **delay** – The delay between this frame and the next in 1/100ths of a second.
- **name** – The name of this frame.
- **tindex** – The index of the entry in the *palette* to use as the transparent colour.
- **alpha** – The alpha value to use for the transparent colour. If -1, the value of the `PNGAlpha` parameter in the *[ImageWriter]* section is used.
- **x_offset** – The x-coordinate at which to render the frame.
- **y_offset** – The y-coordinate at which to render the frame.

Changed in version 8.3: Added the *x_offset* and *y_offset* parameters.

Changed in version 8.2: Added the *tindex* and *alpha* parameters.

Changed in version 5.4: The Frame class moved from skoolkit.skoolhtml to skoolkit.graphics.

Changed in version 5.1: The *udgs* parameter can be a function that returns the array of tiles; added the *name* parameter.

Changed in version 4.0: The *mask* parameter specifies the type of mask to apply (see *Masks*).

New in version 3.6.

HtmlWriter and skoolkit.graphics provide the following image-related methods and functions.

`HtmlWriter.`**handle_image**(*frames*, *fname=''*, *cwd=None*, *alt=None*, *path_id='ImagePath'*)

Register a named frame for an image, and write an image file if required. If *fname* is blank, no image file will be created. If *fname* does not end with '.png', that suffix will be appended. If *fname* contains an image path ID replacement field, the corresponding parameter value from the *[Paths]* section will be substituted.

Parameters

- **frames** – A frame (instance of *Frame*) or list of frames from which to build the image.
- **fname** – The name of the image file.
- **cwd** – The current working directory (from which the relative path of the image file will be computed).
- **alt** – The alt text to use for the image.
- **path_id** – The ID of the target directory (as defined in the *[Paths]* section of the ref file). This is not used if *fname* starts with a '/' or contains an image path ID replacement field.

Returns The `<img ... />` element, or an empty string if no image is created.

Changed in version 7.0: *path_id* defaults to `ImagePath` (previously `UDGImagePath`).

Changed in version 6.4: *frames* may be a single frame.

Changed in version 6.3: *fname* may contain an image path ID replacement field (e.g. `{UDGImagePath}`).

New in version 5.1.

`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.

skoolkit.graphics.**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.

skoolkit.graphics.**overlay_udgs**(*bg*, *fg*, *x*, *y*, *mask=0*, *rattr=None*, *rbyte=None*)
Overlay a foreground array of UDGs (instances of *Udg*) on a background array of UDGs.

Parameters

- **bg** – The background array of UDGs.
- **fg** – The foreground array of UDGs.
- **x** – Pixel x-coordinate on the background at which to overlay the foreground.
- **y** – Pixel y-coordinate on the background at which to overlay the foreground.
- **mask** – The type of mask to apply: 0 (no mask), 1 (OR-AND mask), or 2 (AND-OR mask). If 0, the contents of the foreground and background are combined by OR operations.
- **rattr** – A function that returns the replacement attribute byte for a background UDG over which a foreground UDG is superimposed. It must accept two arguments: the existing background UDG attribute, and the foreground UDG attribute. If *None*, the existing background attributes are left in place.
- **rbyte** – A function that returns the replacement value for each graphic byte of a background UDG over which a foreground UDG is superimposed. It must accept three arguments: the existing background UDG graphic byte, the foreground UDG graphic byte, and the foreground UDG mask byte. If *None*, the mask specified by *mask* is used.

New in version 8.5.

skoolkit.graphics.**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.

9.1.11 HTML page initialisation

If you need to perform page-specific actions or customise the `SkoolKit` and `Game` parameter dictionaries that are used by the *HTML templates*, the place to do that is the *init_page()* method.

HtmlWriter.**init_page**(*skoolkit*, *game*)
Perform page initialisation operations. This method is called after the `SkoolKit` and `Game` parameter dictionaries have been initialised, and provides those dictionaries as arguments for inspection and customisation before a page is formatted. By default the method does nothing, but subclasses may override it.

Parameters

- **skoolkit** – The `SkoolKit` parameter dictionary.

- **game** – The `Game` parameter dictionary.

New in version 7.0.

9.1.12 Writer initialisation

If your AsmWriter or 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.

`AsmWriter.`**`init`**`()`
Perform post-initialisation operations. This method is called after *__init__()* has completed. By default the method does nothing, but subclasses may override it.

New in version 6.1.

`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 skoolkit.skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def init(self):
        # Get character names from the ref file
        self.characters = self.get_dictionary('Characters')
```

9.2 SkoolKit components

SkoolKit relies on several components in order to function:

- *Assembler*
- *Control directive composer*
- *Control file generator*
- *Disassembler*
- *HTML template formatter*
- *Image writer*
- *Instruction utility*
- *Operand evaluator*
- *Operand formatter*
- *Snapshot reader*
- *Snapshot reference calculator*

The objects that are used for these components can be specified in the *[skoolkit]* section of *skoolkit.ini*.

9.2.1 [skoolkit]

Global configuration for SkoolKit can be specified in the `[skoolkit]` section of a file named *skoolkit.ini* either in the current working directory or in *~/.skoolkit*. The default contents of this section are as follows:

```
[skoolkit]
Assembler=skoolkit.z80.Assembler
ControlDirectiveComposer=skoolkit.skoolctl.ControlDirectiveComposer
ControlFileGenerator=skoolkit.snactl
DefaultDisassemblyStartAddress=16384
Disassembler=skoolkit.disassembler.Disassembler
HtmlTemplateFormatter=skoolkit.skoolhtml.TemplateFormatter
ImageWriter=skoolkit.image.ImageWriter
InstructionUtility=skoolkit.skoolparser.InstructionUtility
OperandEvaluator=skoolkit.z80
OperandFormatter=skoolkit.disassembler.OperandFormatter
SnapshotReader=skoolkit.snapshot
SnapshotReferenceCalculator=skoolkit.snaskool
SnapshotReferenceOperations=DJ,JR,JP,CA,RS
```

Most of the parameters in the `[skoolkit]` section specify the objects to use for SkoolKit's pluggable components. The other recognised parameters are:

- `DefaultDisassemblyStartAddress` - the address at which to start disassembling a snapshot when no control file is provided; this is used by *sna2ctl.py* and *sna2skool.py*, and also by *snapinfo.py* when generating a call graph
- `SnapshotReferenceOperations` - the instructions whose address operands are used by the *snapshot reference calculator* to identify entry points in routines and data blocks

9.2.2 Assembler

This object is responsible for converting assembly language instructions and DEFB/DEFM/DEFS/DEFW statements into byte values, or computing their size. It must supply the following API functions, in common with skoolkit.z80.Assembler:

class skoolkit.z80.**Assembler**

assemble(*operation*, *address*)
Convert an assembly language instruction or DEFB/DEFM/DEFS/DEFW statement into a sequence of byte values.

Parameters

- **operation** – The operation to convert (e.g. 'XOR A').
- **address** – The instruction address.

Returns A sequence of byte values (empty if the instruction cannot be assembled).

get_size(*operation*, *address*)
Compute the size (in bytes) of an assembly language instruction or DEFB/DEFM/DEFS/DEFW statement.

Parameters

- **operation** – The operation (e.g. 'XOR A').
- **address** – The instruction address.

Returns The instruction size, or 0 if the instruction cannot be assembled.

9.2.3 Control directive composer

This class is responsible for computing the type, length and sublengths of a DEFB/DEFM/DEFS/DEFW statement, or the operand bases of a regular instruction, for the purpose of composing a control directive. It must supply the following API methods, in common with skoolkit.skoolctl.ControlDirectiveComposer:

class skoolkit.skoolctl.**ControlDirectiveComposer**(*preserve_base*)
Initialise the control directive composer.

Parameters **preserve_base** – Whether to preserve the base of decimal and hexadecimal values with explicit 'd' and 'h' base indicators.

compose(*operation*)
Compute the type, length and sublengths of a DEFB/DEFM/DEFS/DEFW statement, or the operand bases of a regular instruction.

Parameters **operation** – The operation (e.g. 'LD A,0' or 'DEFB 0').

Returns

A 3-element tuple, `(ctl, length, sublengths)`, where:

- `ctl` is 'B' (DEFB), 'C' (regular instruction), 'S' (DEFS), 'T' (DEFM) or 'W' (DEFW)
- `length` is the number of bytes in the DEFB/DEFM/DEFS/DEFW statement, or the operand base indicator for a regular instruction (e.g. 'b' for 'LD A,%00000001')
- `sublengths` is a colon-separated sequence of sublengths (e.g. '1:c1' for 'DEFB 0,"a"'), or *None* for a regular instruction

If **compose()** encounters an error while parsing an operation and cannot recover, it should raise a SkoolParsingError:

class skoolkit.**SkoolParsingError**
Raised when an error occurs while parsing a skool file.

9.2.4 Control file generator

This object is reponsible for generating a dictionary of control directives from a snapshot. Each key in the dictionary is an address, and the associated value is the control directive (e.g. 'b' or 'c') for that address. The control file generator object must supply the following API function, in common with skoolkit.snactl:

skoolkit.snactl.**generate_ctls**(*snapshot*, *start*, *end*, *code_map*, *config*)
Generate control directives from a snapshot.

Parameters

- **snapshot** – The snapshot.
- **start** – Start address. No control directives should be generated before this address.
- **end** – End address. No control directives should be generated after this address.
- **code_map** – Code map filename (may be *None*).
- **config** – Configuration object with the following attributes:
  - *text_chars* - string of characters eligible for being marked as text
  - *text_min_length_code* - minimum length of a string of characters eligible for being marked as text in a block identified as code
  - *text_min_length_data* - minimum length of a string of characters eligible for being marked as text in a block identified as data

- *words* - collection of allowed words; if not empty, a string of characters should be marked as text only if it contains at least one of the words in this collection

Returns A dictionary of control directives.

9.2.5 Disassembler

This class is responsible for converting byte values into assembly language instructions and DEFB/DEFM/DEFS/DEFW statements. It must supply the following API methods, in common with skoolkit.disassembler.Disassembler:

class skoolkit.disassembler.**Disassembler**(*snapshot*, *config*)

Initialise the disassembler.

Parameters

- **snapshot** – The snapshot (list of 65536 byte values) to disassemble.
- **config** – Configuration object with the following attributes:
  - *asm_hex* - if *True*, produce a hexadecimal disassembly
  - *asm_lower* - if *True*, produce a lower case disassembly
  - *defb_size* - default maximum number of bytes in a DEFB statement
  - *defm_size* - default maximum number of characters in a DEFM statement
  - *defw_size* - default maximum number of words in a DEFW statement
  - *wrap* - if *True*, disassemble an instruction that wraps around the 64K boundary

defb_range(*start*, *end*, *sublengths*)

Produce a sequence of DEFB statements for an address range.

Parameters

- **start** – The start address.
- **end** – The end address.
- **sublengths** – Sequence of sublength identifiers.

Returns A list of tuples of the form `(address, operation, bytes)`.

defm_range(*start*, *end*, *sublengths*)

Produce a sequence of DEFM statements for an address range.

Parameters

- **start** – The start address.
- **end** – The end address.
- **sublengths** – Sequence of sublength identifiers.

Returns A list of tuples of the form `(address, operation, bytes)`.

defs_range(*start*, *end*, *sublengths*)

Produce a sequence of DEFS statements for an address range.

Parameters

- **start** – The start address.
- **end** – The end address.

- **sublengths** – Sequence of sublength identifiers.

Returns A list of tuples of the form `(address, operation, bytes)`.

defw_range(*start*, *end*, *sublengths*)
Produce a sequence of DEFW statements for an address range.

Parameters

- **start** – The start address.
- **end** – The end address.
- **sublengths** – Sequence of sublength identifiers.

Returns A list of tuples of the form `(address, operation, bytes)`.

disassemble(*start*, *end*, *base*)
Disassemble an address range.

Parameters

- **start** – The start address.
- **end** – The end address.
- **base** – Base indicator ('b', 'c', 'd', 'h', 'm' or 'n'). For instructions with two numeric operands (e.g. 'LD (IX+d),n'), the indicator may consist of two letters, one for each operand (e.g. 'dh').

Returns A list of tuples of the form `(address, operation, bytes)`.

The 3-element tuples returned by these methods should have the form `(address, operation, bytes)`, where:

- `address` is the address of the instruction
- `operation` is the operation (e.g. 'XOR A', 'DEFB 1')
- `bytes` is a sequence of byte values for the instruction (e.g. `(62, 0)` for 'LD A,0')

The *sublengths* argument of the `defb_range()`, `defm_range()`, `defs_range()` and `defw_range()` methods is a sequence of 2-element tuples of the form `(size, base)`, each of which specifies the desired size (in bytes) and number base for an item in the DEFB/DEFM/DEFS/DEFW statement. `base` may have one of the following values:

- 'b' - binary
- 'c' - character
- 'd' - decimal
- 'h' - hexadecimal
- 'm' - negative
- 'n' - default base

If the first element of *sublengths* has a `size` value of 0, then the method should produce a list of statements with default sizes (as determined by *defb_size*, *defm_size* and *defw_size*), using the specified base.

Changed in version 8.5: Added the ability to disassemble an instruction that wraps around the 64K boundary, along with the *wrap* attribute on the disassembler configuration object to control this behaviour.

9.2.6 HTML template formatter

This class is responsible for formatting HTML templates. It must supply the following API methods, in common with skoolkit.skoolhtml.TemplateFormatter:

class skoolkit.skoolhtml.**TemplateFormatter**(*templates*)

Initialise the template formatter.

Parameters **templates** – A dictionary of templates keyed by template name.

format_template(*page_id*, *name*, *fields*)

Format a template.

Parameters

- **page_id** – The ID of the current page.
- **name** – The template name.
- **fields** – A dictionary of replacement field values.

Returns The text of the formatted template.

9.2.7 Image writer

This class is responsible for constructing images and writing them to files. It must supply the following API methods, in common with skoolkit.image.ImageWriter:

class skoolkit.image.**ImageWriter**(*config=None*, *palette=None*)

Initialise the image writer.

Parameters

- **config** – A dictionary constructed from the contents of the *[ImageWriter]* section of the ref file.
- **palette** – A dictionary constructed from the contents of the *[Colours]* section of the ref file. Each key is a colour name, and each value is a three-element tuple representing an RGB triplet.

If *config* or *palette* is *None*, empty, or missing values, default values are used.

image_fname(*fname*)

Convert the *fname* parameter of an image macro into an image filename with an appropriate extension.

Parameters **fname** – The *fname* parameter of the image macro.

Returns The image filename.

write_image(*frames*, *img_file*)

Write an image file. If this method leaves the image file empty, the file will be removed.

Parameters

- **frames** – A list of *Frame* objects from which to build the image.
- **img_file** – The file object to write the image to.

Returns The content with which the image macro is replaced; if *None*, an appropriate element is used.

9.2.8 Instruction utility

This object is responsible for performing various operations on the instructions in a skool file:

- converting base and case
- replacing addresses with labels (or other addresses) in instruction operands; this is required both for ASM output and for binary output
- generating a dictionary of references (for each instruction that refers to another instruction); this is required for hyperlinking instruction operands in HTML output
- generating a dictionary of referrers (for each instruction that is referred to by other instructions); this is required by the special `EREF` and `REF` variables of the *#FOREACH* macro
- deciding whether to set byte values; this affects the *#PEEK* macro and the *image macros*, and instruction byte values in HTML output

The object must supply the following API functions, in common with skoolkit.skoolparser.InstructionUtility:

class skoolkit.skoolparser.**InstructionUtility**

calculate_references(*entries*, *remote_entries*)
Generate a dictionary of references (for each instruction that refers to another instruction) and a dictionary of referrers (for each instruction that is referred to by other instructions) from the instructions in a skool file.

Parameters

- **entries** – A collection of memory map entries.
- **remote_entries** – A collection of remote entries (as defined by *@remote* directives).

Returns A tuple containing the two dictionaries.

convert(*entries*, *base*, *case*)
Convert the base and case of every instruction in a skool file.

Parameters

- **entries** – A collection of memory map entries.
- **base** – The base to convert to: 0 for no conversion, 10 for decimal, or 16 for hexadecimal.
- **case** – The case to convert to: 0 for no conversion, 1 for lower case, or 2 for upper case.

set_byte_values(*instruction*, *assemble*)
Decide whether to set byte values in the memory snapshot and for an instruction.

If byte values are set in the memory snapshot, then they are available to the *#PEEK* macro and the *image macros*. If byte values are set for an instruction, then they are available for display in HTML output via the `instruction[bytes]` replacement field in the *asm* template.

Parameters

- **instruction** – The instruction.
- **assemble** – The current value of the *assemble* property (as set by the *@assemble* directive).

Returns 2 if both the snapshot and the instruction should have byte values defined, 1 if only the snapshot should, or 0 if neither should.

substitute_labels(*entries*, *remote_entries*, *labels*, *mode*, *warn*)
Replace addresses with labels in the operands of every instruction in a skool file.

Parameters

- **entries** – A collection of memory map entries.
- **remote_entries** – A collection of remote entries (as defined by *@remote* directives).
- **labels** – A dictionary mapping addresses to labels.
- **mode** – The substitution mode: 1 (`@isub`), 2 (`@ssub`), 3 (`@rsub`), or 0 (none).
- **warn** – A function to be called if a warning is generated when attempting to replace an address in an instruction operand with a label. The function must accept two arguments:
  - *message* - the warning message.
  - *instruction* - the instruction object.

Memory map entries and remote entries have the following attributes:

- *ctl* - the entry's control directive ('b', 'c', 'g', 'i', 's', 't', 'u' or 'w' for a memory map entry; *None* for a remote entry)
- *instructions* - a collection of instruction objects

Each instruction object has the following attributes:

- *address* - the address of the instruction as stated in the skool file; note that this will not be the same as the actual address of the instruction if it has been moved by the insertion, removal or replacement of other instructions by `@*sub` or `@*fix` directives
- *keep* - *None* if the instruction has no *@keep* directive; an empty collection if it has a bare *@keep* directive; or a collection of addresses if it has a *@keep* directive with one or more values
- *nowarn* - *None* if the instruction has no *@nowarn* directive; an empty collection if it has a bare *@nowarn* directive; or a collection of addresses if it has a *@nowarn* directive with one or more values
- *operation* - the operation (e.g. 'XOR A') after any `@*sub` or `@*fix` directives have been applied; for an instruction in a remote entry, this is an empty string
- *refs* - the addresses of the instruction's indirect referrers, as declared by a *@refs* directive
- *rrefs* - the addresses of the instruction's direct referrers to be removed, as declared by a *@refs* directive
- *sub* - *True* if the operation was supplied by `@*sub` or `@*fix` directive, *False* otherwise

Each key in the references dictionary should be an instruction object, and the corresponding value should be a 3-element tuple:

```
(ref_instruction, address_s, use_label)
```

- `ref_instruction` - the instruction referred to
- `address_s` - the address string in the operand of the referring instruction (to be replaced by a hyperlink in HTML output)
- `use_label` - whether to use a label as the link text for the hyperlink in HTML output; if no label for `ref_instruction` is defined, or `use_label` is *False*, the address string (`address_s`) will be used as the link text

Each key in the referrers dictionary should be an instruction object, and the corresponding value should be a collection of the entries that refer to that instruction.

Changed in version 8.2: Added the *refs* and *rrefs* attributes to instruction objects.

Changed in version 8.1: Added the *mode* parameter to the `substitute_labels()` method, and changed the required signature of the *warn* function. Added the *nowarn* and *sub* attributes to instruction objects.

9.2.9 Operand evaluator

This object is used by the *assembler* to evaluate instruction operands, and by the *control directive composer* to determine the length and sublengths of DEFB, DEFM and DEFS statements. It must supply the following API functions, in common with skoolkit.z80:

`skoolkit.z80.`**`eval_int`**(*text*)
Evaluate an integer operand.

Parameters **`text`** – The operand.

Returns The integer value.

Raises *ValueError* if the operand is not a valid integer.

`skoolkit.z80.`**`eval_string`**(*text*)
Evaluate a string operand.

Parameters **`text`** – The operand, including enclosing quotes.

Returns A list of byte values.

Raises *ValueError* if the operand is not a valid string.

`skoolkit.z80.`**`split_operands`**(*text*)
Split a comma-separated list of operands.

Parameters **`text`** – The operands.

Returns A list of individual operands.

9.2.10 Operand formatter

This class is used by the *disassembler* to format numeric instruction operands. It must supply the following API methods, in common with skoolkit.disassembler.OperandFormatter:

`class` `skoolkit.disassembler.`**`OperandFormatter`**(*config*)
Initialise the operand formatter.

Parameters **`config`** – Configuration object with the following attributes:

- *asm_hex* - if *True*, default base is hexadecimal
- *asm_lower* - if *True*, format operands in lower case

`format_byte`(*value*, *base*)
Format a byte value.

Parameters

- **`value`** – The byte value.
- **`base`** – The desired base ('b', 'c', 'd', 'h', 'm' or 'n').

Returns The formatted byte value.

`format_word`(*value*, *base*)
Format a word (2-byte) value.

Parameters

- **`value`** – The word value.
- **`base`** – The desired base ('b', 'c', 'd', 'h', 'm' or 'n').

Returns The formatted word value.

is_char(*value*)
Return whether a byte value can be formatted as a character.

Parameters **value** – The byte value.

9.2.11 Snapshot reader

This object is responsible for producing a 65536-element list of byte values from a snapshot file. It must supply the following API functions, in common with skoolkit.snapshot:

skoolkit.snapshot.**can_read**(*fname*)
Return whether this snapshot reader can read the file *fname*.

skoolkit.snapshot.**get_snapshot**(*fname*, *page=None*)
Read a snapshot file and produce a 65536-element list of byte values.

Parameters

- **fname** – The snapshot filename.
- **page** – The page number to map to addresses 49152-65535 (C000-FFFF). This is relevant only when reading a 128K snapshot file.

Returns A 65536-element list of byte values.

If **get_snapshot()** encounters an error while reading a snapshot file, it should raise a SnapshotError:

class skoolkit.snapshot.**SnapshotError**
Raised when an error occurs while reading a snapshot file.

9.2.12 Snapshot reference calculator

This object is responsible for generating a dictionary of entry point addresses from a snapshot. Each key in the dictionary is an entry point address, and the associated value is a collection of entries that jump to, call or otherwise refer to that entry point. This dictionary is needed by *sna2skool.py* for marking each entry point in a skool file with an asterisk, and listing its referrers.

The snapshot reference calculator must supply the following API function, in common with skoolkit.snaskool:

skoolkit.snaskool.**calculate_references**(*entries*, *operations*)
For each instruction address in a memory map entry, calculate a list of the entries containing instructions that jump to, call or otherwise refer to that address.

Parameters

- **entries** – A collection of memory map entries.
- **operations** – A tuple of regular expression patterns. The address operand of any instruction whose operation matches one of these patterns identifies an entry point that will be marked with an asterisk in the skool file.

Returns A dictionary of entry point addresses.

The value of the *operations* argument is derived from the SnapshotReferenceOperations parameter in the [skoolkit] section of *skoolkit.ini*. In its default form, this parameter is a comma-separated list of regular expression patterns that designates 'DJNZ', 'JR', 'JP', 'CALL' and 'RST' operations as those whose address operands will be used to identify entry points in the skool file:

```
SnapshotReferenceOperations=DJ,JR,JP,CA,RS
```

To use a pattern that contains a comma, an alternative (non-alphabetic) separator can be specified in the first character of the parameter value. For example:

```
SnapshotReferenceOperations=;DJ;JR;JP;CA;RS;LD A,\(\i\);LD \(\i\),A
```

This would additionally designate the 'LD A,(nn)' and 'LD (nn),A' operations as identifying entry points. As a convenience for dealing with decimal and hexadecimal numbers, wherever `\i` appears in a pattern, it is replaced by a pattern that matches a decimal number or a hexadecimal number preceded by `$`.

Each memory map entry has the following attributes:

- *ctl* - the entry's control directive ('b', 'c', 'g', 'i', 's', 't', 'u' or 'w')
- *instructions* - a collection of instruction objects

Each instruction object has the following attributes:

- *address* - the address of the instruction
- *bytes* - the byte values of the instruction
- *label* - the instruction's label, as defined by a *@label* directive
- *operation* - the operation (e.g. 'XOR A')
- *refs* - the addresses of the instruction's indirect referrers, as declared by a *@refs* directive
- *rrefs* - the addresses of the instruction's direct referrers to be removed, as declared by a *@refs* directive

Changed in version 8.5: The `SnapshotReferenceOperations` parameter defines a list of regular expression patterns.

Changed in version 8.2: Added the *refs* and *rrefs* attributes to instruction objects.

9.2.13 Component API

The following functions are provided to facilitate access to the components and other values declared in the `[skoolkit]` section of *skoolkit.ini*.

`skoolkit.components.`**`get_component`**(*name*, **args*)
Return a component declared in the `[skoolkit]` section of *skoolkit.ini*.

Parameters

- **`name`** – The component name.
- **`args`** – Arguments passed to the component's constructor.

`skoolkit.components.`**`get_value`**(*name*)
Return a parameter value from the `[skoolkit]` section of *skoolkit.ini*.

Parameters **`name`** – The parameter name.

PYTHON MODULE INDEX

S

INDEX