
SkoolKit Documentation

Release 5.0

Richard Dymond

October 04, 2015

1	What is SkoolKit?	1
1.1	Features	1
1.2	Licence	1
2	Installing and using SkoolKit	3
2.1	Requirements	3
2.2	Installation	3
2.3	Linux/*BSD v. Windows command line	4
3	Disassembly DIY	5
3.1	Getting started	5
3.2	The control file	5
3.3	A skeleton disassembly	6
3.4	Generating a control file	6
3.5	Developing the skool file	7
3.6	Adding pokes, bugs and trivia	8
3.7	Themes	8
4	Supported assemblers	11
4.1	pasmo	11
4.2	SjASMPPlus	12
4.3	z80asm (z88dk)	12
5	Migrating from SkoolKit 4	13
5.1	D directives	13
5.2	Address ranges in control directives	13
5.3	ASM directives	13
5.4	HTML templates and CSS	14
5.5	skool2asm.py	14
5.6	skoolkit4to5.py	15
6	Changelog	17
6.1	5.0 (2015-10-04)	17
6.2	4.5 (2015-07-04)	17
6.3	4.4 (2015-05-23)	18
6.4	4.3 (2015-02-14)	18
6.5	4.2 (2014-12-07)	19
6.6	4.1.1 (2014-09-20)	19
6.7	4.1 (2014-08-30)	19

6.8	4.0 (2014-05-25)	19
6.9	Older versions	20
7	Technical reference	33
7.1	Commands	33
7.2	Parsing, rendering, and modes	42
7.3	Control files	43
7.4	Skool files	52
7.5	Skool file templates	57
7.6	Skool macros	59
7.7	Ref files	75
7.8	HTML templates	89
7.9	ASM modes and directives	99
8	Developer reference	111
8.1	Extending SkoolKit	111

What is SkoolKit?

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

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

1.1 Features

SkoolKit can:

- convert a TAP or TZX file into a ‘pristine’ snapshot (using [tap2sna.py](#))
- disassemble SNA, Z80 and SZX snapshots as well as raw memory files
- distinguish code from data by using a code execution map produced by an emulator
- build still and animated PNG/GIF images from graphic data in the game snapshot (using the [#UDG](#), [#UDGAR-RAY](#), [#FONT](#) and [#SCR](#) macros)
- create hyperlinks between routines and data blocks that refer to each other (by use of the [#R](#) macro in annotations, and automatically in the operands of `CALL` and `JP` instructions)
- neatly render lists of bugs, trivia and POKEs on separate pages (using [\[Bug:*:*\)](#), [\[Fact:*:*\)](#) and [\[Poke:*:*\)](#) sections in a *ref* file)
- produce ASM files that include bugfixes declared in the *skool* file (with [@ofix](#), [@bfix](#) and other ASM directives)
- produce TAP files from assembled code (using [bin2tap.py](#))

For a demonstration of SkoolKit’s capabilities, take a look at the complete disassemblies of [Skool Daze](#), [Back to Skool](#), [Contact Sam Cruise](#), [Manic Miner](#) and [Jet Set Willy](#). 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 Licence

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

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

Installing and using SkoolKit

2.1 Requirements

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

2.2 Installation

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

- from the zip archive or tarball available at skoolkit.ca
- from the DEB package or RPM package available at skoolkit.ca
- from [PyPI](#) by using [easy_install](#) or [pip](#)
- from the [SkoolKit PPA](#) for Ubuntu

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

2.2.1 Windows

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

```
> setup.py install
```

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

2.2.2 Linux/*BSD

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

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

This 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.

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, however, use the `-H` option:

```
$ sna2skool.py -H -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 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 the `-g` option with *sna2skool.py* to perform a rudimentary static code analysis of the snapshot file and generate a corresponding control file:

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

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

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

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

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

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

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

3.5 Developing the skool file

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

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

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

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

Annotating the code and data in a *skool* file is done by adding comments just as you would in a regular ASM file. For example, you might add a comment to the instruction at 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. You could do this manually in the *skool* file itself, or you could regenerate the *skool* file from a new control file. To ensure that you don't lose all the annotations you've already added to the *skool* file, though, you should use *skool2ctl.py* to preserve them.

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

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

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

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

This new skool file, *game-2.skool*, 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 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
```

Supported assemblers

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

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

Note: *SjASMPPlus* 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, IX1`). 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 will have to 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 binary (raw memory) files or tape files that can be used with an emulator.

4.1 *pasmo*

First, create an ASM version of the disassembly:

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

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

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

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

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

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

4.2 SjASMPPlus

First, create an ASM version of the disassembly:

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

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

```
; SjASMPPlus 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 *sjasmpplus* on this source file:

```
$ sjasmpplus game.sjasm
```

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

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

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

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

4.3 z80asm (z88dk)

First, create an ASM version of the disassembly:

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

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

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

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

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

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

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

Migrating from SkoolKit 4

SkoolKit 5 includes some changes that make it incompatible with SkoolKit 4. If you have developed a disassembly using SkoolKit 4 and find that the SkoolKit commands no longer work with your control files or *skool* files, or produce broken output, look through the following sections for tips on how to migrate your disassembly to SkoolKit 5.

5.1 D directives

In SkoolKit 4, the `D` directive in a control file could be used to define a block description or a mid-block comment. In SkoolKit 5, the `D` directive defines block descriptions only; to define a mid-block comment, use the `N` directive instead.

5.2 Address ranges in control directives

In SkoolKit 4, the extent of a sub-block directive in a control file could be specified by an address range; for example:

```
B 40000-40009
```

In SkoolKit 5, address ranges are no longer supported, and this directive must be rewritten with a length parameter:

```
B 40000,10
```

5.3 ASM directives

In SkoolKit 4, ASM directives could be declared in a control file or a *skool* file by starting a line with `; @`; for example, in a control file:

```
; @label:24576=START
```

and in a *skool* file (or skool file template):

```
; @label=START
```

This syntax is no longer supported. In SkoolKit 5, ASM directives must be declared in a control file by using the `@` directive:

```
@ 24576 label=START
```

and in a *skool* file (or skool file template) by starting a line with `@`:

```
@label=START
```

5.4 HTML templates and CSS

If you are using any custom *HTML templates* - in particular the full-page templates or the *anchor* template - or custom CSS, there are some changes to be aware of.

In SkoolKit 4:

- the default full-page templates use the XHTML 1.0 Strict DOCTYPE
- every full-page template contains its own copy of the page footer
- the *anchor* template uses an `<a>` element with a `name` attribute
- every `<a>` element that defines a hyperlink has a `class="link"` attribute

In SkoolKit 5:

- the default full-page templates have been converted to HTML5
- every full-page template uses the *footer* template to format the page footer
- the *anchor* template uses a `` element with an `id` attribute
- the `class="link"` attribute has been removed from every `<a>` element that defines a hyperlink

This means that if you are using any custom full-page templates or a custom *anchor* template, you should ensure that they are consistent (i.e. produce valid HTML5 or XHTML 1.0 as required when used in combination).

In addition, if you are using any custom full-page templates, you should either replace the page footer (the `<div class="footer">` element) with a `{t_footer}` replacement field, or reinstate the CSS rule for `div.footer` from SkoolKit 4:

```
div.footer {  
    clear: both;  
    margin-top: 10px;  
    text-align: center;  
}
```

And finally, if you have defined any custom CSS rules for hyperlinks, you should remove the ‘link’ class from the selectors. For example, this CSS rule from *skoolkit.css* in SkoolKit 4:

```
a.link { color: #ffff00; }
```

has become this in SkoolKit 5:

```
a { color: #ffff00; }
```

5.5 skool2asm.py

In SkoolKit 4 and earlier versions, *skool2asm.py* sported the `-d/--crlf` option (to use CR+LF to end lines instead of the system default), the `-i/--inst-width` option (to set the width of the instruction field), and the `-t/--tabs` option (to use tab to indent instructions). These options have been removed in SkoolKit 5, but their effects can be achieved with the new `-P/--set` option:

SkoolKit 4	SkoolKit 5
--crlf	--set crlf=1
--inst-width N	--set instruction-width=N
--tabs	--set tab=1

5.6 skoolkit4to5.py

The `skoolkit4to5.py` script may be used to convert a control file, *skool* file, skool file template or CSS file that is compatible with SkoolKit 4 into a file that will work with SkoolKit 5. For example, to convert *game.skool*:

```
$ skoolkit4to5.py game.skool > game5.skool
```

Changelog

6.1 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

6.2 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`)

6.3 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*

6.4 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

6.5 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 `` 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

6.6 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](#)

6.7 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](#)

6.8 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)

6.9 Older versions

6.9.1 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

6.9.2 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*

6.9.3 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.

Technical reference

7.1 Commands

7.1.1 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

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

Options:
  -c N, --clear N          Use a 'CLEAR N' command in the BASIC loader and leave
                           the stack pointer alone
  -e ADDR, --end ADDR      Set the end address when reading a snapshot
  -o ORG, --org ORG        Set the origin address (default: 16384 for a snapshot,
                           otherwise 65536 minus the length of FILE)
  -p STACK, --stack STACK  Set the stack pointer (default: ORG)
  -s START, --start START  Set the start address to JP to (default: ORG)
  -t TAPFILE, --tapfile TAPFILE Set the TAP filename
  -V, --version            Show SkoolKit version number and exit
```

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

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 (\$5D90).

Version	Changes
4.5	Added the <code>--clear</code> and <code>--end</code> options, and the ability to convert SNA, SZX and Z80 snapshots
3.4	Added the <code>-V</code> option and the long options
2.2.5	Added the <code>-p</code> option
1.3.1	New

7.1.2 skool2asm.py

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

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

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

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

Convert a skool file into an ASM file 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)
  -H, --hex             Write the disassembly in hexadecimal
  -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')
  -s, --ssub            Apply safe substitutions (@ssub)
  -S ADDR, --start ADDR Start converting at this address
  -u, --upper           Write the disassembly in upper case
  -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.

Version	Changes
5.0	Added the <code>--set</code> option
4.5	Added the <code>--start</code> and <code>--end</code> options
4.1	Added the <code>--writer</code> option
3.4	Added the <code>-V</code> and <code>-p</code> options and the long options
2.2.2	Added the ability to read a <i>skool</i> file from standard input
2.1.1	Added the <code>-u</code> , <code>-D</code> and <code>-H</code> options
1.1	Added the <code>-c</code> option

7.1.3 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'.

Options:
  -E ADDR, --end ADDR      Stop converting at this address
  -i, --isub               Apply instruction substitutions (@isub)
  -S ADDR, --start ADDR    Start converting at this address
  -V, --version            Show SkoolKit version number and exit
```

Version	Changes
5.0	New

7.1.4 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, 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:
  -a, --no-asm-dirs        Do not write ASM directives
  -b, --preserve-base      Preserve the base of decimal and hexadecimal values in
                           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
```

```
-l, --hex-lower      Write addresses in lower case hexadecimal format
-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:
                    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
```

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

Version	Changes
4.5	Added the <code>--start</code> and <code>--end</code> options
4.4	Added the <code>--hex-lower</code> option
3.7	Added the <code>--preserve-base</code> option
3.4	Added the <code>-V</code> option and the long options
2.4	Added the <code>-a</code> option and the ability to preserve some ASM directives
2.2.2	Added the ability to read a <i>skool</i> file from standard input
2.0.6	Added the <code>-h</code> option
1.1	New

7.1.5 skool2html.py

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

For example:

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

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

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

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

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

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

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

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

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

Options:
```



```

-a, --asm-labels      Use ASM labels
-c S/L, --config S/L  Add the line 'L' to the ref file section 'S'; this
                      option may be used multiple times
-C, --create-labels   Create default labels for unlabelled instructions
-d DIR, --output-dir DIR
                      Write files in this directory (default is '.')
-D, --decimal         Write the disassembly in decimal
-H, --hex             Write the disassembly in hexadecimal
-j NAME, --join-css NAME
                      Concatenate CSS files into a single file with this name
-l, --lower           Write the disassembly in lower case
-o, --rebuild-images  Overwrite existing image files
-p, --package-dir     Show path to skoolkit package directory and exit
-P PAGES, --pages PAGES
                      Write only these custom pages (when 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
-t, --time            Show timings
-T THEME, --theme THEME
                      Use this CSS theme; this option may be used multiple
                      times
-u, --upper           Write the disassembly in upper case
-V, --version         Show SkoolKit version number and exit
-w X, --write X       Write only these files, where X is one or more of:
                      B = Graphic glitches      o = Other code
                      b = Bugs                  P = Custom pages
                      c = Changelog             p = Pokes
                      d = Disassembly files     t = Trivia
                      i = Disassembly index     y = Glossary
                      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 *skool* files, *ref* files, CSS files, JavaScript files, font files, and files listed in the [\[Resources\]](#) section of the *ref* file:

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

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

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

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

7.1.6 skool2sft.py

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

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

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

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

Convert a skool file into a skool file template 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
                          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
-l, --hex-lower	Write addresses in lower case hexadecimal format
-S ADDR, --start ADDR	Start converting at this address
-V, --version	Show SkoolKit version number and exit

Version	Changes
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	New

7.1.7 sna2skool.py

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

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

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

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

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

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

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

	Start disassembling at this address (default=16384)
-t, --text	Show ASCII text in the comment fields
-T FILE, --sft FILE	Use FILE as the skool file template
-V, --version	Show SkoolKit version number and exit
-w W, --line-width W	Set the maximum line width of the skool file (default: 79)
-z, --defb-zfill	Pad decimal values in DEFB statements with leading zeroes

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

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

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

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

Ver- sion	Changes
5.0	Added support for SpecEmu’s 64K code execution map files
4.4	Added the --ctl-hex-lower and --end options
4.3	Added the --line-width option
3.4	Added the -V and -R options and the long options
3.3	Added the -M option, along with support for code execution maps produced by Fuse, SpecEmu, Spud, Zero and Z80; added the ability to read 128K SNA snapshots
3.2	Added the -p option, and the ability to read SZX snapshots and 128K Z80 snapshots
2.4	Added the -T option
2.1.2	Added the -L option
2.1	Added the -H option
2.0.6	Added the -h option
2.0.1	Added the -o, -r and -l options, and the ability to read binary files
2.0	Added the -n, -m and -z options
1.0.5	Added the -t option
1.0.4	Added the -g and -s options

7.1.8 tap2sna.py

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

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

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

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

Convert a TAP or TZX file (which may be inside a zip archive) into a Z80 snapshot. INPUT may be the full URL to a remote zip archive or TAP/TZX file,

or the path to a local file. Arguments may be read from FILE instead of (or as well as) being given on the command line.

Options:

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

Note that support for TZX files is limited to block types 0x10 (standard speed data), 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.) 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, and modify memory with XOR and ADD operations before the snapshot is saved. For more information on the operations that the `--ram` option can perform, run:

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

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

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

then:

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

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

Ver- sion	Changes
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

7.1.9 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
  -V, --version                 Show SkoolKit version number and exit
```

Version	Changes
5.0	New

7.2 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.

7.2.1 Parsing

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

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

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

HTML mode

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

- *@assemble*
- *@keep*
- *@label*
- *@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.

7.2.2 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.

7.3 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.

7.3.1 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.
```

7.3.2 Register values

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

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

7.3.3 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.
```

7.3.4 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.
```

7.3.5 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.
```

7.3.6 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).

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.

7.3.7 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 ; }
```

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:B1
B 40004,3,1:T2
```

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

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

which is equivalent to:

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

DEFS statements may specify a byte value other than zero; for example:

```
60000 DEFS 20,170
60020 DEFS 40,85
```

These statements can be encoded in a control file thus:

```
S 60000,60,20:170,40:85
```

7.3.8 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.

7.3.9 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 or as a character, attach a b (binary), c (character), d (decimal) or h (hexadecimal) 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,d1,b1,h2
S 40008,8,8:c"! "
```

will result in something like this:

```
40000 DEFB %10101010,23,43,$5F
40004 DEFB 56
40005 DEFB %111110000
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
40004 DEFB $0F,%111110000,93
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
```

7.3.10 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
```

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.
```

7.3.11 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).

7.3.12 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
```

7.3.13 Limitations

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

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

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

7.3.14 Revision history

Ver- sion	Changes
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 <i>@ignoreua</i> 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

7.4 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).

7.4.1 Skool file format

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

- 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: ; * @bcdgirstuw; 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
```

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

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

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

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
```

Entry line format

Each line in an entry may start with one of ; * @bcdgirstuw, 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)
- `d` denotes the first instruction in a *data definition entry*
- `g` denotes the first instruction in a game status buffer entry
- `i` denotes an ignored entry
- `r` denotes the first instruction in a *remote entry*
- `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*).

7.4.2 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.

7.4.3 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.

7.4.4 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
```

7.4.5 Data definition entries

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

For example:

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

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

If this data definition entry were not present, the memory snapshot would contain zeroes at addresses 44919-44926, and the image created by the `#UDG` macro would be blank. The reason for this is that by default, the skool file parser does not convert assembly language instructions into byte values. However, this behaviour can be changed by using the `@assemble` directive.

7.4.6 Remote entries

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

For example:

```
r26880 main
```

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

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

```
r29012 main
29015
```

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

```
#R29015@main
```

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

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

7.4.7 Revision history

Version	Changes
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 <i>s</i> 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
2.0	Added support for data definition entries and remote entries

7.5 Skool file templates

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

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

7.5.1 Skool file template format

A skool file template has the same layout as a *skool* file, except that the lines in *b*, *c*, *g*, *s*, *t*, *u* and *w* blocks that correspond to Z80 instructions look like this:

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

where:

- *x* is one of the characters * b c g s t u w (with the same meaning as in a *skool file*)
- *X* is one of the characters B C S T W (with the same meaning as in a *control file*)
- ##### is the address at which to start disassembling
- *N* is the number of bytes to disassemble (or a list of sublengths; see *Sub-block syntax*)
- *c* is the index of the column in which the comment marker (;) appears in the line (if it does appear)
- *comment*, if present, is the instruction-level comment for the line on which the instruction occurs

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

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

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

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

7.5.2 Sub-block syntax

The syntax for specifying B, C, S, T and W sub-blocks is analogous to the syntax used in *control files*. A brief summary of the syntax is given here.

Sequences of DEFB statements can be declared on a single line by a comma-separated list of sublengths thus:

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

which is equivalent to:

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

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

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

```
b30000 DEFB 1,2,3,4,"Hello!"
30010 DEFM 5,6
30012 DEFM "B",7,8
```

Such statements are preserved in a skool file template thus:

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

DEFB, DEFM, DEFS and DEFW statements may contain numeric values in various bases or as characters. For example:

```
b40000 DEFB %10101010,23,43,$5F
40004 DEFB 56
40005 DEFB %11110000
40006 DEFB $2B,$80
40008 DEFS 8,"!"
```

These statements may be preserved in a skool file template thus:

```
bB40000,b1:d2:h1,d1,b1,h2
S40008,8:c"!"
```

Instruction operands may also contain numeric values in various bases or as characters. For example:

```
c50000 LD A,%00011000
50002 LD B,"!"
50004 LD (IX+$1A),%00001111
```

These instructions may be preserved in a skool file template by using b (binary), c (character), d (decimal) and h (hexadecimal) prefixes on sublength parameters thus:

```
cC50000,b2,c2,hb4
```

7.5.3 Skool file template comments

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

7.5.4 Data definition entries

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

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

```
d32400 DEFB 0

; Number of lives
bB32400,1
```

Note that in order to take effect, a 'd' block must appear before the block that it overrides.

7.5.5 Revision history

Ver- sion	Changes
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
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 <i>s</i> and <i>S</i> directives and support for DEFS statements with non-zero byte values
3.1.4	Added support for DEFB and DEFM statements that contain both strings and bytes
2.4	New

7.6 Skool macros

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

Skool macros have the following general form:

```
#MACRO rparam1, rparam2, ... [, oparam1, oparam2, ...]
```

where:

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

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

```
#UDG39144
```

is equivalent to:

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

and:

```
#UDG30115,,2
```

is equivalent to:

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

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

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

7.6.1 General macros

#BUG

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

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

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

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

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

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

For example:

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

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

In ASM mode, this instance of the #BUG macro expands to ‘bug’.

See also [#FACT](#) and [#POKE](#).

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

#CALL

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

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

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

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

For example:


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

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

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

Version	Changes
3.1	Added support for ASM mode
2.1	New

#CHR

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

```
#CHRnum
```

or:

```
#CHR(num)
```

For example:

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

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

Version	Changes
3.1	New

#D

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

```
#Daddr
```

- `addr` is the address of the entry.

For example:

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

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

#EREFS

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

```
#EREF$addr
```

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

See also [#REFS](#).

Version	Changes
3.1	Added support for ASM mode

#FACT

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

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

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

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

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

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

For example:

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

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

See also [#BUG](#) and [#POKE](#).

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

#HTML

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

```
#HTML(text)
```

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

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

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

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

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

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

See also [#UDGTABLE](#).

Version	Changes
3.1.2	New

#LINK

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

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

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

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

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

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

For example:

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

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

In ASM mode, this instance of the #LINK macro expands to ‘glossary’.

Version	Changes
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)]<items>LIST#
```

- class is the CSS class to use for the element

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

For example:

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

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

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

Version	Changes
3.2	New

#POKE

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

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

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

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

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

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

For example:

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

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

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

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

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

#R

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

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

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

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

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

For example:

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

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

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

Ver- sion	Changes
3.5	Added the ability to resolve (in HTML mode) the address of an entry point in another disassembly when an appropriate <i>remote entry</i> is defined
2.0	Added support for the <code>@code</code> notation

#REFS

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

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

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

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

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

See also [#ERES](#).

Version	Changes
3.1	Added support for ASM mode
1.0.6	Added the <code>prefix</code> parameter

#REG

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

```
#REGreg
```

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

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

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

```
abcdefghijklmnopqrstuvwxyz'
```

For example:

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

#SPACE

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

```
#SPACE [num]
```

or:

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

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

For example:

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

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

```
&#160;&#160;&#160;&#160;&#160;&#160;&#160;&#160;
```

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

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

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

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

#TABLE

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

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

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

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

For example:

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

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

By default, cells will be rendered as `<td>` elements. To 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 }
; TABLE#
```

See also [#UDGTABLE](#).

#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](#).

7.6.2 Image macros

The [#FONT](#), [#SCR](#), [#UDG](#) and [#UDGARRAY](#) 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.

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
```

#FONT

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

```
#FONT[: (text) ] addr [, chars, attr, scale] [{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)
- `CROP` is the cropping specification (see [Cropping](#))
- `fname` is the name of the image file (default: `'font'`); `'png'` or `'gif'` will be appended (depending on the default image format specified in the [ImageWriter](#) section of the *ref* file) if not present

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

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

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

For example:

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

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

Version	Changes
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the element
4.0	Added support for keyword arguments
3.6	Added the <code>text</code> parameter, and made the <code>chars</code> parameter optional
3.0	Added image-cropping capabilities
2.0.5	Added the <code>fname</code> parameter and support for regular 8x8 fonts

#SCR

In HTML mode, the #SCR (SCReenshot) macro expands to an 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] [{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)
- CROP is the cropping specification (see [Cropping](#))
- `fname` is the name of the image file (default: `'scr'`); `'.png'` or `'.gif'` will be appended (depending on the default image format specified in the [\[ImageWriter\]](#) section of the `ref` file) if not present

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

For example:

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

Version	Changes
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the <code></code> element
4.0	Added support for keyword arguments
3.0	Added image-cropping capabilities and the <code>df</code> and <code>af</code> parameters
2.0.5	Added the <code>scale</code> , <code>x</code> , <code>y</code> , <code>w</code> , <code>h</code> and <code>fname</code> parameters

#UDG

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

```
#UDGaddr[,attr,scale,step,inc,flip,rotate,mask][: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](#))
- MASK is the mask specification (see below)
- CROP is the cropping specification (see [Cropping](#))
- `fname` is the name of the image file (if not given, a name based on `addr`, `attr` and `scale` will be generated); `'.png'` or `'.gif'` will be appended (depending on the default image format specified in the [\[ImageWriter\]](#) section of the `ref` file) if not present

The 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)

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

For example:

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

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

Version	Changes
4.3	Added the ability to create frames
4.2	Added the ability to specify alt text for the <code></code> element
4.0	Added the <code>mask</code> parameter and support for AND-OR masking; added support for keyword arguments
3.1.2	Made the <code>attr</code> parameter optional
3.0	Added image-cropping capabilities
2.4	Added the <code>rotate</code> parameter
2.3.1	Added the <code>flip</code> parameter
2.1	Added support for masks
2.0.5	Added the <code>fname</code> parameter

#UDGARRAY

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

```
#UDGARRAYwidth[,attr,scale,step,inc,flip,rotate,mask];SPEC1[;SPEC2;...][{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](#))
- `CROP` is the cropping specification (see [Cropping](#))
- `fname` is the name of the image file; `.png` or `.gif` will be appended (depending on the default image format specified in the [\[ImageWriter\]](#) section of the *ref* file) if not present

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

```
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)

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

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

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

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

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

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

For example:

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

In HTML mode, this instance of the `#UDGARRAY` macro expands to an `` 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*.

Version	Changes
4.2	Added the ability to specify alt text for the element
4.0	Added the <code>mask</code> 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 <code>rotate</code> parameter
2.3.1	Added the <code>flip</code> parameter
2.2.5	Added support for masks
2.0.5	New

Alt text

The value of the `alt` attribute in the element created by an image macro 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; each one has the form:

```
name[,delay]
```

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

For example:

```
; #UDGTABLE {  
; #FONT:(hello)$3D00(hello*) |  
; #FONT:(there)$3D00(there*) |  
; #FONT:(peeps)$3D00(peeps*) |  
; #UDGARRAY*hello,50;there;peeps(hello_there_peeps.gif)  
; } 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.

Cropping

Each image macro accepts 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.

Masks

The `#UDG` and `#UDGARRAY` 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 `GIFTransparency=1` or `PNGAlpha=0` in the [\[ImageWriter\]](#) section.

7.6.3 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 `DEFB`, `DEFM`, `DEFS` and `DEFW` statements in the *skool* file. Each macro expands to an empty string.

#POKES

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

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

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

For example:

```
The UDG looks like this:

#UDG32768(udg_orig)

But it's supposed to look like this:

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

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

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

#POPS

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

```
#POPS
```

Version	Changes
3.1	Added support for ASM mode

#PUSHS

The `#PUSHS` (PUSH Snapshot) macro saves the current 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.

Version	Changes
3.1	Added support for ASM mode

7.7 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.

7.7.1 [Bug:*:*]

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

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

Second paragraph.

...
```

where:

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

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

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

7.7.2 [Changelog:*

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

```
[Changelog:title]
Intro text.

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

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

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

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

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

Version	Changes
2.2.5	New

7.7.3 [Colours]

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

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

or:

```
name=#RGB
```

where:

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

Recognised colour names and their default RGB values are:

- TRANSPARENT: 0,254,0 (#00fe00)
- BLACK: 0,0,0 (#000000)
- BLUE: 0,0,197 (#0000c5)
- RED: 197,0,0 (#c50000)
- MAGENTA: 197,0,197 (#c500c5)
- GREEN: 0,198,0 (#00c600)
- CYAN: 0,198,197 (#00c6c5)
- YELLOW: 197,198,0 (#c5c600)
- WHITE: 205,198,205 (#cdc6cd)

- BRIGHT_BLUE: 0,0,255 (#0000ff)
- BRIGHT_RED: 255,0,0 (#ff0000)
- BRIGHT_MAGENTA: 255,0,255 (#ff00ff)
- BRIGHT_GREEN: 0,255,0 (#00ff00)
- BRIGHT_CYAN: 0,255,255 (#00ffff)
- BRIGHT_YELLOW: 255,255,0 (#ffff00)
- BRIGHT_WHITE: 255,255,255 (#ffffff)

Version	Changes
3.4	Added support for hexadecimal RGB triplets
2.0.5	New

7.7.4 [Config]

The Config section contains configuration parameters in the format:

```
name=value
```

Recognised parameters are:

- GameDir - the root directory of the game's HTML disassembly; if not specified, the base name of the *skool* or *ref* file given on the *skool2html.py* command line will be used
- HtmlWriterClass - the name of the Python class to use for writing the HTML disassembly of the game (default: `skoolkit.skoolhtml.HtmlWriter`); if the class is in a module that is not in the module search path (e.g. a standalone module that is not part of an installed package), the module's location may be specified thus: `/path/to/moduledir:module.classname`
- RefFiles - a semicolon-separated list of extra *ref* files to use (in addition to the one named on the *skool2html.py* command line, and any others with the same filename prefix)
- SkoolFile - the name of the main *skool* file to use if not given on the *skool2html.py* command line; if not specified, the *skool* file with the same base name as the *ref* file will be used

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

Version	Changes
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

7.7.5 [Fact:*:*]

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

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

Second paragraph.

...
```

where:

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

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

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

7.7.6 [Game]

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

<code>name=value</code>

Recognised parameters are:

- `AddressAnchor` - the format of the anchors attached to instructions on disassembly pages and entries on memory map pages (default: `{address}`)
- `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.’; the string `$VERSION` is replaced by the version number of SkoolKit)
- `Font` - the base name of the font file to use (default: `None`); multiple font files can be declared by separating their names with semicolons
- `Game` - the name of the game, which appears in the title of every page, and also in the header of every page (if no logo is defined); if not specified, the base name of the *skool* file is used
- `GameStatusBufferIncludes` - a comma-separated list of addresses of entries to include on the ‘Game status buffer’ page in addition to those that are marked with a `g` (see the *skool file format reference*)
- `InputRegisterTableHeader` - the text 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
- `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
- `TitlePrefix` - the prefix to use before the game name or logo in the header of the main index page (default: 'The complete')
- `TitleSuffix` - the suffix to use after the game name or logo in the header of the main index page (default: 'RAM disassembly')

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}`.

Note that an address anchor that starts with an upper case letter (e.g. `#FF00`) will be interpreted as a skool macro, and so any format string that could produce such an anchor should be avoided.

Version	Changes
4.3	Added the <code>AddressAnchor</code> parameter
4.1	Added the <code>LinkInternalOperands</code> parameter
4.0	Set default values for the <code>InputRegisterTableHeader</code> and <code>OutputRegisterTableHeader</code> parameters; added the <code>Copyright</code> , <code>Created</code> and <code>Release</code> parameters (which used to live in the [Info] section in SkoolKit 3)
3.7	Added the <code>JavaScript</code> parameter
3.5	Added the <code>Font</code> , <code>LogoImage</code> and <code>StyleSheet</code> parameters (all of which used to live in the [Paths] section, <code>LogoImage</code> by the name <code>Logo</code>)
3.4	Added the <code>LinkOperands</code> parameter
3.1.2	Added the <code>InputRegisterTableHeader</code> and <code>OutputRegisterTableHeader</code> parameters
2.0.5	Added the <code>Logo</code> parameter
2.0.3	Added the <code>GameStatusBufferIncludes</code> parameter

7.7.7 [Glossary:*)]

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

```
[Glossary:term]
First paragraph.

Second paragraph.

...
```

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

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

Version	Changes
3.1.3	Added support for multiple paragraphs

7.7.8 [GraphicGlitch:*.*)]

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

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

Second paragraph.

...
```

where:

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

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

7.7.9 [ImageWriter]

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

```
name=value
```

Recognised parameters are:

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

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

```
#FONT32768,26
```

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

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

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

Ver- sion	Changes
3.0.1	Added the <code>DefaultFormat</code> , <code>GIFEnableAnimation</code> , <code>GIFTransparency</code> , <code>PNGAlpha</code> and <code>PNGEnableAnimation</code> parameters
3.0	New

7.7.10 [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 (see below); `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

7.7.11 [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

7.7.12 [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:

- `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:*\)](#), [\[Page:*\)](#) or [\[PageContent:*\)](#) 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
2.5	Added the <code>UnusedMap</code> page ID
2.2.5	Added the <code>Changelog</code> page ID
2.0.5	New

7.7.13 [\[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, every type is shown); entry types are identified as follows:
 - `b` - DEFB blocks
 - `c` - routines
 - `g` - game status buffer entries
 - `G` - entries whose address appears in the `GameStatusBufferIncludes` parameter in the [\[Game\]](#) section
 - `s` - blocks containing bytes that are all the same value

- t - messages
- u - unused addresses
- w - DEFW blocks
- Intro - the text (which may contain HTML markup and *skool macros*) displayed at the top of the memory map page (default: '')
- LengthColumn - 1 to display the 'Length' column, or 0 not to (default: 0)
- 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)

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 *MemoryMap* template is used.

Ver- sion	Changes
4.0	Added the <code>EntryDescriptions</code> and <code>LengthColumn</code> parameters, and support for the <code>G</code> identifier in the <code>EntryTypes</code> parameter
2.5	New

7.7.14 [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

By default, the index page is written to `CodeID/CodeID.html`, and the disassembly pages are written in a directory named `CodeID`.

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 <code>Source</code> parameter optional
2.0	New

7.7.15 [Page:*)

A `Page:*)` section may be used to either declare a page that already exists, or define a custom page in the HTML disassembly (in conjunction with a corresponding [\[PageContent:*\)](#) section). The section 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; this defaults to the contents of the corresponding [\[PageContent:*\)](#) section, but may be specified here if the source can be written on a single line

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 can be defined 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 stock *Page* template is used.

Note that a `Page:*)` section may be empty; if so, it may be omitted from the *ref* file.

Version	Changes
3.5	The <code>JavaScript</code> parameter specifies the JavaScript file(s) to use
2.1	New

7.7.16 [PageContent:*)

A `PageContent:*)` section contains the HTML source of the body of a custom page (optionally defined in a [\[Page:*\)](#) section). The section name takes the form:

[PageContent:PageID]

where `PageID` is the unique ID of the page.

The HTML source may contain *skool macros*.

Version	Changes
2.1	New

7.7.17 [PageHeaders]

The `PageHeaders` section defines the header text for every page in the HTML disassembly. Each line has the form:

`PageID=header`

where:

- `PageID` is the ID of the page
- `header` is the header text

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: 'Data')
- `Asm-u` - disassembly pages for 'u' blocks (default: 'Unused')
- `Asm-w` - disassembly pages for 'w' blocks (default: 'Data')
- `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
- `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 header text for a page is the same as the title defined in the `[Titles]` section, except where indicated above.

The header text for a page defined by a `[MemoryMap:*)`, `[OtherCode:*)`, `[Page:*)` or `[PageContent:*)` section also defaults to the title, but can be overridden in this section.

Note that the header of the disassembly index page (`GameIndex`) is not defined in this section; it is composed from the values of the `TitlePrefix` and `TitleSuffix` parameters in the `[Game]` section.

Version	Changes
4.0	New

7.7.18 [Paths]

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

ID=path

where:

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

Recognised file IDs and their default paths are:

- `Bugs` - the 'Bugs' page (default: `reference/bugs.html`)
- `Changelog` - the 'Changelog' page (default: `reference/changelog.html`)
- `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`)
- `UnusedMap` - the 'Unused addresses' memory map page (default: `maps/unused.html`)

Recognised directory IDs and their default paths are:

- `CodePath` - the directory in which the disassembly pages are written (default: `asm`)
- `FontImagePath` - the directory in which font images (created by the `#FONT` macro) are placed (default: `images/font`)
- `FontPath` - the directory in which font files specified by the `Font` parameter in the `[Game]` section are placed (default: `.`)
- `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: `images/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: `images/udgs`)

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`.

Version	Changes
4.3	Added the <code>CodeFiles</code> file ID
3.1.1	Added the <code>FontPath</code> directory ID
2.5	Added the <code>UnusedMap</code> file ID
2.2.5	Added the <code>Changelog</code> file ID
2.1.1	Added the <code>CodePath</code> directory ID
2.0.5	Added the <code>FontImagePath</code> directory ID
2.0	New

7.7.19 [Poke:*:*]

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

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

Second paragraph.

...
```

where:

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

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

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

7.7.20 [Resources]

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

```
fname=destDir
```

where:

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

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

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

Version	Changes
3.6	New

7.7.21 [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

7.7.22 [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')
- `Asm-c` - disassembly pages for 'c' blocks (default: 'Routine at')
- `Asm-g` - disassembly pages for 'g' blocks (default: 'Game status buffer entry at')
- `Asm-s` - disassembly pages for 's' blocks (default: 'Unused RAM at')
- `Asm-t` - disassembly pages for 't' blocks (default: 'Data at')
- `Asm-u` - disassembly pages for 'u' blocks (default: 'Unused RAM at')
- `Asm-w` - disassembly pages for 'w' blocks (default: 'Data at')
- `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')

The title of a page defined by a `[MemoryMap:*)`, `[OtherCode:*)`, `[Page:*)` or `[PageContent:*)` section defaults to the page ID, but can be overridden in this section.

Version	Changes
4.0	Added the <code>Asm-*</code> page IDs
2.5	Added the <code>UnusedMap</code> page ID
2.2.5	Added the <code>Changelog</code> page ID
2.0.5	New

7.7.23 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:

```
[PageContent:Custom]
<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>
```

7.7.24 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:

```
[PageContent:Custom]
<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.

7.8 HTML templates

Every page in an HTML disassembly is built from a single full-page template and several 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:

- `index_href` - the relative path to the disassembly index page
- `page_header` - the page header text (as defined in the [\[PageHeaders\]](#) section)
- `page_id` - the page ID (e.g. `GameIndex`, `MemoryMap`)
- `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.

In addition to the universal identifiers, the following page-level identifiers are available in every full-page template:

- `m_javascript` - replaced by any number of copies of the [javascript](#) subtemplate
- `m_stylesheet` - replaced by one or more copies of the [stylesheet](#) subtemplate
- `t_footer` - replaced by a copy of the [footer](#) subtemplate

7.8.1 Asm

The `Asm` template is the full-page template that is used to build disassembly pages.

It contains the following identifiers (in addition to the universal and page-level identifiers):

- `disassembly` - replaced by sequences of copies of the [asm_instruction](#) subtemplate, punctuated by copies of the [asm_comment](#) subtemplate
- `entry` - a dictionary of parameters corresponding to the current memory map entry (see below)
- `next_entry` - a dictionary of parameters corresponding to the next memory map entry (see below)
- `prev_entry` - a dictionary of parameters corresponding to the previous memory map entry (see below)
- `registers_input` - replaced by any number of copies of the [asm_register](#) subtemplate
- `registers_output` - replaced by any number of copies of the [asm_register](#) subtemplate

The parameters in the `prev_entry`, `entry` and `next_entry` dictionaries 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](#))
- `annotated` - '1' if any instructions in the entry have a non-empty comment field, '0' otherwise
- `byte` - the LSB of the entry address
- `description` - the entry description
- `exists` - '1' if the entry exists, '0' otherwise
- `href` - the relative path to the disassembly page for the entry (useful only for `prev_entry` and `next_entry`)
- `label` - the ASM label of the first instruction in the entry
- `labels` - '1' if any instructions in the entry have an ASM label, '0' otherwise
- `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')

To see the default Asm template, run the following command:

```
$ skool2html.py -r Template:Asm
```

7.8.2 GameIndex

The `GameIndex` template is the full-page template that is used to build the disassembly index page.

It contains the following identifier (in addition to the universal and page-level identifiers):

- `m_index_section` - replaced by any number of copies of the *index_section* subtemplate

To see the default `GameIndex` template, run the following command:

```
$ skool2html.py -r Template:GameIndex
```

7.8.3 MemoryMap

The `MemoryMap` template is the full-page template that is used to build memory map pages and the 'Game status buffer' page.

It contains the following identifiers (in addition to the universal and page-level identifiers):

- `MemoryMap` - a dictionary of the parameters in the corresponding *[MemoryMap:*)* section
- `m_map_entry` - replaced by one or more copies of the *map_entry* subtemplate

To see the default `MemoryMap` template, run the following command:

```
$ skool2html.py -r Template:MemoryMap
```

7.8.4 Page

The `Page` template is the full-page template that is used to build custom pages defined by *[Page:*)* and *[PageContent:*)* sections.

It contains the following identifier (in addition to the universal and page-level identifiers):

- `content` - replaced by the value of the `PageContent` parameter in the corresponding *[Page:*)* section

To see the default `Page` template, run the following command:

```
$ skool2html.py -r Template:Page
```

7.8.5 Reference

The `Reference` template is the full-page template that is used to build the 'Bugs', 'Trivia', 'Pokes', 'Glossary', 'Graphic glitches' and 'Changelog' pages.

It contains the following identifiers (in addition to the universal and page-level identifiers):

- `entries` - replaced by one or more copies of the [changelog_entry](#) subtemplate (on the ‘Changelog’ page), or the [reference_entry](#) subtemplate (on the ‘Bugs’, ‘Trivia’, ‘Pokes’, ‘Glossary’ and ‘Graphic glitches’ pages)
- `m_contents_list_item` - replaced by one or more copies of the [contents_list_item](#) subtemplate

To see the default Reference template, run the following command:

```
$ skool2html.py -r Template:Reference
```

7.8.6 anchor

The `anchor` template is the subtemplate used to format a page anchor (by default, a `` element with an `id` attribute).

It contains the following identifier (in addition to the universal identifiers):

- `anchor` - the value of the `id` attribute

To see the default `anchor` template, run the following command:

```
$ skool2html.py -r Template:anchor
```

7.8.7 asm_comment

The `asm_comment` template is the subtemplate used by the [Asm](#) full-page template to format block start comments, mid-block comments and block end comments.

It contains the following identifiers (in addition to the universal identifiers):

- `entry` - a dictionary of parameters corresponding to the current memory map entry (see [Asm](#))
- `m_paragraph` - replaced by one or more copies of the [paragraph](#) subtemplate
- `t_anchor` - replaced by a copy of the [anchor](#) subtemplate (when formatting a block start comment or a mid-block comment), or by an empty string (when formatting a block end comment)

To see the default `asm_comment` template, run the following command:

```
$ skool2html.py -r Template:asm_comment
```

7.8.8 asm_instruction

The `asm_instruction` template is the subtemplate used by the [Asm](#) full-page template to format an instruction (including its label, address, operation and comment).

It contains the following identifiers (in addition to the universal identifiers):

- `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](#))
- `annotated` - ‘1’ if the instruction has a comment field, ‘0’ otherwise
- `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
- `entry` - a dictionary of parameters corresponding to the memory map entry that contains the instruction (see [Asm](#))

- `label` - the instruction's ASM label
- `operation` - the assembly language operation (e.g. 'LD A,B'), with operand hyperlinked if appropriate
- `t_anchor` - replaced by a copy of the [anchor](#) subtemplate

To see the default `asm_instruction` template, run the following command:

```
$ skool2html.py -r Template:asm_instruction
```

7.8.9 asm_register

The `asm_register` template is the subtemplate used by the [Asm](#) full-page template to format each row in a table of input register values or output register values.

It contains the following identifiers (in addition to the universal identifiers):

- `description` - the register's description (as it appears in the register section for the current entry in the *skool* file)
- `entry` - a dictionary of parameters corresponding to the current memory map entry (see [Asm](#))
- `name` - the register's name (e.g. 'HL')

To see the default `asm_register` template, run the following command:

```
$ skool2html.py -r Template:asm_register
```

7.8.10 changelog_entry

The `changelog_entry` is the subtemplate used by the [Reference](#) full-page template to format each entry on the 'Changelog' page.

It contains the following identifiers (in addition to the universal identifiers):

- `description` - the changelog entry intro text
- `num` - '1' or '2', depending on the order of the entry on the page
- `t_anchor` - replaced by a copy of the [anchor](#) subtemplate (with the entry title as the anchor name)
- `t_changelog_item_list` - replaced by a copy of the [changelog_item_list](#) subtemplate
- `title` - the changelog entry title

To see the default `changelog_entry` template, run the following command:

```
$ skool2html.py -r Template:changelog_entry
```

7.8.11 changelog_item_list

The `changelog_item_list` template is the subtemplate used by the [changelog_entry](#) subtemplate to format a list of changelog items, and also by the `changelog_item` or [list_item](#) subtemplate to format a list of subitems or subsubitems etc.

It contains the following identifiers (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.

- `m_changelog_item` - replaced by one or more copies of the `changelog_item` subtemplate if it exists, or the *list_item* subtemplate otherwise

To see the default `changelog_item_list` template, run the following command:

```
$ skool2html.py -r Template:changelog_item_list
```

7.8.12 contents_list_item

The `contents_list_item` template is the subtemplate used by the *Reference* full-page template to format each item in the contents list on the ‘Bugs’, ‘Trivia’, ‘Pokes’, ‘Glossary’, ‘Graphic glitches’ and ‘Changelog’ pages.

It contains the following identifiers (in addition to the universal identifiers):

- `href` - the URL to the entry on the page
- `title` - the entry title

To see the default `contents_list_item` template, run the following command:

```
$ skool2html.py -r Template:contents_list_item
```

7.8.13 footer

The `footer` template is the subtemplate used by the full-page templates 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.

7.8.14 img

The `img` template is the subtemplate used to format `` elements.

It contains the following identifiers (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
```

7.8.15 index_section

The `index_section` template is the subtemplate used by the *GameIndex* full-page template to format each group of links on the disassembly index page.

It contains the following identifiers (in addition to the universal identifiers):

- `header` - the header text for the group of links (as defined in the name of the *[Index:*:*)* section)
- `m_index_section_item` - replaced by one or more copies of the *index_section_item* subtemplate

To see the default `index_section` template, run the following command:

```
$ skool2html.py -r Template:index_section$
```

7.8.16 index_section_item

The `index_section_item` template is the subtemplate used by the `index_section` subtemplate to format each link in a link group on the disassembly index page.

It contains the following identifiers (in addition to the universal identifiers):

- `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 `index_section_item` template, run the following command:

```
$ skool2html.py -r Template:index_section_item
```

7.8.17 javascript

The `javascript` template is the subtemplate used by the full-page templates to format each `<script>` element in the head of a page.

It contains the following identifier (in addition to the universal identifiers):

- `src` - the relative path to the JavaScript file

To see the default `javascript` template, run the following command:

```
$ skool2html.py -r Template:javascript
```

7.8.18 link

The `link` template is the subtemplate used to format the hyperlinks created by the `#BUG`, `#FACT`, `#POKE`, `#LINK` and `#R` macros, and the hyperlinks in instruction operands on disassembly pages.

It contains the following identifiers (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
```

7.8.19 list

The `list` template is used by the `#LIST` macro to format a list.

It contains the following identifiers (in addition to the universal identifiers):

- `class` - the CSS class name for the list
- `m_list_item` - replaced by any number of copies of the `list_item` subtemplate

To see the default `list` template, run the following command:

```
$ skool2html.py -r Template:list$
```

New in version 4.2.

7.8.20 list_item

The `list_item` template is the subtemplate used by the *list* template to format each item in the list, and also by the *changelog_item_list* subtemplate to format each item in a changelog item list.

It contains the following identifier (in addition to the universal identifiers):

- `item` - replaced by the text of the list item

To see the default `list_item` template, run the following command:

```
$ skool2html.py -r Template:list_item
```

New in version 4.2.

7.8.21 map_entry

The `map_entry` template is the subtemplate used by the *MemoryMap* full-page template to format each entry on the memory map pages and the 'Game status buffer' page.

It contains the following identifiers (in addition to the universal identifiers):

- `MemoryMap` - a dictionary of parameters from the corresponding *[MemoryMap:*)* section
- `entry` - a dictionary of parameters corresponding to the current memory map entry

The parameters in the `entry` dictionary 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*)
- `byte` - the LSB of the entry address
- `description` - the entry description
- `exists` - '1'
- `href` - the relative path to the disassembly page for the entry
- `label` - the ASM label of the first instruction in the entry
- `labels` - '1' if any instructions in the entry have an ASM label, '0' otherwise
- `location` - the address of the entry as a decimal number
- `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')

To see the default `map_entry` template, run the following command:

```
$ skool2html.py -r Template:map_entry
```

7.8.22 paragraph

The `paragraph` template is the subtemplate used to format each paragraph in the following items:

- memory map entry descriptions (on disassembly pages and memory map pages)
- mid-block comments and block end comments on disassembly pages
- entries on the ‘Bugs’, ‘Trivia’, ‘Pokes’, ‘Glossary’, ‘Graphic glitches’ and ‘Changelog’ pages

It contains the following identifier (in addition to the universal identifiers):

- `paragraph` - the text of the paragraph

To see the default `paragraph` template, run the following command:

```
$ skool2html.py -r Template:paragraph
```

7.8.23 reference_entry

The `reference_entry` template is the subtemplate used by the *Reference* full-page template to format each entry on the ‘Bugs’, ‘Trivia’, ‘Pokes’, ‘Glossary’ and ‘Graphic glitches’ pages.

It contains the following identifiers (in addition to the universal identifiers):

- `contents` - replaced by the pre-formatted contents of the relevant *[Bug:*.*)*, *[Fact:*.*)*, *[Poke:*.*)*, *[Glossary:*.*)* or *[GraphicGlitch:*.*)* section
- `num` - ‘1’ or ‘2’, depending on the order of the entry on the page
- `title` - the entry title

To see the default `reference_entry` template, run the following command:

```
$ skool2html.py -r Template:reference_entry
```

7.8.24 reg

The `reg` template is the subtemplate used by the *#REG* macro to format a register name.

It contains the following identifier (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
```

7.8.25 stylesheet

The `stylesheet` template is the subtemplate used by the full-page templates to format each `<link>` element for a CSS file in the head of a page.

It contains the following identifier (in addition to the universal identifiers):

- `href` - the relative path to the CSS file

To see the default `stylesheet` template, run the following command:

```
$ skool2html.py -r Template:stylesheet
```

7.8.26 table

The `table` template is used by the `#TABLE` macro to format a table.

It contains the following identifiers (in addition to the universal identifiers):

- `class` - the CSS class name for the table
- `m_table_row` - replaced by any number of copies of the `table_row` subtemplate

To see the default `table` template, run the following command:

```
$ skool2html.py -r Template:table$
```

New in version 4.2.

7.8.27 table_cell

The `table_cell` template is the subtemplate used by the `table_row` template to format each non-header cell in the table row.

It contains the following identifiers (in addition to the universal identifiers):

- `class` - the CSS class name for the cell
- `colspan` - the number of columns spanned by the cell
- `contents` - the contents of the cell
- `rowspan` - the number of rows spanned by the cell

To see the default `table_cell` template, run the following command:

```
$ skool2html.py -r Template:table_cell
```

New in version 4.2.

7.8.28 table_header_cell

The `table_header_cell` template is the subtemplate used by the `table_row` template to format each header cell in the table row.

It contains the following identifiers (in addition to the universal identifiers):

- `colspan` - the number of columns spanned by the cell
- `contents` - the contents of the cell
- `rowspan` - the number of rows spanned by the cell

To see the default `table_header_cell` template, run the following command:

```
$ skool2html.py -r Template:table_header_cell
```

New in version 4.2.

7.8.29 table_row

The `table_row` template is the subtemplate used by the `table` template to format each row in the table.

It contains the following identifier (in addition to the universal identifiers):

- `cells` - replaced by one or more copies of the `table_cell` or `table_header_cell` subtemplate

To see the default `table_row` template, run the following command:

```
$ skool2html.py -r Template:table_row
```

New in version 4.2.

7.9 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.

7.9.1 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*](#).

7.9.2 Bugfix modes

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

@ofix mode

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

For example:

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

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

`@ofix` mode is invoked by passing the `-f 1` option to [*skool2asm.py*](#).

@bfix mode

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

For example:


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

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

@rfix mode

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

For example:

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

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

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

7.9.3 ASM directives

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

@assemble

The @assemble directive controls whether assembly language instructions are converted into byte values for the purpose of populating the memory snapshot.

```
@assemble=N
```

- N is 1 to start converting at the next instruction, or 0 to stop

For example:

```
; The eight bytes of code in this routine are also used as UDG data.
; .
; #HTML(#UDG44919)
@assemble=1
c44919 LD DE,46572      ;
44922 CP 200           ;
44924 JP 45429          ;
@assemble=0
```

The @assemble=1 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.

Note that DEFB, DEFM, DEFS and DEFW statements are always converted into byte values and inserted into the memory snapshot; the @assemble directive is only required for assembly language instructions.

Version	Changes
5.0	New

@bfix

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

```
@bfix=INSTRUCTION
```

- INSTRUCTION is the replacement instruction

For example:

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

@bfix block directives

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

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

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

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

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

Typically, though, it is desirable to define a block that will be removed in @bfix mode right next to the block that 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
```

@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

@ignoreua

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

To apply the directive to an entry title:

```
@ignoreua
; Prepare data at 32768
c32768 LD A, (HL)
```

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

To apply the directive to an entry description:

```
; Prepare data in page 128
;
@ignoreua
; This routine operates on the data at 32768.
c49152 LD A, (HL)
```

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

To apply the directive to a 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 (during the rendering phase) 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 (during the rendering phase) 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, warnings would be printed (during the rendering phase) about the comment containing addresses (32522, 32600) that have not been converted to labels.

To apply the directive to a block end comment:

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

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

To apply the directive to an instruction-level comment:

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

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

Version	Changes
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 makes an instruction substitution in *@isub mode*.

```
@isub=INSTRUCTION
```

- INSTRUCTION is the replacement instruction

For example:

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

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

@isub block directives

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

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

@keep

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

For example:

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

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

@label

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

```
@label=LABEL
```

- LABEL is the label to apply

For example:

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

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

@nolabel

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

For example:

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

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

The output in `@bfix` mode will be:

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

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

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

@nowarn

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

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

For example:

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

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

For another example:

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

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

@ofix

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

```
@ofix=INSTRUCTION
```

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

For example:

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

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

@ofix block directives

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

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

@org

The @org directive inserts an ORG assembler directive.

```
@org=ADDRESS
```

- ADDRESS is the ORG address

@rem

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

```
@rem=COMMENT
```

- COMMENT is a suitably illuminating comment

For example:

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

Version	Changes
2.4	The = is required

@rfix block directives

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

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

@rsub

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

```
@rsub=INSTRUCTION
```

- INSTRUCTION is the replacement instruction

For example:

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

@rsub block directives

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

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

@set

The @set directive sets a property on the ASM writer.

```
@set-name=value
```

- name is the property name
- value is the property value

@set directives 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)
- warnings - 1 to print any warnings that are produced while writing ASM output (after parsing the skool file), or 0 to suppress them (default: 1)
- wrap-column-width-min - the minimum width of a wrappable table column (default: 10)

For example:

```
@set-bullet=+
```

This @set directive sets the bullet character to '+'.

Version	Changes
3.4	Added the <i>handle-unsupported-macros</i> and <i>wrap-column-width-min</i> properties
3.3.1	Added the <i>comment-width-min</i> , <i>indent</i> , <i>instruction-width</i> , <i>label-colons</i> , <i>line-width</i> and <i>warnings</i> properties
3.2	New

@ssub

The @ssub directive makes an instruction substitution in @ssub mode.

```
@ssub=INSTRUCTION
```

- INSTRUCTION is the replacement instruction

For example:

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


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

@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

Developer reference

8.1 Extending SkoolKit

8.1.1 Extension modules

While creating a disassembly of a game, you may find that SkoolKit’s suite of *skool macros* is inadequate for certain tasks. For example, the game might have large tile-based sprites that you want to create images of for the HTML disassembly, and composing long *#UDGARRAY* macros for them would be too tedious. Or you might want to insert a timestamp 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/python2.7/dist-packages/skoolkit
```

(The package directory may be different on your system.) With *game.py* in place, add the following line to the *[Config]* section of your disassembly’s *ref* file:

```
HtmlWriterClass=skoolkit.game.GameHtmlWriter
```

If you don’t have a *ref* file yet, create one (ideally named *game.ref*, assuming the *skool* file is *game.skool*); if the *ref* file doesn’t have a *[Config]* section yet, add one.

Now whenever *skool2html.py* is run on your *skool* file (or *ref* file), SkoolKit will use the `GameHtmlWriter` class instead of the core `HtmlWriter` class.

To get *skool2asm.py* to use `GameAsmWriter` instead of the core `AsmWriter` class when it's run on your *skool* file, add the following `@writer` ASM directive somewhere after the `@start` directive, and before the `@end` directive (if there is one):

```
@writer=skoolkit.game.GameAsmWriter
```

The *skoolkit* package directory is a reasonable place for an extension module, but it could be placed in another package, or somewhere else as a standalone module. For example, if you wanted to keep a standalone extension module 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.

8.1.2 #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.skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def sprite(self, cwd, sprite_id, fname):
        img_path = self.image_path(fname)
        if self.need_image(img_path):
            udgs = self.build_sprite(sprite_id)
            self.write_image(img_path, udgs)
        return self.img_element(cwd, img_path)
```

With this method (and an appropriate implementation of the *build_sprite* method) in place, it's possible to use a `#CALL` macro like this:

```
#UDGTABLE
{ #CALL:sprite(3,jumping) }
{ Sprite 3 (jumping) }
TABLE#
```

Adding a #CALL method to the AsmWriter subclass is equally simple. The timestamp-creating method (let's call it *timestamp*) would look something like this:

```
import time
from 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.

8.1.3 Skool macros

Another way to add a custom method is to implement it as a skool macro. The main differences between a skool macro and a #CALL method are:

- a #CALL macro's parameters are automatically evaluated and passed to the #CALL method; a skool macro's parameters must be parsed and evaluated manually (typically by using one or more of the *macro-parsing utility functions*)
- every optional parameter in a skool macro can be assigned a default value if omitted; in a #CALL method, only the optional arguments at the end can be assigned default values if omitted, whereas any others are set to *None*
- numeric parameters in a #CALL macro are automatically converted to numbers before being passed to the #CALL method; no automatic conversion is done on the parameters of a skool macro

In summary: a #CALL method is generally simpler to implement than a skool macro, but skool macros are more flexible.

Implementing a skool macro is done by adding a method named *expand_macroname* to the HtmlWriter or AsmWriter subclass in the extension module. So, to implement a #SPRITE or #TIMESTAMP macro, we would add a method named *expand_sprite* or *expand_timestamp*.

A skool macro method must accept either two or three parameters, depending on whether it is implemented on a subclass of AsmWriter or HtmlWriter:

- *text* - the text that contains the skool macro
- *index* - the index of the character after the last character of the macro name (that is, where to start looking for the macro's parameters)
- *cwd* - the current working directory of the file from which the macro is being executed; this parameter must be supported by skool macro methods on an HtmlWriter subclass

A skool macro method must return a 2-tuple of the form (*end*, *string*), where *end* is the index of the character after the last character of the macro's parameter string, and *string* is the HTML or text to which the macro will be expanded.

The *expand_sprite* method on GameHtmlWriter may therefore look something like this:

```
from skoolkit.skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    # #SPRITEspriteId[{x,y,width,height}](fname)
    def expand_sprite(self, text, index, cwd):
        end, img_path, crop_rect, sprite_id = self.parse_image_params(text, index, 1)
```

```
if self.need_image(img_path):
    udgs = self.build_sprite(sprite_id)
    self.write_image(img_path, udgs, crop_rect)
return end, self.img_element(cwd, img_path)
```

With this method (and an appropriate implementation of the *build_sprite* method) in place, the `#SPRITE` macro might be used like this:

```
#UDGTABLE
{ #SPRITE3(jumping) }
{ Sprite 3 (jumping) }
TABLE#
```

The *expand_timestamp* method on *GameAsmWriter* would look something like this:

```
import time
from 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")
```

Note that if the string part of the return value of a skool macro method contains skool macros, then they will be expanded.

8.1.4 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=())`

Parse a string of comma-separated integer parameters. The string will be parsed until either the end is reached, or an invalid character is encountered. The set of valid characters consists of the comma, '\$', the digits 0-9, and the letters A-F and a-f; if *names* is not empty, the set of valid characters also includes '=' and the letters g-z.

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.

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 4.0: Added the *names* parameter and support for keyword arguments; *index* defaults to 0.

`skoolkit.skoolmacro.parse_params(text, index, p_text=None, chars=',', except_chars=' ', only_chars='')`

Parse a string of the form `params [(p_text)]`. The parameter string *params* will be parsed until either the

end is reached, or an invalid character is encountered. The default set of valid characters consists of '\$', '#', the digits 0-9, and the letters A-Z and a-z.

Parameters

- **text** – The text to parse.
- **index** – The index at which to start parsing.
- **p_text** – The default value to use for text found in parentheses.
- **chars** – Characters to consider valid in addition to those in the default set.
- **except_chars** – If not empty, all characters except those in this string are considered valid.
- **only_chars** – If not empty, only the characters in this string are considered valid.

Returns

A 3-tuple of the form (end, params, p_text), where:

- end is the index at which parsing terminated
- params is the parameter string
- p_text is the text found in parentheses (if any)

Changed in version 3.6: Added the *except_chars* and *only_chars* parameters.

HtmlWriter also provides a method for parsing the parameters of an image-creating skool macro.

```
HtmlWriter.parse_image_params(text, index, num=0, defaults=(), path_id='UDGImagePath',
                             fname='', chars='', ints=None, names=(), frame=False,
                             alt=False)
```

Parse a string of the form:

```
params[{x,y,width,height}][ (fname[*frame][|alt]) ]
```

The parameter string `params` may contain comma-separated values and is parsed until either the end is reached, or an invalid character is encountered. The default set of valid characters consists of the comma, '\$', the digits 0-9, and the letters A-F and a-f; if *names* is not empty, the set of valid characters also includes '=' and the letters g-z.

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.
- **path_id** – The ID of the target directory for the image file (as defined in the [\[Paths\]](#) section of the *ref* file).
- **fname** – The default base name of the image file.
- **chars** – Characters to consider valid in addition to those in the default set.
- **ints** – A list of the indexes (0-based) of the parameters that must evaluate to an integer; if *None*, every parameter must evaluate to an integer.
- **names** – The names of the parameters; if not empty, keyword arguments in the parameter string `params` are parsed.

- **frame** – Whether to parse and return the frame name.
- **alt** – Whether to parse and return the alt text.

Returns

A list of the form `[end, image_path, frame, alt, crop_rect, value1, value2...]`, where:

- `end` is the index at which parsing terminated
- `image_path` is either the full path of the image file (relative to the root directory of the disassembly) or `fname` (if `path_id` is blank or `None`)
- `frame` is the frame name (present only if `frame` is `True`)
- `alt` is the alt text (present only if `alt` is `True`)
- `crop_rect` is `(x, y, width, height)`
- `value1, value2` etc. are the parameter values

Changed in version 3.6: If `path_id` is blank or `None`, `image_path` is equal to `fname`; added the `ints` parameter.

Changed in version 4.0: Added the `names` parameter and support for keyword arguments.

Changed in version 4.2: Added the `frame` and `alt` parameters.

8.1.5 Expanding skool macros

Both `HtmlWriter` and `AsmWriter` provide methods for expanding skool macros. These are useful for basing a `#CALL` method or custom macro on an existing macro.

`HtmlWriter.expand(text, cwd)`

Return `text` with skool macros expanded. `cwd` is the current working directory, which is required by macros that create images or hyperlinks.

`AsmWriter.expand(text)`

Return `text` with skool macros expanded.

8.1.6 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)`

Return the contents of a `ref` file section.

Parameters

- **section_name** – The section name.
- **paragraphs** – If `True`, return the contents as a list of paragraphs.
- **lines** – If `True`, return the contents (or each paragraph) as a list of lines; otherwise return the contents (or each paragraph) as a single string.

`HtmlWriter.get_sections(section_type, paragraphs=False, lines=False)`

Return a list of 2-tuples of the form `(suffix, contents)` or 3-tuples of the form `(infix, suffix, contents)` derived from `ref` file sections whose names start with `section_type` followed by a colon. `suffix` is the part of the section name that follows either the first colon (when there is only one) or the second colon

(when there is more than one); *infix* is the part of the section name between the first and second colons (when there is more than one).

Parameters

- **section_type** – The section name prefix.
- **paragraphs** – If *True*, return the contents of each section as a list of paragraphs.
- **lines** – If *True*, return the contents (or each paragraph) of each section as a list of lines; otherwise return the contents (or each paragraph) as a single string.

`HtmlWriter.get_dictionary(section_name)`

Return a dictionary built from the contents of a *ref* file section. Each line in the section should be of the form *X=Y*.

`HtmlWriter.get_dictionaries(section_type)`

Return a list of 2-tuples of the form (*suffix*, *dict*) derived from *ref* file sections whose names start with *section_type* followed by a colon. *suffix* is the part of the section name that follows the first colon, and *dict* is a dictionary built from the contents of that section; each line in the section should be of the form *X=Y*.

8.1.7 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.

A simple `#PEEK` macro that expands to the value of the byte at a given address might be implemented by using *snapshot* like this:

```
from skoolkit.skoolhtml import HtmlWriter
from skoolkit.skoolasm import AsmWriter
from skoolkit.skoolmacro import parse_ints

class GameHtmlWriter(HtmlWriter):
    # #PEEKaddress
    def expand_peek(self, text, index, cwd):
        end, address = parse_ints(text, index, 1)
        return end, str(self.snapshot[address])

class GameAsmWriter(AsmWriter):
    # #PEEKaddress
    def expand_peek(self, text, index):
        end, address = parse_ints(text, index, 1)
        return end, str(self.snapshot[address])
```

`HtmlWriter` also provides some methods for saving and restoring memory snapshots, which can be useful for temporarily changing graphic data or the contents of data tables. These methods are described below.

`HtmlWriter.push_snapshot(name='')`

Save the current memory snapshot for later retrieval (by `pop_snapshot()`), and put a copy in its place.

Parameters *name* – An optional name for the snapshot.

`HtmlWriter.pop_snapshot()`

Discard the current memory snapshot and replace it with the one that was most recently saved (by `push_snapshot()`).

`HtmlWriter.get_snapshot_name()`

Return the name of the current memory snapshot.

8.1.8 Graphics

If you are going to implement custom image-creating #CALL methods or skool macros, you will need to make use of the `skoolkit.skoolhtml.Udg` class.

The `Udg` class represents an 8x8 graphic (8 bytes) with a single attribute byte, and an optional mask.

class `skoolkit.skoolhtml.Udg` (*attr*, *data*, *mask=None*)
Initialise the UDG.

Parameters

- **attr** – The attribute byte.
- **data** – The graphic data (sequence of 8 bytes).
- **mask** – The mask data (sequence of 8 bytes).

A simple #INVERSE macro that creates an inverse image of a UDG might be implemented like this:

```
from skoolkit.skoolhtml import HtmlWriter, Udg
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)
        img_path = self.image_path('inverse{0}_{1}'.format(address, attr))
        if self.need_image(img_path):
            udg_data = [b ^ 255 for b in self.snapshot[address:address + 8]]
            udg = Udg(attr, udg_data)
            self.write_image(img_path, [[udg]])
        return end, self.img_element(cwd, img_path)
```

The `Udg` class provides two methods for manipulating an 8x8 graphic: *flip* and *rotate*.

`Udg.flip` (*flip=1*)
Flip the UDG.

Parameters **flip** – 1 to flip horizontally, 2 to flip vertically, or 3 to flip horizontally and vertically.

`Udg.rotate` (*rotate=1*)
Rotate the UDG 90 degrees clockwise.

Parameters **rotate** – The number of rotations to perform.

If you are going to implement #CALL methods or skool macros that create animated images, you will need to make use of the `skoolkit.skoolhtml.Frame` class.

The `Frame` class represents a single frame of an animated image.

class `skoolkit.skoolhtml.Frame` (*udgs*, *scale=1*, *mask=0*, *x=0*, *y=0*, *width=None*, *height=None*, *delay=32*)
Create a frame of an animated image.

Parameters

- **udgs** – The two-dimensional array of tiles (instances of `Udg`) from which to build the frame.
- **scale** – The scale of the frame.
- **mask** – 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.

New in version 3.6.

Changed in version 4.0: The *mask* parameter specifies the type of mask to apply (see [Masks](#)).

HtmlWriter provides the following image-related convenience methods.

`HtmlWriter.image_path(fname, path_id='UDGImagePath')`

Return the full path of an image file relative to the root directory of the disassembly. If *fname* does not end with *'png'* or *'gif'*, an appropriate suffix will be appended (depending on the default image format). If *fname* starts with a *'/'*, it will be removed and the remainder returned (in which case *path_id* is ignored). If *fname* is blank, *None* is returned.

Parameters

- **fname** – The name of the image file.
- **path_id** – The ID of the target directory (as defined in the [\[Paths\]](#) section of the *ref* file).

`HtmlWriter.need_image(image_path)`

Return whether an image file needs to be created. This will be true only if the file doesn't already exist, or all images are being rebuilt. Well-behaved image-creating methods will call this to check whether an image file needs to be written, and thus avoid building an image when it is not necessary.

Parameters **image_path** – The full path of the image file relative to the root directory of the disassembly.

`HtmlWriter.write_image(image_path, udgs, crop_rect=(), scale=2, mask=0)`

Create an image and write it to a file.

Parameters

- **image_path** – The full path of the file to which to write the image (relative to the root directory of the disassembly).
- **udgs** – The two-dimensional array of tiles (instances of [Udg](#)) from which to build the image.
- **crop_rect** – The cropping rectangle, (*x*, *y*, *width*, *height*), where *x* and *y* are the x- and y-coordinates of the top-left pixel to include in the final image, and *width* and *height* are the width and height of the final image.
- **scale** – The scale of the image.
- **mask** – The type of mask to apply to the tiles: 0 (no mask), 1 (OR-AND mask), or 2 (AND-OR mask).

Changed in version 4.0: The *mask* parameter specifies the type of mask to apply (see [Masks](#)).

`HtmlWriter.write_animated_image(image_path, frames)`

Create an animated image and write it to a file.

Parameters

- **image_path** – The full path of the file to which to write the image (relative to the root directory of the disassembly).
- **frames** – A list of the frames (instances of *Frame*) from which to build the image.

New in version 3.6.

`HtmlWriter.img_element(cwd, image_path, alt=None)`

Return an `` element for an image file.

Parameters

- **cwd** – The current working directory (from which the relative path of the image file will be computed).
- **image_path** – The full path of the image file relative to the root directory of the disassembly.
- **alt** – The alt text to use for the image; if *None*, the base name of the image file (with the ‘.png’ or ‘.gif’ suffix removed) will be used.

`HtmlWriter.screenshot(x=0, y=0, w=32, h=24, df_addr=16384, af_addr=22528)`

Return a two-dimensional array of tiles (instances of *Udg*) built from the display file and attribute file of the current memory snapshot.

Parameters

- **x** – The x-coordinate of the top-left tile to include (0-31).
- **y** – The y-coordinate of the top-left tile to include (0-23).
- **w** – The width of the array (in tiles).
- **h** – The height of the array (in tiles).
- **df_addr** – The display file address to use.
- **af_addr** – The attribute file address to use.

`HtmlWriter.flip_udgs(udgs, flip=1)`

Flip a 2D array of UDGs (instances of *Udg*).

Parameters

- **udgs** – The array of UDGs.
- **flip** – 1 to flip horizontally, 2 to flip vertically, or 3 to flip horizontally and vertically.

`HtmlWriter.rotate_udgs(udgs, rotate=1)`

Rotate a 2D array of UDGs (instances of *Udg*) 90 degrees clockwise.

Parameters

- **udgs** – The array of UDGs.
- **rotate** – The number of rotations to perform.

8.1.9 HtmlWriter initialisation

If your `HtmlWriter` subclass needs to perform some initialisation tasks, such as creating instance variables, or parsing *ref* file sections, the place to do that is the *init()* method.

`HtmlWriter.init()`

Perform post-initialisation operations. This method is called after `__init__()` has completed. By default the method does nothing, but subclasses may override it.

For example:

```
from skoolkit.skoolhtml import HtmlWriter

class GameHtmlWriter(HtmlWriter):
    def init(self):
        # Get character names from the ref file
        self.characters = self.get_dictionary('Characters')
```


E

expand() (skoolkit.skoolasm.AsmWriter method), 116
expand() (skoolkit.skoolhtml.HtmlWriter method), 116

F

flip() (skoolkit.skoolhtml.Udg method), 118
flip_udgs() (skoolkit.skoolhtml.HtmlWriter method), 120
Frame (class in skoolkit.skoolhtml), 118

G

get_dictionaries() (skoolkit.skoolhtml.HtmlWriter method), 117
get_dictionary() (skoolkit.skoolhtml.HtmlWriter method), 117
get_section() (skoolkit.skoolhtml.HtmlWriter method), 116
get_sections() (skoolkit.skoolhtml.HtmlWriter method), 116
get_snapshot_name() (skoolkit.skoolhtml.HtmlWriter method), 117

I

image_path() (skoolkit.skoolhtml.HtmlWriter method), 119
img_element() (skoolkit.skoolhtml.HtmlWriter method), 120
init() (skoolkit.skoolhtml.HtmlWriter method), 120

N

need_image() (skoolkit.skoolhtml.HtmlWriter method), 119

P

parse_image_params() (skoolkit.skoolhtml.HtmlWriter method), 115
parse_ints() (in module skoolkit.skoolmacro), 114
parse_params() (in module skoolkit.skoolmacro), 114
pop_snapshot() (skoolkit.skoolhtml.HtmlWriter method), 117

push_snapshot() (skoolkit.skoolhtml.HtmlWriter method), 117

R

rotate() (skoolkit.skoolhtml.Udg method), 118
rotate_udgs() (skoolkit.skoolhtml.HtmlWriter method), 120

S

screenshot() (skoolkit.skoolhtml.HtmlWriter method), 120

U

Udg (class in skoolkit.skoolhtml), 118

W

write_animated_image() (skoolkit.skoolhtml.HtmlWriter method), 119
write_image() (skoolkit.skoolhtml.HtmlWriter method), 119