Updating the Online SuperBASIC Manual

Release 0.1.1

Norman Dunbar

Nov 18, 2023

CONTENTS

Contents:

CHAPTER

ONE

PART ONE - INTRODUCTION

This part of the 'book' gives you an introduction to the process of updating the manual.

There's not much to it really, you have a number of options open to you and each one has a little more involvement that the option that came before.

Interested? Read on …

Contents of Part 1:

1.1 Introduction

Greetings!

If you are reading this then you are either an insomniac like me, or you have a mild interest in maybe helping to update the SuperBASIC Online Manual.

Now, you might think that it will be difficult to do so, given all that fancy HTML and the like, but nothing could be further from the truth. The manual is written in a special form of plain text and converted to the glorious format that you see online, and indeed, can download for your own offline enjoyment.

The special format is known as ReStructuredText (yes, all one word!) and is pretty simple to use and understand, for the most part. And, as I've deliberately avoided using the more complex or esoteric features of ReStructuredText in the original files, you don't have to worry about them!

1.2 How Can I Help?

There are a few different ways that you can help:

Just Email!
: The easiest "fix" is to get someone else to do it for you! Just send an email, let's say via the QL Users list where whatever is "wrong" can be discussed prior to fixing, if anything needs fixing. Obviously, you should provide as much detail as possible about:

  - What you think is wrong;
  - Where you think it is wrong;
  - What you think it should be;
  - Why you think this.

Log Bugs
Bugs can be logged against the source repository on GitHub using the issues tracker. As the owner of the repository, I will be informed and will be able to do something to fix them. For the sake of argument, a bug in this context is anything "wrong" with the Manual as it is. You could include:

- Spelling errors;
- Bad grammar;
- Stuff that is downright incorrect;
- Updates for new software which changes or adds to the existing commands;
- New commands in the documented toolkits and/or emulators;
- And so on.

Fix Bugs
You can, if you wish, fix any bugs - but check the issues log first to be sure that they have not been reported and maybe are in progress. Just in case.

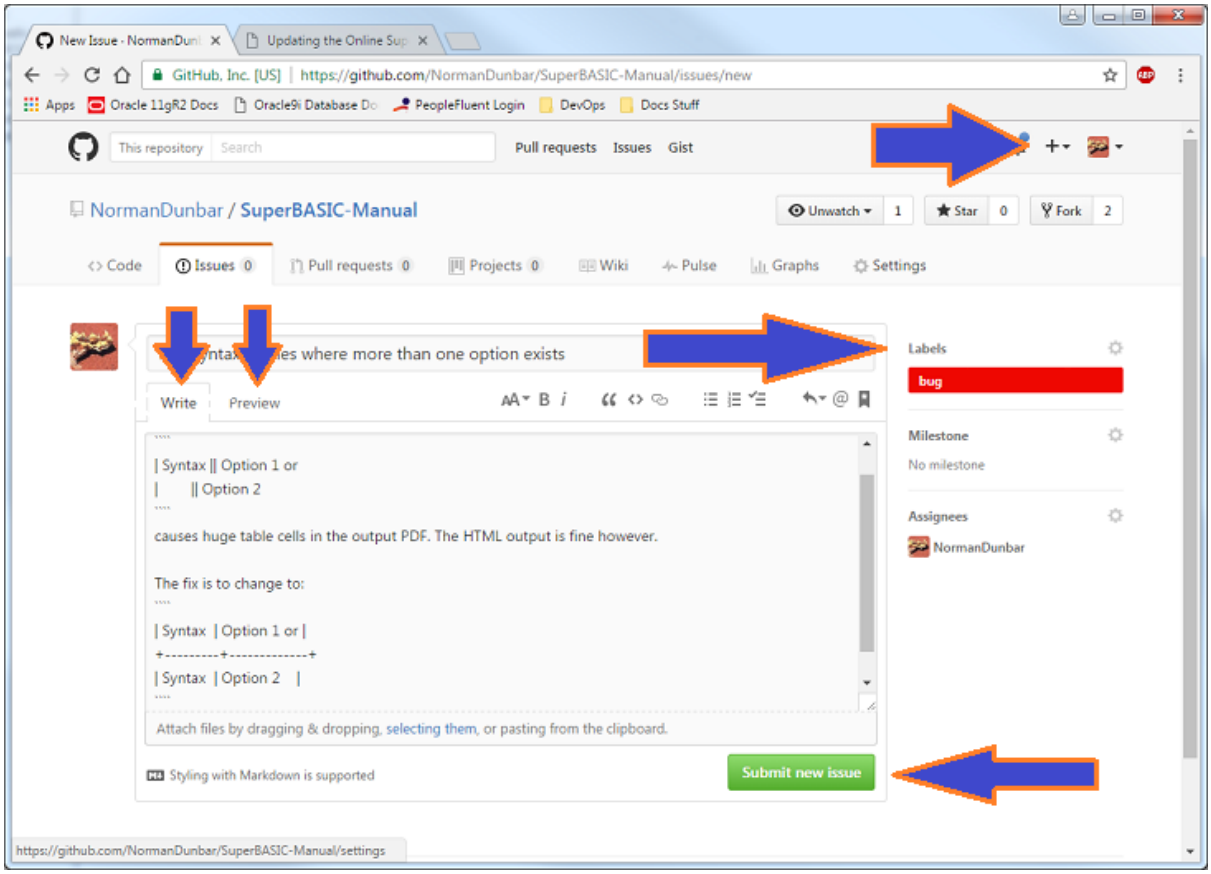
How to fix a big? See below for details, but fork my repository, edit the files as appropriate and commit them to your own fork then issue a "Pull Request" to me to merge your fixes into the main repository.

1.3 Log Bugs on GitHub

The steps to log a bug are simple:

- Go to The SuperBASIC GitHub Repository.
- Click on the '+' over on the right side of the screen and select "New Issue".

On the screen that appears:



- Fill in a descriptive title.
- Fill in a good description of the problem as you see it. If possible, suggest fixes, changes etc. Give examples, if possible. Markdown formatting is allowed in the description - if you know how to use it, that is.
- Over on the right, there's a "gear wheel" icon next to "Labels". Click it and select a category. Bug, Enhancement or Question are good ones. Click as many as you feel necessary and click the gear wheel again.
- Go to the "Preview" tab and check it all looks right. Go back to the "Write" tab and make fixes if necessary.
- Click "Submit new issue".

1.4 Fix Bugs Yourself

If you have the desire to fix things, the process is quite simple too but it does require the use of a Windows, Mac or Linux computer - unless you are attempting to edit files actually in the browser - and the installation of the Git software.

- Fork The SuperBASIC GitHub Repository. You will need a GitHub account for this, but it's free, there's no spam and you can have as many or as few repositories as you like.
- Once forked, you can clone it to your computer, or edit things online, but to be honest, do it locally as it's better that way.
- Commit your changes.

- Push your changes back to your repository on GitHub.
- Raise a "pull request" to me, to examine and merge your changes into the main repository.
- You can, if you wish, delete your fork of the repository, or keep it for later.

The steps above are covered in their own section later on in this "book".

In order to actually do your own fixes, or adding new details to the manual, you will need some software, and sadly, our beloved QL and the various emulator children it has begat, are of no use I'm afraid. We need Linux, Windows and perhaps a Mac.

1.5 Software Required

I did all my stuff on Linux, well most of it, I did install everything on a Windows 7 box, just to see if it was any good. Surprisingly enough it was, but then again, that's the value of Open Source stuff. It *usually* works!

To fix things, you only need the following software:

- A decent text editor. *Not* a word processor like Word or Open/Libre Office. On Windows I would advise `Notepad++`. On Linux, I use the default text editor in Linux Mint 18 which is `xed` but any decent one will do. It helps if it has or understands ReStructuredText highlighting, but this isn't essential.
- Git. You need git to be able to `pull` and `push` changes from and to the repository. You can download the entire repository as a zip file, but any changes you make can't then be pushed back.

So far so not too exciting, however, if you intend to add any new details to the manual, or it you want to build a local copy to test your changes, you will need more:

- Python - because Sphinx-doc needs Python to work.
- Sphinx-doc - which takes the ReStructuredText that we create and converts it into other formats.

That's the main software list covered in brief, there are some optional software that you might find useful too:

- Pandoc - this is literally the Swiss Army Knife of document conversion. I use it at work to convert ReStructuredText into Word's docx format because I'm sick and tired of Word deciding that my desired formatting is not what it thinks I should have, or that two consecutive bullet points should have totally different bullets or indents or bullet sizes! Word drives me absolutely nuts!
- A C++ Compiler. Useful if you ever have the need to build any of the utilities in the Tools folder.

1.6 How and Where to Get the Software

1.6.1 The Editor

Head on over to https://notepad-plus-plus.org/ and download the latest version. Install it and that's all there is to it. This is probably one of the finest text editors I've used on any operating system and I cannot recommend it highly enough.

1.6.2 Git

Git is needed to allow the SuperBASIC-Manual repository to be updated. The download details, for Windows users, can be found at https://git-for-windows.github.io/. If you are on Linux then your software repository will have an up to date versions which you should install using the appropriate package manager.

Mac users simply need to head on over to https://git-scm.com/download/mac where you will get all the details you need.

1.6.3 Sphinx-Doc

Overall, the biggie of the conversion process is a tool called Sphinx, or Sphinx-doc to avoid searching Google for statues of lion beasts from Ancient Egypt! ;-)

Head over to http://www.sphinx-doc.org/en/stable/install.html where you will find details of installing just about everything you need for Windows, Linux and Macs to run the Sphinx documentation builder.

Sphinx needs Python and it needs a version of Python that is 2.7 or higher (at the time of writing.)

1.6.4 Python

Windows and Mac users should go to https://www.python.org/downloads/, the main download site for Python. Look for a link to the Python download for your Operating System. Linux users, just install Python from your software repository in the usual manner.

Version 2.7 is suitable, but 3.x is probably better in the long run if you plan on doing more Python work on Windows.

Run the installer and install the package somewhere suitable. Make sure you update your path to suit, or nothing will work!

1.6.5 Pip

Once Python is installed, install pip:

```
python get-pip.py
```

Simple stuff. pip is the Python Installer Program and is useful in installing Python utilities. We need it to install Sphinx.

1.6.6 Sphinx

Once Pip is installed, install Sphinx:

```
pip install sphinx
```

Once this is complete, make sure it worked by running the command:

```
sphinx-build --version
```

Which will show you that your PATH etc are set up well enough to build things. You should see something like:

```
Sphinx (sphinx-build) 1.4.6
```

1.6.7 Sphinx RTD Theme

Sphinx used to come with a Read The Docs theme installed automatically, but more recent versions don't include it for some unknown reason. Not to worry, if you try to build docs using that theme, you will be prompted to install it as follows:

```
pip install sphinx_rtd_theme
```

1.7 Optional Software

The software above is all you need to be able to run the *building* of the docs, however, the following might prove useful in other work you might want to do later.

1.7.1 Pandoc

Pandoc is used to convert the cleaned up HTML into ReStructuredText format. This can also convert numerous input formats into many different output formats, it is the *Swiss Army Knife* of document conversions.

Head on over to http://pandoc.org/installing.html where you will find instructions and downloads for installing Pandoc.

Pandoc is a command-line tool to convert between different document formats. It can read in the following formats:

- Commonmark, docbook, docx, epub, haddock, html, json, latex, markdown, markdown_github, markdown_mmd, markdown_phpextra, markdown_strict, mediawiki, native, odt, opml, org, rst, t2t, textile & twiki.

And it can convert those to any of the following:

- Asciidoc, beamer, commonmark, context, docbook, docbook5, docx, dokuwiki, dzslides, epub, epub3, fb2, haddock, html, html5, icml, json, latex, man, markdown, markdown_github, markdown_mmd, markdown_phpextra, markdown_strict, mediawiki, native, odt, opendocument, opml, org, plain, revealjs, rst, rtf, s5, slideous, slidy, tei, texinfo, textile & zimwiki.

As I mentioned, I use it at work with something resembling the following command:

```
pandoc -f rst -t docx --toc --toc-depth 3 --reference-docx pandoc_reference.
↪docx -o RMANRestore.docx RMANRestore.rst
```

That reads my `RMANRestore.rst` text file, references the file `pandoc_reference.docx` to determine the formatting of the various styles I need, and writes out `RMANRestore.docx` as a properly formatted Word document.

1.7.2 A C++ Compiler

On Linux, just about everything uses the G++ compiler. That's definitely what I use, however, Windows is the proverbial nightmare. Visual Studio Express Edition is freely available for download and use, but it's no longer proper C++ as Microsoft went over to their "java-like" .net nonsense years ago. Even their C++ compiler cannot compile proper standard C++ code - does that sound familiar? Microsoft and standards not matching up?

The best ever compiler on Windows was always Borland C++ and many years ago, they gave away the 5.5 version to anyone who wanted it. I used it happily for years on Windows. Sadly, Borland sold out to Embarcadero, but the 5.5 version is still available and still free.

However, Embarcadero recently started giving away version 10 of the compiler which is right up to date. You can get it at https://www.embarcadero.com/free-tools then follow the links to the C++ compiler.

You will need to create an account, but this only causes a few special offers in your inbox, some of them useful!

If anyone is interested, this is what Embarcadero have to say about their free tool:

> *This free download of the C++ Compiler for C++Builder includes C++11 language support, the Dinkumware STL (Standard Template Library) framework, and the complete Embarcadero C/C++ Runtime Library (RTL). In this free version, you'll also find a number of C/C++ command line tools—such as the high performance linker and resource compiler.*

1.7.3 All Done!

That's (about) it! Nothing *too* excessive now, was it? Too much software perhaps? Well, nobody said that it was going to be easy. But remember, once all this is installed, you have a very good and useful document production system which can be used for many things, not just updating the SuperBASIC Manual.

CHAPTER

TWO

PART TWO - UPDATING THE MANUAL

This part of the 'book' gives you a detailed walk through of the process of updating the manual.

Contents of Part 2:

2.1 Obtaining the Source

2.1.1 Free Git Book

You might like to download and read the *Pro Git* book, by Scott Chacon & Ben Straub. This is a book published by Apress but which is given away for free on the web at https://git-scm.com/book/en/v2 - various formats are available.

2.1.2 Introduction

Before you can start making changes to the SuperBASIC manual, you must:

- `Fork` the repository on GitHub;
- `Clone` the fork down to your computer;
- Configure git;
- Make your changes or additions;
- Commit the changes to your local repository;
- `Push` the changes to your forked repository;
- Create a `pull` request to me;

At this point, after I've merged your changes, you can either:

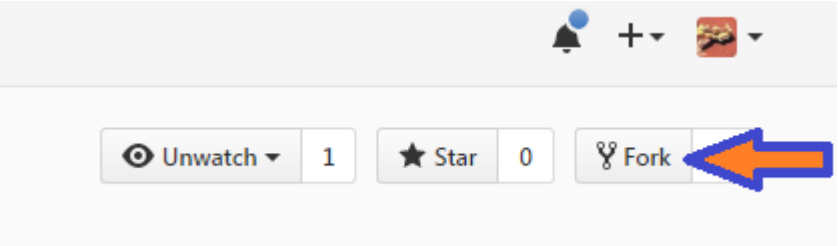
- Delete your fork; or;
- Keep it, and keep it in sync with mine.

Note: The following assumes that you are fully up to date with installing all the required software detailed in the previous part of the book.

2.1.3 Forking the Repository

Forking the main repository is simple.

- Go to the repository on GitHub.
- Click on the `Fork` button at the top of the screen, on the right:



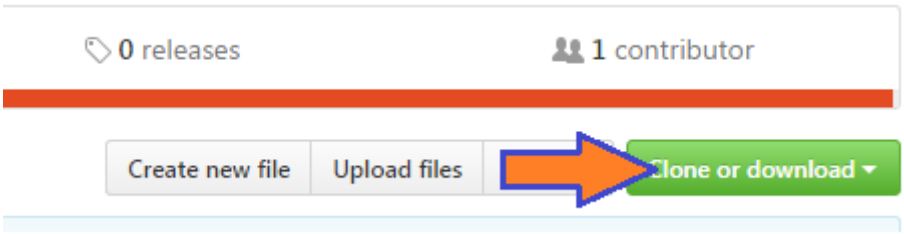
- If you are a member of one or more organisations, as well as being an individual, you will be prompted to select a location. Choose whichever one applies.
- That's it. After a while, the repository will show up in your list of repositories.

You are now ready to `pull` the source code from your repository to your local computer, ready to edit.

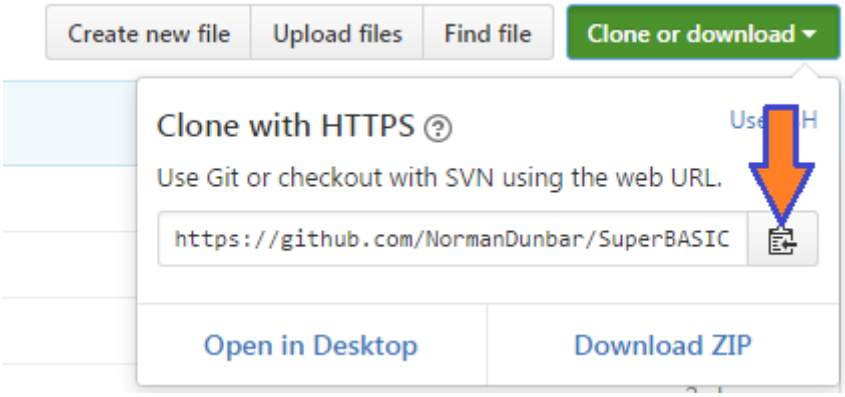
2.1.4 Pull the Source

In your favourite browser, alternatively you could use Internet Explorer, go to https://github.com/<your-name>/SuperBASIC-Manual (letter case *is* significant here by the way). This is your repository page.

On the right side, in a natty green colour, there's a button with the text "clone or download" written on it.



Click it. A small dialogue will open up. The URL for your repository is preset in the text box, and the button on the right will copy the URL to the clipboard. Click it.



Change form the browser to your command line session. Make sure that you are in a location where you are happy to create a new folder to download the source files to. I use a top level `SourceCode` folder, so

on my computer, this would be where I need to be:

```
cd SourceCode
```

Now, type the following command:

```
git clone <paste>
```

Where "<paste>" means that you should paste in the URL you copied from GitHub. The final command should look something like this:

```
git clone https://github.com/<your-name>/SuperBASIC-Manual.git
```

Obviously, <your-name> would be as appropriate!

After a few seconds or minutes - depending on your internet speed - you should find a new folder created, with the name `SuperBASIC-Manual`.

2.1.5 Configure Git

In your command line session, change into your new folder:

```
cd SuperBASIC-Manual
```

The following configuration options are pretty much mandatory. They set the defaults for your username and email address for all your git work, now and in the future.

```
git config --global user.name "Your Name"
git config --global user.email YourName@YourDomain.co.uk
```

The `--global` flag means exactly that, they apply to all of git, not just this repository. You should also configure your default editor, for example, on Windows, to use Notepad++:

```
git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -
↪multiInst -nosession"
```

Alternatively, if you are on a 64 bit Windows system:

```
git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.
↪exe' -multiInst
```

On Linux, this works:

```
git config --global core-editor xed
```

Line feeds are a nice little "gotcha" to watch out for. Windows editors *can* silently convert Linux/Mac/QL line feeds (LF) into Carriage Return plus Line Feed characters (CRLF). This is, to put it mildly, rather irritating!

If you are on Windows, set the following configuration option so that your editor doesn't mess things up for everyone else when you check in a file:

```
git config --global core.autocrlf true
```

When you check out a file, git will convert the LF to CRLF automatically for you. If you are using notepad++ as your editor then the above is not really required as that editor can happily cope with LF as well as CRLF. Just be sure that you don't have it set up to save everything in CRLF mode. (Settings->Preferences->New Document)

Mac and Linux users should configure the following option instead:

```
git config --global core.autocrlf input
```

This will ensure that if a Windows file with CRLF is checked out by you, it will be fixed when you check it back in.

Hopefully now, everyone is happy.

2.1.6 Edit the Source

Very importantly, you should *never* work in the master branch. I've set up the repository with a *working* branch, and this is where the work should be done. When completely finished, the working branch can be merged back into the master branch.

Important: We never, ever mess up the master branch!

So, in a command line session, we do the following:

```
cd SourceCode\SuperBASIC-Manual
git branch
```

Git will respond with:

```
* master
```

So we need to be well away from that particular branch:

```
git checkout working
git branch
```

And git should respond with:

```
  master
* working
```

The working branch is where we need to be, always, so check often!

Editing the source is *mostly* done in plain text. The benefit of ReStructuredText is that it is *mostly* just typing and no special formatting is required. However, please see the section on ReStructuredText in Part 3, for details of the features we use for the manual.

The basic design of a keyword's entry in the manual is as follows, please try to stick with Rich's original format as much as possible, as the following template entry attempts to demonstrate:

```
..  _create-keyword:

```

(continues on next page)

(continued from previous page)

```
CREATE_Keyword
==============

+----------+-------------------------------------------------------------------
↪---+
| Syntax   | CREATE_keyword title$, #channel [ Some_text$] [ more_text$]     ␣
↪    |
+----------+-------------------------------------------------------------------
↪---+
| Location | QL ROM, Toolkit II, etc                                          ␣
↪    |
+----------+-------------------------------------------------------------------
↪---+

At this point here, there will be a number of paragraphs describing the␣
↪command,
what it does, how it does it - if necessary - and so on. Very, very brief␣
↪examples
of it's use may be found here.

There may be minimal examples of the calling conventions, where these need␣
↪special
attention, delimiters or whatever, not covered in the ``Syntax`` table␣
↪entries above.

**Example**
Please use the letter case demonstrated here for the section title -␣
↪**Example**.

The example section has a bold title, and gives a more complete example of␣
↪the keyword's
usage in context.

::

    1000 REMark Demonstration of the (fictitious) CREATE_keyword command.
    1005 :
    1010 OPEN #3, "con_"
    1015 CREATE_keyword "OPEN", #3
    1020 CREATE_keyword "TITLE", #3, KeyWord$
    1025 CREATE_keyword "SYNTAX", #3, Syntax_1$, Syntax_2$
    1030 CREATE_keyword "LOCATION", #3, "QL ROM"
    1035 CREATE_keyword "DESCRIPTION", #3, Description$
    1040 CREATE_keyword "NOTE 1", #3, Note$(1)
    1045 CREATE_keyword "NOTE 2", #3, Note$(2)
    1050 CREATE_keyword "NOTE 3", #3, Note$(3)
    1055 CREATE_keyword "CROSS-REFERENCE", #3, Link$
    1060 CREATE_keyword "CLOSE", #3
    1065 CLOSE #3
```

(continues on next page)

(continued from previous page)

```
**Note 1**
For notes, which are optional, please ensure that each note's section uses␣
↪the letter case
demonstrated here - **Note n** or **Notes** as appropriate.

**Note 2**
There may be notes sections if the keyword demands special attention. Notes␣
↪will be numbered
from 1 upwards for general purpose "applies to all" notes. Things to be aware␣
↪of, how to
crash the QL by misuse of the command etc.

**Note 3**
Where the normal QL differs from Minerva, or SMS etc, use a separate note for␣
↪each.

**CROSS-REFERENCE**
Again, please use Rich's letter case for this section, which again has a bold␣
↪heading. This
section should describe, very briefly, other similar commands located␣
↪elsewhere in this or
other files.

Please make sure that if this is a simple list of keyword links, see below,␣
↪that they do not
split across lines.

This is a link :ref:`lower-case-keyword`
```

Note: You should notice the first line in the above template. It is a link target. All keywords should have a link target set up just before the section header for that keyword.

Link targets are simply the keyword converted to lower case, with underscores replaces by a single hyphen, and spaces replaced with two hyphens. Percent and dollar characters are replaced by '-pct' and '-dlr' respectively.

The following examples should help make things clear:

- **DIM** - `.. _dim:` - no special needs here!
- **WHEN ERRor** - `.. _when--error:` - space replaced by '–', a double hyphen.
- **PRINT_USING** - `.. _print-using:` - underscore replaced by '-', a single hyphen.
- **DEV_NAME$** - `.. _dev-name-dlr:` - Underscore replaced. Dollar replaced by '-dlr'.
- **CHAN_B%** - `.. _chan-b-pct:` - underscore replaces. Percent replaced by '-pct'.

Warning: If you add a new keyword, and it has *exactly* the same name as another keyword in a different toolkit, then *do not* add a duplicate link target for the new keyword, leave the existing one alone and simply add your new keyword after the existing one with the same name, without adding a link target.

Duplicate target names are not allowed and changing the existing one, perhaps to give a numeric suffix, will invalidate all links pointing at it.

It is perhaps better to have all links point at the first one and the viewer can scroll down to see the others. The source should resemble the following:

```
.. _search:

SEARCH
======

This keyword was here first and has the original link target defined above.
↪ Yada yada yada.

SEARCH
======

Different toolkit, same keyword. There is no link target here. Yada yada␣
↪yada.
```

2.1.7 Commit Your Changes Locally

Git is a *distributed* version control system. You has a local copy of your GitHub repository and you commit locally, and nothing ever leaves your computer, you don't even have to have an internet connection up and running. Eventually, though, you have to `push` your changes back to your fork of my repository.

You should commit frequently and often is the rule I've heard, but I'm a firm believer in only committing - as far as possible - when something is finished. You should however, be aware that the bigger the changes you make to a file, or files, means that there is a higher chance of a conflict when you come to commit, if someone else has amended the same region of the same file(s) as you have. This is a version control problem in general and is not specific to git.

Conflicts are not something you will come across when dealing with your own forked repositories - unless you conflict with yourself by changing the same part(s) of the same file(s) in two or more separate branches.

What Has Changed

Before you `commit` your changes, it is good practice to be sure of what you have changed. The `git status` command will show you all the files that have changed since the most recent `commit`:

```
git status
```

Note: In the following, you might notice that file names etc are somewhat different from the real SuperBASIC-Manual repository file names. Don't worry about this, it's because there are no current

edits happening in that repository that can be used as an example.

Git responds with the following:

```
On branch working
Your branch is up-to-date with 'origin/working'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   source/Part2.rst
        modified:   source/p2_download.rst

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        source/images/CloneButton.png
        source/images/GetURL.png

no changes added to commit (use "git add" and/or "git commit -a")
```

What does it all mean? Well, starting at the top, we have this:

```
On branch working
Your branch is up-to-date with 'origin/working'.
```

Which informs you of the branch you are working on, and if you are ahead of the GitHub repository as far as new commits are concerned, here I'm up to date and on the *working* branch.

Next up, we see the list of files that have been modified since they were last committed:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   source/Part2.rst
        modified:   source/p2_download.rst
```

In this case, I've edited the two files listed above. Git gives a helpful hint as to what commands I should type in and execute in order to `stage` the files for a `commit` - `git add`, or, how to revert the changes I made - get rid of them altogether and take me back to the most recent clean version of the file(s) - `git checkout ...`.

Following on from the modified & tracked files, git shows a list of new (untracked) files within this local repository:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        source/images/CloneButton.png
        source/images/GetURL.png
```

Untracked files are files that git is not tracking changes to, and which have not been added to the `.gitignore` file. These are, basically, brand new files. Some or all of these files may be added to the repository at a later date, but for now, git is aware that they exist, but doesn't really care!

And finally, a warning that if I `commit` now, nothing will happen:

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Staging Changes

You can add files to be committed individually, or using wildcards:

```
git add source/images/CloneButton.png
git add source/images/getURL.png
```

Or:

```
git add source/images/*.png
```

If the `git status` command's output is acceptable, and you simply want to add all modified files, and all unstaged files, then you can indeed stage *everything* as follows:

```
git add --all
```

Git will not produce any messages unless something went wrong. If we check `git status` again, we see something different to the above:

```
On branch working
Your branch is up-to-date with 'origin/working'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   source/Part2.rst
        new file:   source/images/CloneButton.png
        new file:   source/images/GetURL.png
        modified:   source/p2_download.rst
```

Git is telling us here that all the listed files are about to be committed.

Note: If you amend a file that is already staged and thus appears in the list of files about to be committed, it will then also show up as a changed file, waiting to be staged. If you commit, the already staged (and unchanged) file will be committed leaving the amended file to be staged etc at a later date.

If you want the latest amendments included in your commit, you have to `git add` the file again. This will overwrite the previously staged file.

Commit

Committing your staged files means that they are added to your local repository, permanently. This doesn't mean you can't go back to a previous version of course, that is still possible.

When you commit, you must supply a message with brief (or otherwise) details of what you have done in this commit. If you don't supply one, your default editor will open and you should type your message there instead. It's easier just to supply brief details on the command line, with the `-m` option, as the following example shows:

```
git commit -m "Details on getting the source files and editing them added."
```

Git will respond with something similar to the following:

```
[working cc61ecf] Details on getting the source files and editing them added.
 4 files changed, 421 insertions(+), 143 deletions(-)
 create mode 100644 source/images/CloneButton.png
 create mode 100644 source/images/GetURL.png
```

2.1.8 Push Changes Back to GitHub

When you `commit` your changes, they are only stored locally, on your computer, you now need to `push` those changes, and perhaps others, back to your GitHub repository. You can do this at any time when connected to the internet.

The following example shows the use of the `git push` command:

```
git push
```

There's not much to it! Git will do some background processing and then something like the following output will appear on screen:

```
Counting objects: 8, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 24.88 KiB | 0 bytes/s, done.
Total 8 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/SinclairQL/UpdatingSuperBASICManual.git
   dca541e..cc61ecf  working -> working
```

That's it. All your locally committed changes have now been pushed back online to your GitHub repository. However, although they are in your repository, they are not in mine, and it's mine that the online documentation at ReadTheDocs.io is generated from, so, how do you go about getting your changes merged into my repository, and update the online manual? Read on.

2.1.9 Create a Pull Request

At this point I could write lots and lots of text about how you would go about creating a pull request. It's not difficult, and in fact, when you push a change to a forked repository and then visit (or refresh) the repository page in your browser, you will notice that GitHub has seen your commit, understood that you have a forked repository, and will put a big green button with "create a pull request" on it.

Click the button, add a descriptive heading, add a descriptive - ahem - description of what you did etc, and OK. Job done, I'll hear from GitHub that you have raised a `pull` request and I'll be able to inspect it, communicate with you, and - eventually - merge your changes into the main repository, which will cause a rebuild of the documentation over on http://superbasic-manual.readthedocs.io.

If you really need *more* details, then get yourself a copy of the (free) book "Pro Git" by Scott Chacon, from https://git-scm.com/book/en/v2, and check out chapter 6 which is all about GitHub. There is a whole section on creating and maintaining `pull` requests. It starts around page 203.

2.1.10 Delete Your Fork

Your work is complete. Well, it is when you hear back from me that your work has been `pull`ed into the main repository, so at this point you have the option of deleting your forked repository as it is no longer needed.

You don't have to delete it though, if you think that you might do some more work in the future, however, this means that from time to time, and *definitely* before you do any more work, you need to sync your fork of the main repository with any work being done and pulled into the main repository itself. This is explained below, in some detail.

2.1.11 Keeping Your Fork in Sync

If you think that you might do some more work on the manual at a later date, then why not keep a hold of your forked repository and update it as and when required to sync it with the main repository. It's not all that difficult at all.

Add an Upstream Remote

The first and most important thing to do is add the main repository as a `remote`. Don't worry about the term `remote` it's just a name as far as we need to be bothered!

Open up a command line session and execute the following commands:

```
cd SuperBASIC-Manual
git remote add upstream https://github.com/NormanDunbar/SuperBASIC-Manual.git
```

That's it! You can check that it worked as follows:

```
git remote -v
```

The output should resemble the following:

```
upstream https://github.com/NormanDunbar/SuperBASIC-Manual.git (fetch)
upstream https://github.com/NormanDunbar/SuperBASIC-Manual.git (push)
```

From now on, the name *upstream* will refer to my repository, the one you originally forked back at the start of this section. *Upstream/master* and *upstream/working* refer to the appropriate branches on my repository.

Sync Your Fork

Your fork may be behind the upstream repository, as the main repository is now referred to in your development system, so before making any changes, bring your fork up to date. You need to be sure that all your local changes are `push`ed to your fork first though.

The following commands will commit and push your latest work back to your fork. This will not be necessary if you have not done any work since the last push.

```
git status
git add ....
git commit -m "Pushing my latest work before syncing from upstream."
git push
```

Now that your fork is up to date will your local work, you should fetch the upstream/master branch. Doing this will also cause your development area to effectively do a `git checkout upstream/master` command, so you will need to `checkout` your own *master* branch afterwards:

```
git fetch upstream
git checkout master
git merge upstream/master
git push
```

That's the *master* branch taken care of. We don't however, do our work in that branch, we use the *working* branch, so we also need to be sure that we have synced our *working* branch with upstream too:

```
git fetch upstream/working
git checkout working
git merge upstream/working
git push
```

Now we are up to date *locally* and our fork is at the same commit point as the upstream repository. We can now go to work. You will (probably):

- Make changes and test in the local *working* branch;
- `Commit` those changes and `push` them to your fork's *working* branch;
- `Check out` your *master* branch
- `Merge` the changes from your *working* branch;
- `Push` those to your fork's *master* branch;
- Raise a `pull` request in the normal manner.

All of these steps are described in some detail, above.

CHAPTER

THREE

PART THREE - RESTRUCTUREDTEXT

Contents of Part 3:

3.1 ReStructured Text

ReStructuredText (which apparently always has to be written like 'reStructuredText`!) is a plain text markup language. It is supposed to be readable even without conversion to other formats, unlike DocBook XML, etc, but it's not as simple to read as something like Markdown, for example. However, it's quite a powerful system.

Check out this cheatsheet for better information on using RST[1] in Sphinx (which is what we are doing after all).

For a full description of the RST "language", try any of the following links:

- Docutils main page
- Quick introduction
- Deep stuff
- Sphinx primer

And, the ever faithful, Wikipedia.

RST, can be converted into numerous other output formats using Sphinx (as we are for the SuperBASIC Manual) or by using a utility named `pandoc` we can even create Word and/or Libre Office documents - with user defined styles etc - as desired.

3.1.1 Comments

You may find that it is useful to add a comment to the RST source code for the manual page you are editing/creating. This is done as follows:

```
.. This is a comment. It is on a single line.
```

Comments have no effect on the generated output in HTML etc. If the comment needs to be longer than a single line, add additional lines and/or paragraphs (see below) as necessary.

[1] RST is what I'm calling the ReStructuredText files from now on. It saves typing, and I'm basically lazy!

3.1.2 Section Headings

A section heading is simply the heading text, underlined by some combination of punctuation characters. Sphinx is quite good at determining the hierarchy, but it pays to be consistent. I use the following:

```
==================
Title of Document
==================

Heading 1
=========

Heading 2
---------

Heading 3
~~~~~~~~~

Heading 4
"""""""""""
```

The document title is overlined as well as underlined.

You can go down another couple of levels, but it gets a bit messy. HTML only allows 6 levels of heading, so that's probably about as much as anyone should need. Unless you work in a legal establishment, of course, they do like to complicate things!

It is advisable to add *reference link destinations* above each section header, if you intend to set up links, from other source files, to any of the sections in "this" source file. For example, if another source file wishes to link to Heading 2 above, this file should give heading 2 a link destination, as follows:

```
.. _Heading-two:

Heading 2
---------
```

See *Links to Other RST documents* below for full details.

3.1.3 Paragraphs

Start typing at the left margin. Don't stop typing at the end of a line - but if you do, don't worry, the line feed will be ignored, and replaced with a space as the various lines making up the paragraph are merged.

Leave a blank line between paragraphs, otherwise it all runs into one.

If you happen to indent a paragraph with a leading tab, or spaces, then the following will happen:

> This paragraph is full of useless text and is indented using a TAB. Each line of the paragraph - thanks to the Notepad++ editor - will also align with the indent.
>
> This paragraph, on the other hand, is indented using leading spaces, 4 to be precise, and subsequent lines also indent with 4 spaces - thanks again to Notepad++. You can see what the results are in both cases.

3.1.4 Character Formatting

Bold formatting is facilitated by a pair of asterisks wrapped around the text you want to be bold. No spaces are allowed between the leading asterisks and the text, or the end of the text and the trailing asterisks.

```
This will be **bold** text.
```

Which gives the following:

This will be **bold** text.

Italic text uses a single asterisk:

```
This will be *italic* text.
```

Which gives the following:

This will be **italic** text.

You want both? Sorry, they cannot be nested. As this shows:

```
**This is bold *this should be italic+bold, but isn't* - but this is still␣
↪bold!**
```

This is bold *this should be italic+bold, but isn't* - but this is still bold!

Subscript and superscript need a slightly different format. You put the text to be raised or lowered in between backtick characters, not single quotes:

```
e=mc :sup:`2`
```

e=mc ^{2}

This is fine if you want the space prior to the superscripted text, but if not, use the 'invisible space' character to prevent the space from appearing. That character is explained below. For example:

```
e=mc\ :sup:`2`
```

e=mc^{2}

Which, to me, is much better looking *and* it won't accidentally get used as a word break in a paragraph!

Subscript is the same:

```
Sulphuric Acid has the formula: H\ :sub:`2`\ SO\ :sub:`4`.
```

Sulphuric Acid has the formula: H_2SO_4.

3.1.5 Escaping Special Characters

The asterisk (*), the backtick (`), the invisible space, the underscore (_), the backslash (\) (and more?) are special and have to be escaped using a back slash character (\):

```
- Asterisk = \*
- Underscore = \_
- Backslash = \\
- Backtick = \`
- Invisible\ space (did you see it?)
```

Which gives the following:

- Asterisk = *
- Underscore = _
- Backslash = \
- Backtick = `
- Invisiblespace (did you see it?)

The invisible space is used in places where the end of some special formatting needs to be shown to be separate from the following text. Sometimes, for example, when using sub_{script} or $super^{script}$, or occasionally when you have a full stop following the end of a URL, for example. The invisible space tells the parser to stop parsing and that the text following the "space" is to be considered as a separate parsing entity.

The text above, where we have subscript and superscript, was created as follows:

```
Sometimes, for example, when using sub\ :sub:`script` or super\ :sup:`script`,
↪ or occasionally ...
```

You can see the use of the invisible space to separate the ':sub:' and ':sup:' from the first part of the word.

3.1.6 Syntax/Location Table

If a command only has a single form, then a table like this will suffice:

```
+----------+-------------------------------------+
| Syntax   | Command(a,b)                        |
+----------+-------------------------------------+
| Location | QL ROM, TOOKIT II, SOMEWHERE ELSE   |
+----------+-------------------------------------+
```

The above will render into the following:

Syntax	Command(a,b)
Location	QL ROM, TOOKIT II, SOMEWHERE ELSE

However, if there are alternate forms for the command, the table should look like this:

```
+----------+-----------------------------------+
| Syntax   | Command(a,b) or                   |
|          +-----------------------------------+
|          | Command(a,b,c)                    |
+----------+-----------------------------------+
| Location | QL ROM, TOOKIT II, SOMEWHERE ELSE |
+----------+-----------------------------------+
```

which becomes the following after a build:

Syntax	Command(a,b) or Command(a,b,c)
Location	QL ROM, TOOKIT II, SOMEWHERE ELSE

Which gives a better layout in the generated HTML.

You might think that the following will work:

```
+----------+-----------------------------------+
| Syntax   | Command(a,b) or                   |
|          | Command(a,b,c)                    |
+----------+-----------------------------------+
| Location | QL ROM, TOOKIT II, SOMEWHERE ELSE |
+----------+-----------------------------------+
```

Unfortunately, it doesn't. The data in the cell(s) are reformatted and multiple spaces and/or line feeds will be removed and replaced by a single space - just like HTML does. You can see how it *doesn't* look correct here:

Syntax	Command(a,b) or Command(a,b,c)
Location	QL ROM, TOOKIT II, SOMEWHERE ELSE

3.1.7 Tables

We use grid tables, where we design the table according to how we want it to look:

```
+--------+--------+--------+
| Cell 1 | Cell 2 | Cell 3 |
+--------+--------+--------+
| Cell 4 | Cell 5 | Cell 6 |
+--------+--------+--------+
```

Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

We can add headings too:

```
+--------+--------+--------+
| Head 1 | Head 2 | Head 3 |
+========+========+========+
| Cell 1 | Cell 2 | Cell 3 |
+--------+--------+--------+
| Cell 4 | Cell 5 | Cell 6 |
+--------+--------+--------+
```

Head 1	Head 2	Head 3
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

Cells can span along the row:

```
+--------+--------+--------+
| Head 1 | Head 2 | Head 3 |
+========+========+========+
| Cell 1 | Cell 2 and 3    |
+--------+--------+--------+
| Cell 4 | Cell 5 | Cell 6 |
+--------+--------+--------+
```

Head 1	Head 2	Head 3
Cell 1	Cell 2 and 3	
Cell 4	Cell 5	Cell 6

And also, down the columns:

```
+--------+--------+--------+
| Head 1 | Head 2 | Head 3 |
+========+========+========+
| Cell 1 | Cell 2 | Cell 3 |
|        +--------+--------+
| Cell 4 | Cell 5 | Cell 6 |
+--------+--------+--------+
```

Head 1	Head 2	Head 3
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

Getting text on separate lines within cells is interesting, and a bit of a hack. You've already seen it above for the Syntax/Location tables, but here we are again:

```
+--------+----------------+
| Head 1 | Head 2         |
```

(continues on next page)

(continued from previous page)

```
+========+=================+
| Cell 1 | Cell 2 - line 1 |
|        | Cell 2 - line 2 |
+--------+-----------------+
| Cell 4 | Cell 5          |
+--------+-----------------+
```

Head 1	Head 2
Cell 1	Cell 2 - line 1 Cell 2 - line 2
Cell 4	Cell 5

Looks good eh? No? Oh well, this works, but only for HTML output:

```
+--------+------------------+
| Head 1 || Head 2          |
+========+==================+
| Cell 1 || Cell 2 - line 1 |
|        || Cell 2 - line 2 |
+--------+------------------+
| Cell 4 || Cell 5          |
+--------+------------------+
```

Head 1	Head 2
Cell 1	Cell 2 - line 1 Cell 2 - line 2
Cell 4	Cell 5

We have to "double pipe" all entries down the column(s) we want to have text on separate lines, unfortunately, plus the headings or we get strange indents.

The best looking result, which works in HTML, PDF and (as far as I can tell) all other formats, is to merge cells:

```
+--------+-----------------+
| Head 1 | Head 2          |
+========+=================+
| Cell 1 | Cell 2 - line 1 |
|        +-----------------+
|        | Cell 2 - line 2 |
```

(continues on next page)

(continued from previous page)

```
+--------+----------------+
| Cell 4 | Cell 5         |
+--------+----------------+
```

Head 1	Head 2
Cell 1	Cell 2 - line 1 Cell 2 - line 2
Cell 4	Cell 5

3.1.8 Lists

Bullet Point Lists

These start with a hyphen and a space. Then the text of the list item follows on the same line. Do not press enter until you wish to start a new paragraph, and if intended to be part of the list item, it should be indented to line up under the first character after the hyphen space.

```
- This is a single line item in a list.
- This is another.

  However, this is not a third, but a second paragraph of the second list␣
↪item.

- And here we go back again. Item 3.
```

- This is a single line item in a list.
- This is another.

  However, this is not a third, but a second paragraph of the second list item.
- And here we go back again. Item 3.

You can nest lists:

```
- A top level item.

  - a nested item.
  - And another.

- Another top  level item.
```

- A top level item.
  - a nested item.
  - And another.
- Another top level item.

And so on.

Enumerated Lists

These are similar to the above, but start with a digit, or a hash dot (#.):

```
#. Item 1.
#. Item 2.
#. Item 3.

  #. Nested Item 1
  #. Nested Item 2

4. Item 4.
```

1. Item 1.
2. Item 2.
3. Item 3.
    1. Nested Item 1
    2. Nested Item 2
4. Item 4.

In the above, we had to start item 4 with the digit 4. There might be a way to get this to work automagically, I'm still looking.

We can use Letters too:

```
a. Item a.
#. Item b.
#. Item c.
```

a. Item a.
b. Item b.
c. Item c.

Or Roman Numbers, but how? This should, but doesn't work:

```
I. Item I
#. Item II.
#. Item III.
#. Item IV.
#. Item V.
```

I. Item I
II. Item II.
III. Item III.
IV. Item IV.
V. Item V.

As you can see the above doesn't work! It should allow i,ii,iii etc and I,II,III and so on, but this one doesn't seems to work.

Definition Lists

This is how we do definition lists:

```
A Term
    A definition goes here.

B Term
    B definition goes here. You can have many paragraphs too.

    Like this one you are currently reading. Just make sure that each␣
↪paragraph is separated by a blank line, and indented to the same level.
```

A Term
: A definition goes here.

B Term
: B definition goes here. You can have many paragraphs too.

  Like this one you are currently reading. Just make sure that each paragraph is separated by a blank line, and indented to the same level.

3.1.9 Program Listings

In-line

This is an example of an inline section of code, `1000 A$ = 'This is Fun!'`. We use the following to make it so:

```
This is an example of an inline section of code, ``1000 A$ = 'This is Fun!'``.
```

Single line

A single line of code is defined thus:

```
::

    100 PRINT "Hello World!"
```

Which renders to the following:

```
100 PRINT "Hello World!"
```

The double colon can go at the end of the preceding line, followed by a blank, followed by the indented code, or, can be on a line of its own, followed by a blank then the indented code.

If the double colon is at the end of a line of text, then a single colon will be created in the output. If the double colon is on a line of its own, then no colons will be generated.

Multiple lines

Multiple lines of code are defined thus:

```
::

    100 REPeat Silly
    110   PRINT "Hello World!"
    120 END REPeat Silly
```

Which renders to the following:

```
100 REPeat Silly
110   PRINT "Hello World!"
120 END REPeat Silly
```

The double colon can go at the end of the preceding line, followed by a blank, followed by the indented code, or, can be on a line of its own, followed by a blank then the indented code.

3.1.10 Code Highlighting

Give the above for single and multiple line code listings, you can add syntax highlighting, if desired. Sphinx uses a Python library called pygments to do the syntax highlighting. To use this in your docs, simply set up your code blocks as follows:

```
.. code-block:: python

    total = 0

    # Remember, range(a, b) is from a to (b-1) inclusive!
    for x in range(1, 667):
        total += x

    print("The sum of all the numbers from 1 to 666 is %d" % (total))
```

Unfortunately, MC68000 assembly language and SuperBASIC do not have highlighters defined. yet!

The above source code renders to the following:

```
total = 0

# Remember, range(a, b) is from a to (b-1) inclusive!
for x in range(1, 667):
    total += x

print("The sum of all the numbers from 1 to 666 is %d" % (total))
```

3.1.11 Links

Normal http:// type links do not need any special formatting - they are identified as links to a URL, and will be rendered as such. Links within the document, either in the current source file, or to other source files, do require a bit of formatting.

The general format for links is:

```
`link text here <URL Here>`__
```

The various different link types are detailed below.

External links

For example:

```
This is a link to my `GitHub repository <https://github.com/NormanDunbar/
↪SuperBASIC-Manual.git>`__. Try it.
```

This is a link to my GitHub repository. Try it.

Chapter & Section Headings

Each chapter and section heading becomes it's own link but *only within the same source file*. You simply enclose the chapter or section header in backticks and a trailing underscore to create a link:

```
This one links to `Code Highlighting`_ above.
```

This one links to *Code Highlighting* above.

Embedded underscores etc need to be escaped.

Sadly, cross source file links cannot be created this way, as this example demonstrates. It tries to link to a section heading in part two:

```
`Forking the Repository`_ is a good place to start...
```

`Forking the Repository`_ is a good place to start...

All you get is an errors when building the document:

```
ERROR: Unknown target name: "forking the repository".
```

If you wish to put your own text rather than the section header, you should replace spaces, underscores etc with a hyphen in the link's URL to get to that section and lower case the remaining letters. For example:

```
This is a link to the section above on `Escaping stuff <#escaping-special-
↪characters>`__.
```

This is a link to the section above on *Escaping stuff* .

However, if the section in in another document, *and* only HTML is to ever be generated, then the following will work, but hard codes the output file:

```
This is a link to the `Python <software.html#python>`__ section in the␣
↪Software chapter.
```

This is a link to the Python section in the Software chapter.

Sadly, this latter link will only work for HTML formatted output. If you generate other formats - PDF for example - then there will be warnings that the link is not valid and indeed, the link will not work.

For best results, regardless of the output format desired, see the next item.

Links to Other RST documents

If you need to link to a section heading in this, or *another* RST document, then you must use the ':ref:' directive as shown in the following:

```
This is a link to :ref:`P2_FreeGitBook` which should be found in part 2 of␣
↪this manual.
```

Additionally, and in order to make the link work, you need to create the link named 'P2_FreeGitBook' at the appropriate place in the other document. For best results links to other documents should point at a section or sub-section heading, so put the link just above that location:

```
.. _P2_FreeGitBook:

Free Git Book
-------------

You might like to download and read the *Pro Git* book, by Scott Chacon & Ben␣
↪Straub. This is a book published by Apress but which is given away for free␣
↪on the web at https://git-scm.com/book/en/v2 - various formats are␣
↪available.

...
```

This is a link to *Free Git Book* which should be found in part 2 of this manual.

You will note that the text inserted as the link text, is the section header of the section immediately following the link definition (in the other document), in this case, 'Free Git Book'.

3.1.12 Footnotes

And finally, for now, footnotes. You add a footnote indicator to your text like this:

```
This text has a footnote [#foot_1]_ embedded in it.
```

This text has a footnote[2] embedded in it.

Note: You can, if you don't like the space before the footnote indicator in the generated text, use an invisible space as described above. I prefer to use one myself, but it's not essential. For example:

[2] This is the example footnote.

```
This text has a footnote\ [#foot_1]_ embedded in it.
```

At the *bottom* of the file you are adding the footnote to, you add the following:

```
.. :rubrick: Footnotes
```

And then, beneath that you add the footnote text for *all* the footnotes in this current file:

```
.. [#foot_1] This is the example footnote.
.. [#foot_2] This is another footnote.
```

There are other ways of adding footnotes, but I find this to be the best as you explicitly name each footnote. RST allows auto numbering of footnotes, but then you must remember to add in new footnotes in the same order after the `.. rubrick: Footnotes`, as they appear in the text - or it all gets messy!

CHAPTER

FOUR

PART FOUR - THE MIGRATION

Contents of Part 4:

4.1 Introduction

This part describes some of the trials and tribulations I went through to create the current (as of today, anyway!) online version of Rich Mellor's **Online SuperBASIC Manual** as seen originally at http://www.rwapsoftware.co.uk/SBASIC_reference_manual_online/Foreword.html and now, online at http://superbasic-manual.readthedocs.io/en/latest/ where the latest (English language) version can be found. Anyone feel like translating?[1]

4.1.1 Why Bother?

Many years ago, Rich wrote a proper book, using Text 87, I believe, and this was published by Roy Wood's company *Q Branch*. Although I never had a copy, I am informed that it was a very well received book, and regular updates were provided. The book itself was over 1,000 loose leaf pages - which helped in the updating.

Time goes by (that's its job after all!).

The internet arrives.

The QL is somewhat left behind, given a pretty basic lack of TCP/IP networking abilities.

A group of *old farts*[2] still hang on to the QL, after all, it is one of the better computer systems around, given its age, and the fact that it did have one of the better BASIC languages around, at the time. (1984!)

Even better, it had a *genuine* "32" bit processor, which was sort of correct, but it had an 8 bit data bus, so each of those 32 bits needed 4 fetches! Still, advertising.

[1] Doing this in English was hard enough, I don't suspect there will ever be a translation!

[2] *Old farts* is a genuine term of endearment, and not what you might be thinking!

4.1.2 The First HTML Version

Some work was done by authors unknown, to convert the original Text 87 documents into a first draft HTML document. This involved much messing around with printer drivers and then reading through the generated (and binary) file to replace certain characters with suitable HTML alternatives.

Note: Getting text out of a Text87 file is a nightmare as the internal file format is not documented, and the author doesn't appear to answer emails on the subject. A printer driver type thing is about all there is. Especially if you wish or need to retain some of the formatting.

The first version was a team effort - if you were involved, let me know and I'll update this book to include your name - and a set of very rough HTML files was the result. There were bugs, but the browsers of the time coped beautifully - by simply ignoring them! The section on the ED keyword, for example, had a table of key presses which never actually showed any of the arrow keys, for example.

There were the odd occasional spelling mistake - as would be expected in a work of this size. There probably still are!

There were tables that were no longer tables - HTML ignores multiple spaces and tabs, replacing them with a single space, in most cases. Not helpful to a nice tabular layout.

And there were program listings which were not program listings, they were paragraphs of italic text. All the lines ran together and any of the special HTML characters - '<', '>' or '&', for example, went awol as they were illegal characters in a non `<pre> .. </pre>` section. Sigh.

However, it was a good start.

4.1.3 My Own Involvement

Rich put a request out on the ql-users mailing list for help, and also on the Sinclair QL Forum. I believe the silence was deafening! And, being an author myself, pretty much as expected. So much for a community eh?[3]

I used to avoid having a lunch break in whatever location I was working at the time. This meant that I had to sit around for an hour or so, twiddling my thumbs or reading the Interwebs. Until one day I decided to have a look at tidying up the online manual - as requested by Rich in his call for help. I was bored!

I managed to *manually* convert all the forewords and appendices to something resembling half-decent HTML, but it was still pretty dire to be honest. Each file is many hundreds of lines long, and full of *stuff* generated by the various conversion programs. It wasn't easy to work with and each chapter took ages to convert from really crappy HTML into mildly crappy HTML.

By the time I left that contract, I had only got as far as converting the first three chapters of the keywords - A, B and C. That was it. At least the program listings looks like listings now, but it was an absolute nightmare converting each line from something like[4]:

```
... <span style="font-style: italic;">100 do_stuff(a, b, c)...</span> ...
↪<span style="font-style: italic;"><br>110 do_more_stuff: and_more_stuff(d,e,
↪f) ...</span>
```

[3] To be fair, this is a common occurrence in many communities. Everyone wants stuff done, but only a few dedicated people do anything to do the work. Others, sadly, don't, can't or won't!

[4] Yes, most of it was all on one, long, line!

into:

```
<pre>
100 do_stuff(a, b, c)...
110 do_more_stuff: and_more_stuff(d,e,f) ...
</pre>
```

Which is fine for a couple of lines, but a nightmare for some of the larger examples, even with a text editor that lets me record and save the odd occasional macro to do the stripping out of the various s etc.

Then, I left that contract and spent almost 3 years working in a location that was completely tied down - no personal internet usage, no ability to install even useful tools that could have helped me do my job etc. So, no work done at all on the manual I'm afraid - my wife wasn't too keen on me *wasting* my spare time doing *computer stuff*, so doing anything at home was a non-starter too. Apparently, I needed to *get a life* - whatever one of those is.[5]

4.1.4 This Version

This version came about as a result of something else I was having to do. Producing a web site from text files created in ReStructuredText. I was rather impressed and ran a little trial before creating a project on *GitHub* and on *ReadTheDocs* to cater for the need to have the source files in a version control system - so that when I screwed up an edit, I could always revert the changes! And ReadTheDocs gave me the ability to host the manual for free, well, for the inclusion of a couple of small adverts, occasionally.

Every time I commit a change and push it to the working branch, the working version of the manual gets rebuild in HTML, PDF and EPUB formats. This is the rough and ready versions and is not really suitable for general use.

The master branch is only ever updated, other than when I make a mistake and forget to checkout the working branch of course, when I finish a complete file, such as `KeywordsX.html` which will become `KeywordsX.clean.rst` and then built. The docs on the master branch are the good copies, ready for use.

Never, ever update the docs for the master branch, unless you have already proven them in working first.

[5] I think it means shopping for shoes and clothes, for her that is!

CHAPTER

FIVE

SEARCH