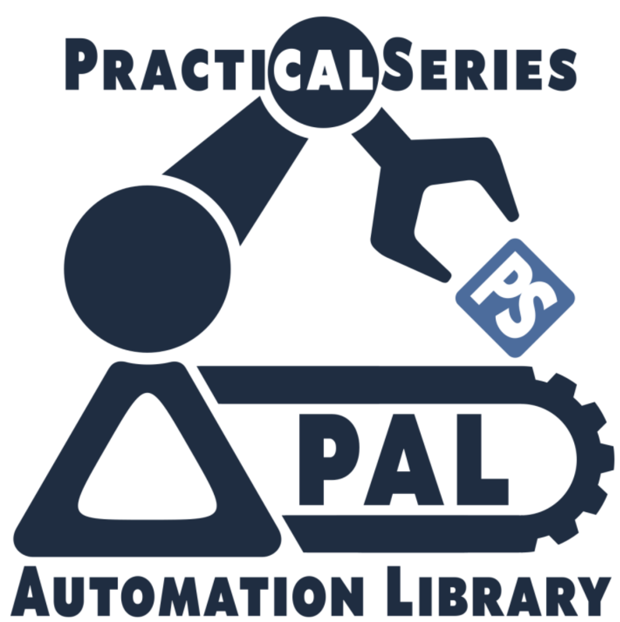
Practical Series

PRACTICAL SERIES AUTOMATION LIBRARY

SOFTWARE CONTROL MECHANISM

AUTHOR: MICHAEL GLEDHILL



Published By: Practical Series of Publications
Published in the United Kingdom
mg@practicalseries.com

Copyright 2021 Michael Gledhill

Document No.: PS2001-5-2302-011

Document Template: PS2001-5-nnnnn-nnn R02.00 SHORT GxP Blank (Ind-Calisto)

LICENCE This document and associated software are made available under the MIT License:

The MIT License (MIT)
Copyright © 2021 Michael Gledhill

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Based on template: PS2001-5-nnnnn-nnn R02.00 SHORT GxP Blank (Ind-Calisto) — Indexable PDF format WebIndex:ber

DOCUMENT AUTHORISATION

	NAME	POSITION	SIGNATURE	DATE
Author	Michael Gledhill	Lead Engineer		24 May 2022

The signature of the author confirms that the document has been prepared in accordance with an approved document management process, that all content is technically complete and that all relevant material has been included.

Reviewed by	Frank Greenwood	Project Manager		24 May 2022

The signature of the reviewer indicates that the document has been checked for technical content and that it complies with the technical standards, specifications and conventions.

Approved by	Christopher Wish	Quality Manager		24 May 2022

The signature of the Approver indicates that the document has been checked for compliance with the quality management Procedures.

REVISION

REVISION	DATE	REVISED BY	DESCRIPTION
R02.00	24 May 2022	Michael Gledhill	Properties standardised across all documents Changes to interrupt and functional group names
R01.00	21 Mar 2021	Michael Gledhill	First release for use

CONTENTS

1. **Introduction** **9**

1.1 Software Control Mechanism requirements **10**

1.1.1 Module revision numbering mechanism 10

1.1.2 A version control system 11

1.2 Scope and purpose of this document **13**

1.3 Ownership, status & relationship to other documents **14**

1.3.1 Ownership of the document 14

1.3.2 The status of this document 14

1.3.3 Relationship to other documents 14

1.3.4 Users of this document 14

2. **Approach to version control** **17**

2.1 Version control requirements of the SCM **19**

3. **The software revision numbering mechanism** **21**

3.1 Workflow arrangements **22**

3.2 Master branch revision states **23**

3.3 Development branch names **24**

3.4 Development branch commit tags **26**

3.5 Merging a development branch **27**

3.6 Individual module revision numbers **29**

3.6.1 Recording revision numbers within a programmable block 30

3.6.2 Recording revision numbers within a data block 34

3.6.3 Recording revision numbers within a User Data Type (UDT) 37

3.6.4 Software Module Register (SMR) 38

3.7 OB1 module revision numbers **38**

3.8 Commit points and filenames **41**

3.8.1 OB 1 and filenames 42

3.9 **Parallel development branches** 43

3.10 **OB 1 and the Merging of branches** 45
3.10.1 Merging a single branch or the first branch to merge 46
3.10.2 Merging additional parallel branches 50

3.11 **Nested branches** 56

3.12 **A note on commit messages** 57

4. **The website revision numbering mechanism** 63

4.1 **Workflow arrangements** 63

4.2 **Master branch revision states** 64

4.3 **Development branch names** 65

4.4 **Development branch commit tags** 67

4.5 **Merging of development branches** 67

4.6 **Individual page and file revision numbers** 68
4.6.1 Recording revision numbers within web page files 71

5. **Software storage and folder structures** 75

5.1 **An overview of the Project structure** 76

5.2 **Engineering stations** 78
5.2.1 ES software folders 80
5.2.2 Software development area (`1000 Software Projects`) 82
5.2.3 The Workspace and local repository (`2500 Git Projects`) 85
5.2.4 Understanding the Simatic Workspace 87
5.2.5 Understanding the Workspace as a local repository 95
5.2.6 Commit point archives 96
5.2.7 Maser ES — local repository backup to NAS 96
5.2.8 Remote repository 100

5.3 **Web development platforms** 102
5.3.1 WDP software folders 104
5.3.2 Understanding the website structure 105
5.3.3 Local repository 112
5.3.4 Master WDP — local repository backup to NAS 112
5.3.5 Remote repository 114
5.3.6 The live website 115

5.4 NAS based Project documentation 117
5.4.1 Understanding the Project folder structure 119
5.4.2 Project registry 123
5.4.3 Document versions 124

6. References and glossary 127

6.1 Document references 127

6.2 Glossary of terms 128

BLANK PAGE

1 Introduction

This document is the *Software Control Mechanism* (SCM) and is applicable to all Simatic Controller software developed for the *Practical Series Automation Library* of software modules (the PAL).

The Practical Series Automation Library (PAL) is a library of software modules and templates that have been made available for the Siemens Simatic S7-1500 range of controllers (and to a lesser extent the S7-1200 range).

The PAL software is configured and deployed using the Siemens Simatic TIA Portal programming environment.

The library is freely available under the MIT Open-source licence (see page 2 of this document).

This document, the Software Control Mechanism, has been produced by Michael Gledhill, under his authority as the lead engineer of the Practical Series Automation Library of software modules project.

1.1 Software Control Mechanism requirements

There are two principal requirements for the PAL Software Control Mechanism:

① Establish a mechanism for numbering and storing the various software module versions throughout the development, test and qualification phases of the Project

② Establish a mechanism for the storage and tracking of software module revisions within a formal Version Control System (VCS)

Expanding on these subjects:

1.1.1 Module revision numbering mechanism

The Validation Plan (VP) *[Ref. 001]*, established that software version control was a necessary requirement for the project and that all software modules within the Project must have individual revision and status information that covers all phases of the software development:

- Software development (system build)
- Testing (at both a modular and integrated level)
- Qualification
- Release for use

The revision system must also be applicable to the TIA Projects as a whole (rather than just the individual modules within the projects); to clarify, the software modules do not exist within their own right, each software module is stored in TIA Portal project that expands as each new software module is developed.

These TIA Portal projects are backed up and multiple revisions may be in used at the same time, all of these TIA Projects must also be part of the Software Control Mechanism).

1.1.2 A version control system

A version control system (VCS) is a mechanism for recording changes made to any files within a software project. It records all the changes, what files were affected by each change and a reason explaining why those changes were made. It also records who made the change and the time and date of the change.

The VCS keeps a record of every change made within the project and allows any file that has been modified to be reverted back to a previous state. It means that if a software module is changed, the original module can always be recovered by the VCS.

Version control systems generally have other facilities too, they are able to show the differences between two different versions of the software (even down to lines within a file), they allow multiple people to work on the project at the same time—even to work on the same file at the same time, and they provide mechanisms for resolving conflicts (where two different people have modified the same section of a file).

Version control systems can be applied to any kind of project; it can be a website, a documentation project, a software application, engineering control system—anything at all, as long as it's a collection of files that can be stored on computer.

The version control system does not itself edit or modify any of the files within the project; it simply records the changes and, where it recognises a file type, is able to display those changes that have occurred to it.

The version control system does not care what software application is used to modify files within the project, it can be anything: text editor, word processor, file manager, graphics editor, specialist programming application &c. It cares only, that a file under its control has been modified and why the modification was made.

Version control systems simply record any change made within a collection of files (the project), who made it, when it was made and the reason why. That is all.

A VCS could be applied to TIA Portal projects, these are stored as archived files (essentially `zip` files); however, these types of files are proprietary and are not directly accessible to the VCS. The VCS could, under these conditions, store each archived file, it would not, however, be able to access the internal components of the file to determine what changes have been made to any particular part of it (i.e. it could not identify a particular change to a particular module).

With the advent of TIA Portal V16, Siemens introduced the concept of *Workspaces*, these are environments (essentially, just Windows folders) into which the programmable aspects of a TIA Project (blocks, data types and tags) can be exported (or imported) as XML[1]files.

This is a new concept, previous versions of TIA Portal did not offer the facility of exporting software modules in a widely accessible (text based) format, the software could only be read by the proprietary TIA Portal package itself.

The benefit of this new Workspace facility is that the exported files are stored as XML files, and XML files are an ideal format for version control systems (VCSs), version control systems can read every aspect of an XML file and identify any changes that have been made, and, just as importantly, keep track of all these changes. Additionally, each block, data type and tag table is exported as its own XML file and as such allows the tracking of each individual element within the software library. It would for example, be possible to identify all the changes made to a particular Function (e.g. FC01001) and determine at which point in the revision history each change was made.

This was the purpose of Siemens adding this Workspace facility to TIA Portal, it allows proper version control of the software being developed in a TIA Portal project. It also does not require a proprietary Siemens VCS, any and all VCS systems can track text-based files (it is fundamentally, what they were designed to do).

To make things easier, Siemens also allow third-party *"add-ins"* to be created that can interface with these new Workspaces. One such add-in *(created by Siemens)* provides an interface to the version control system Git and its online partner GitHub.

The Git add-in allows TIA Portal to interface with a Git controlled Workspace, Git also supports various graphical user interfaces, in particular, Git can be controlled and managed from within the Visual Studio Code (VSC) text editor, VSC is widely used within the PSP and will be the preferred solution for providing a VCS interface for the PAL software.

---

[1] XML or eXtensible Mark-up Language files are text files that are both machine and human readable; very similar to HTML (HyperText mark-up Language) and widely used to store documents in a manageable and readable format; it contains both content and structure.

1.2 Scope and purpose of this document

This document is applicable to all software modules developed as part of the PAL, it explains the mechanisms used to document and control the different versions of each software module as it progresses through the various project phases. It also details the Git version control system and its application within the Project to track all software changes and provide regression mechanisms to access earlier software versions.

Broadly, this document covers the following:

① Software revision numbering system

- Software development (system build)
- Testing
- Deployment (commissioning) & Qualification
- Release

② Git Version Control System

- Purpose of
- A tracking and development philosophy

③ GitHub Online repository

- Purpose
- Accessing and control

1.3 Ownership, status & relationship to other documents

This document is an ancillary document for the Project, the ownership of the document (those whom control it and are able to modify it), its status within the Project and its relationship to all other primary documents are important factors and are explained below:

1.3.1 Ownership of the document

This document has been produced, and is controlled and maintained by the Practical Series of Publications (PSP).

This document and all the documents that it references are subject to the change control management procedures for this project.

1.3.2 The status of this document

This document is an internal PSP document and is **not** a deliverable item under the terms of the project.

1.3.3 Relationship to other documents

This document expands on the software revision tracking and control and revision numbering mechanisms discussed in the Functional Specification (FS) *[Ref. 002, § 4.5]* and the Software Design Specification (SDS) *[Ref. 003, § 5.2.4]*.

Its place in the document structure for the Project is shown in Figure 1.1.

1.3.4 Users of this document

This document is technical in nature and users of it should be familiar with the TIA Portal, Git and GitHub version control systems and the terminology common to those applications.

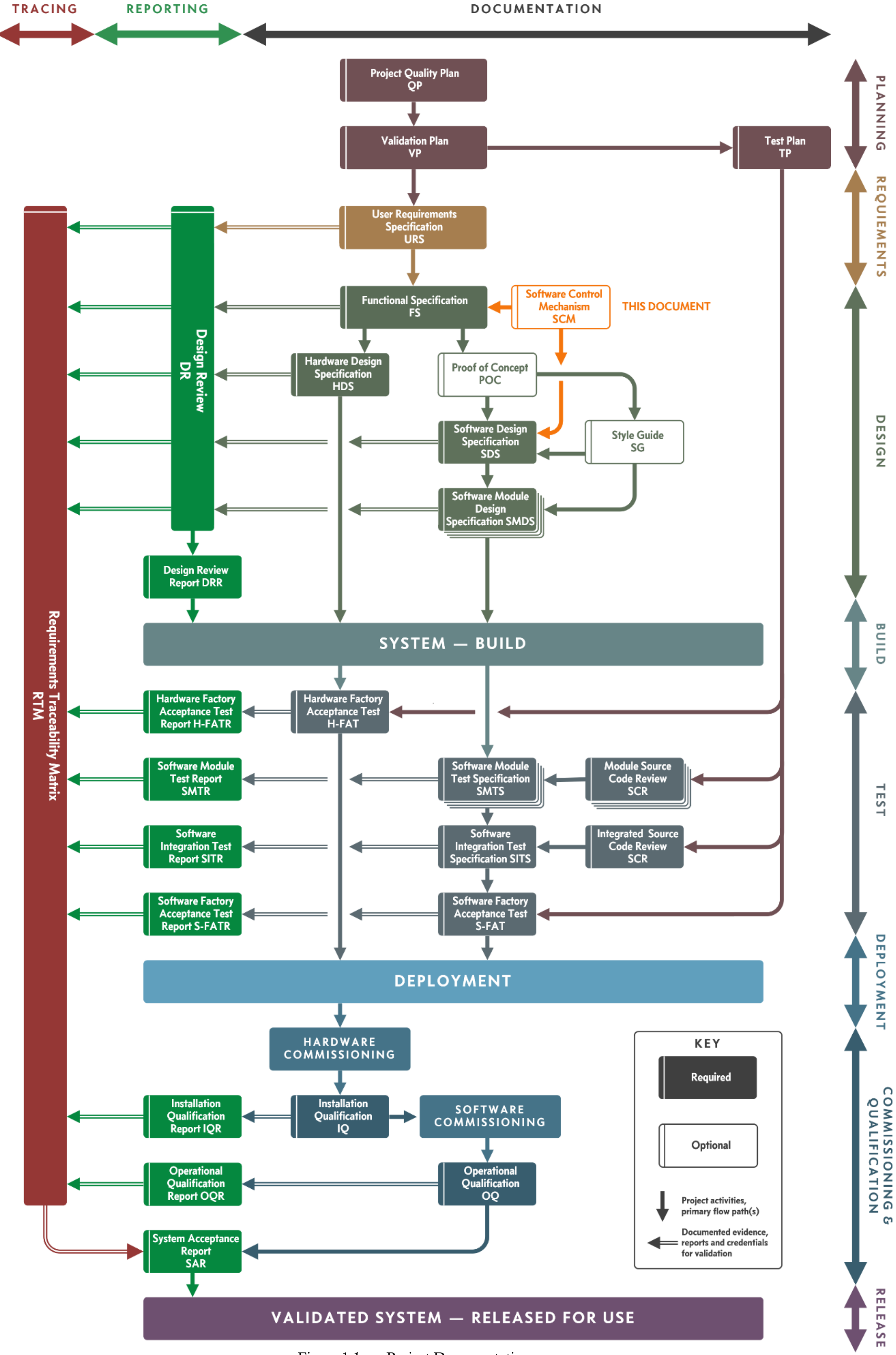


Figure 1.1 Project Documentation

BLANK PAGE

2 Approach to version control

The SCM detailed here must primarily work within the confines of the Git (and GitHub) version control systems. It must, in addition, provide a navigable set of revision numbers, both for each module within the PAL and for each revision of the TIA Portal project that contains those modules.

The revision numbering mechanism must be clear, readable (by humans) and explain the current status of the software (i.e. under development, under test, in qualification or released for use).

Git and GitHub use commit numbers derived from the checksum of files being added to the repository. These appear at best to be seemingly random seven-digit[1] hexadecimal numbers. They do not represent a meaningful number that is useful for team members trying to identify a revision path.

---

1 These commit numbers are of course not random numbers. They are a checksum carried out of all the files in a commit, plus a header that contains other information (the commit numbers that immediately preceded this commit, plus some information about directory structures &c.).

A checksum is basically a function applied to the binary value of every byte in a file that gives a reproducible figure that can be used to check to see if two files are the same or to identify data corruption within a file.

The commit number used by Git is a checksum encoded by using the SHA-1 algorithm (Secure Hash Algorithm 1). This produces a 20-byte (40 digit) hexadecimal number that uniquely identifies a commit. The commit number shown is just the first seven digits of the full commit number. This is usually enough to uniquely identify a commit (even on very large projects).

The first seven digits of a commit number gives 268 million unique values, the full 20-byte number has 1.5×10^{48} unique values (a similar number to the quantity of atoms that make up the Earth); these values also only apply within a repository (two different repositories can have the same commit number, they don't interact with each other).

The chance of a duplicate 20 byte commit number is vanishingly small, and is generally not a consideration, even on every large projects.

Git and GitHub do however, allow any commit point to have an associated tag, this is entirely at the discretion of the user and (*other than the requirement of being unique*) can be anything at all.

This allows each commit point to be tagged with a more meaningful (*semantic*) version number. Something that makes sense to humans.

This sematic version numbering scheme (used to tag each commit point) will provide a unique number that identifies the current revision of the software module and also provide status information about which of the phases of software development the software is currently in:

- Software development (system build)
- Under test
- Commissioning and qualification
- Released for use

The software version numbering scheme will be incremental in nature (the revision numbers will only go up), this provides a traceable approach to the software, it will always be possible to distinguish between earlier and later versions of the software, simply by examining the version numbers.

To complicate matters, the individual software modules do not exist within their own right, each software module is stored in TIA Portal project that expands as each new software module is developed. These TIA Portal projects exist on multiple development branches (see section 3) within the VCS (Git and GitHub) repositories, these TIA Projects must also form part of the Software Control Mechanism(SCM).

The testing of individual software modules (software module testing) and the testing of multiple modules (software integration testing) is carried out at specific intervals throughout the course of the Project, each such test must have its own TIA Portal "test" project as a record of the test (allowing the test to be repeated if required). Again, the SCM must provide a mechanism for recording and storing each test revision of the software.

2.1 Version control requirements of the SCM

There are seven components that are necessary and required by the SCM in terms of version control and management:

① Version tracking of individual modules within a TIA Project or files within a website

② Version tracking of the TIA Projects containing the individual modules

③ Filename allocation to the various TIA Projects

④ Workflow arrangements for the VCS, including branching and merging procedures

⑤ Local storage locations of TIA Projects and VCS repositories

⑥ Remote storage of the VCS repositories (GitHub)

⑦ Internal (PSP) backup mechanism for TIA Projects

Each of these components is addressed in the remainder of this document.

BLANK PAGE

3 The software revision numbering mechanism

This section describes a revision numbering strategy for the PAL software under the control of the Git and GitHub version control systems.

As stated previously, Git and GitHub use commit numbers to identify individual submissions to the repository, these are commonly referred to as *hash* or *sha* (pronounced *shar* to rhyme with *bar*) numbers. These are unique seven-digit hexadecimal numbers, and while they identify exactly, a particular revision within the repository, they do not do so in a way that can be easily interpreted by humans trying to understand the work-flow of the project (given two commit numbers: `[af25d47]` and `[9cf63b1]`, it would not be possible to say, just by looking at them, which came first), commit numbers can be considered completely random, but non-repeating numbers.

Git and GitHub both have the facility to *tag* any commit point, this tag must be unique, but it is entirely at the discretion of the user and can contain up to 25 characters.

The SCM numbering mechanism will use these *commit tags* to identify the particular revision of both a software module and a TIA Project.

The tags will also from the basis for naming the TIA Portal project at each commit point.

3.1 Workflow arrangements

The workflow within the Git repository consists of a single main branch, the **master** branch.

The **master** branch (after some initial development work to establish it) will, generally, only contain either finished (released) modules, software that has passed some level of testing or qualification or software that has been released for use.

Released modules are modules that have undergone a software module test (SMT) and have passed that test (i.e. a module that is deployable) —it does not indicate that all software modules are finished, just that the module in question is complete, tested and deployable.

When the software as a whole (all modules), has completed module testing, integration testing, has been commissioned and qualified, then the software as a whole will be released for use.

Development work can take place at any time and will always take place on a separate branch. Development branches always spur from some definite commit point on the **master** branch.

A development branch must have a very restricted scope. I.e. a single module or group of related modules.

Generally, a development branch will contain all the things associated with that module (i.e. the function, any data types, data blocks &c.).

When the module development is complete and tested, it will be merged back to the **master** branch, the merge point will be given a five-character tag (Figure 3.1):

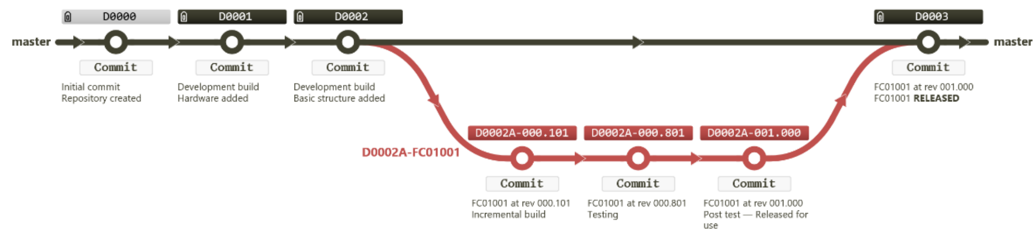


Figure 3.1 master branch workflow (typical)

3.2 Master branch revision states

The project progresses through various different states along the **master** branch. Each state is a commit point and this in turn has a tag with five characters. Each such commit point is referred to as a *primary commit*.

Each primary commit tag is given a letter that represents the condition of a particular commit point.

STATE	EXAMPLE	MEANING	DESCRIPTION
D	D0126	Development	The software as a whole is in the build phase and has not been fully tested. Certain modules may have undergone module testing and are released for use. This is on a module-by-module basis.
P	P0001	Proving (test)	The software is released for integration testing. All modules within the software have undergone module testing and have been released for use.
Q	Q0001	Qualification	The software is deployed for commissioning, installation qualification (IQ) and operational qualification (OQ).
R	R0001	Released	The software is released for use.

Table 3.1 **master** branch commit point tags

The **master** branch commit tags have the following format:

SNNNN

Where S is the state letter (Table 3.1):

- D—Development
- P—Proving (testing)
- Q—Qualification
- R—Released

NNNN is a number; this starts at 0001 (there is a special case for the first commit to the repository, this has value 0000) for each particular state and is incremented by one for each subsequent issue.

E.g. D0001 → D0002 → D0003 → P0001 → P0002 → R0001 → R0002 &c.

3.3 Development branch names

Generally, development never takes place along the **master** branch. The only exception to this is at the start of the project when the repository is created. The initial commit typically takes place on the **master** branch (this is always tagged D0000), and there may be subsequent commits on the **master** branch to establish the repository structure: folders, configuration files (e.g. `.gitignore` and `.gitkeep` files &c.) and other common repository files (`README.md`, `LICENCE.md` &c.).

Such development can take place along the **master** branch until some suitable point is reached; this point is usually where module development begins; after this, only minor changes will take place on the **master** branch, such changes will be to address any conflicts that occur when merging multiple development branches back to the **master** branch (see § 3.9), or to update some ancillary file information (e.g.`README.md`).

At this point, no significant development work can take place on the **master** branch.

Development work always takes place on a separate **development** branch.

A development branch must have a very restricted scope. I.e. a single module or group of related modules.

Each **development** branch is taken from the latest primary commit point on the **master** branch (generally referred to as the **HEAD**). The name given to a **development** branch is always in the format:

SNNNNb-MMYYYYY

Where SNNNN is the commit point tag on the **master** branch from which the development branch diverges.

The b character is an ordinal character identifying multiple branches that start from the same **master** branch commit point, the first branch receives character A, the second B, the third C &c.

The remainder of the branch name refers to the object being developed; these are generally software modules. MMYYYYY specifies the object under development, for example FC01001. It could equally apply to just a data type e.g. UT01000.

Here, MM refers to the type of module (OB, FB, FC, DB, UT &c.) and YYYYY to the module number (these are always numerical, five-digit numbers with leading zeros where necessary)[2].

This arrangement can be seen below:

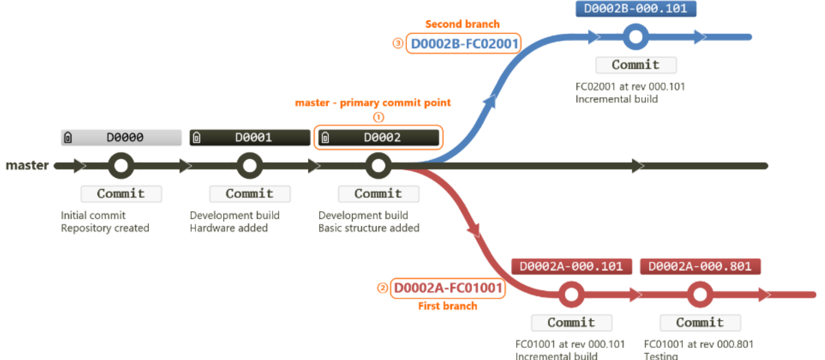


Figure 3.2 Multiple development branches from a **master** branch primary commit point

Here, two development branches diverge from the latest commit point on the **master** branch, point ①. The first branch is used to develop module FC01001, the second to develop module FC02001.

The first branch name takes the **master** branch commit point (D0002), followed by the ordinal character, since this is the first branch the ordinal character is A. Giving D0002A. This is followed by a dash (-) and the module number, in this case FC01001.

---

2 Where some other type of change is being made, for example, standardising an arrangement of comment information across multiple blocks, the MMYYYYY format can be replaced with some more meaningful name e.g. UNIFICATION &c.

The final branch name is thus D0002A-FC01001. Point ② in Figure 3.2.

The second branch is also attached to the **master** branch commit point D0002, but in this case it is the second branch, giving an ordinal character of B. In this case the module being developed is FC02001.

This gives a final branch name for the second branch of: D0002B-FC02001. Point ③ in Figure 3.2.

3.4 Development branch commit tags

All development work takes place on the development branch. There will be many such branches through the course of the Project.

Each development branch will consist of multiple commits, these commits are referred to as *secondary commits* (c.f. *primary commits* made on the **master** branch). These secondary commits will mostly be incremental builds (an incremental build is just a point at which the work was committed to preserve the software at a particular point, these incremental builds occur often, allowing the software to be recovered if necessary. The reasons behind an incremental build are at the discretion of the developer, it may be a significant point in the development of the software, alternatively, it may be just a commit because it was the end of the day).

Each secondary commit on a development branch is tagged, in the form:

`SNNNNb-nnn.amm`

Where SNNNNb is the first part of the branch name (before the dash), see § 3.3. This is the originating **master** branch commit point and the branch ordinal character.

The remaining characters (`nnn.amm`) are all numerical and reflect the current revision of the module under development, the details of this format are explained in § 3.6.

The development branch will be complete when the module being developed on that branch has successfully undergone its software module test and the module is at a release revision, at this point the **development** branch can be merged back onto the **master** branch.

3.5 Merging a development branch

When all the work on a development branch is complete, that branch can be merged back onto the **master** branch.

Consider the following example:

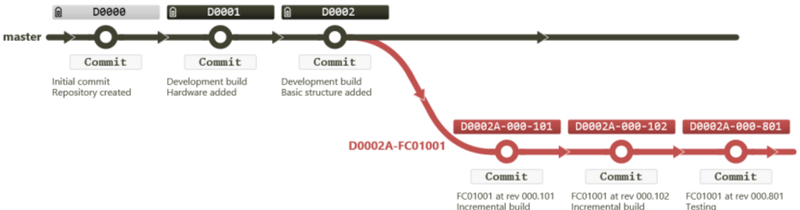


Figure 3.3 Example development branch

Here a development branch (D0002A-FC01001) was initiated from **master** branch commit point D0002.

There have been three commits on the development branch: D0002A-000.101, D0002A-000.102 and D0002A-000.801; the first two were incremental builds and the last was a software module test. Let us assume that the module passed its module test and is now finished (released for use).

There will now be an additional (and final) commit on the development branch to reflect the released revision of the module:

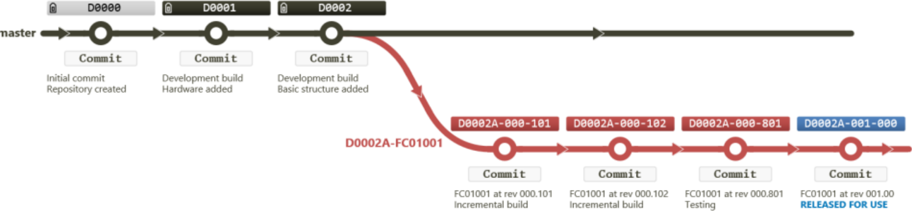


Figure 3.4 Example development branch, final commit

The development branch can now be merged back to the **master** branch, following the merge, a new *primary* commit point must be created on the **master** branch. This will have a revised revision data for OB1 (see § 3.7) and will have the format SNNNN.

This new commit point must be given the next, logical tag for the **master** branch. In this case, the last primary commit tag on the master branch was D0002, the next, logical primary tag is thus D0003 (an increment of one on the last master branch tag).

Note: *Here, there is a transition from one development tag to the next (D0002 to D0003). It would be perfectly possible for this to be a transition to a different state, i.e. it could be going from development (D) to proving (P), in which case the numbering restarts at 0001.*

Diagrammatically, this is:

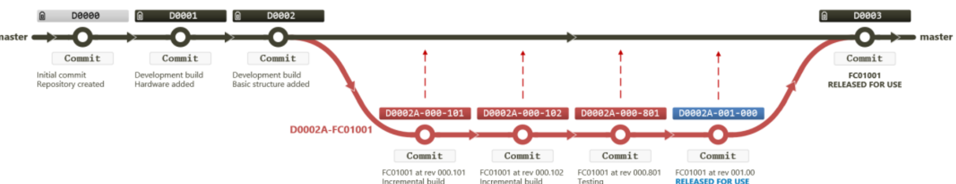


Figure 3.5 Example development branch, merge to **master**

Once the merge has taken place, all the secondary commits made on the development branch will become part of the **master** branch, thus:

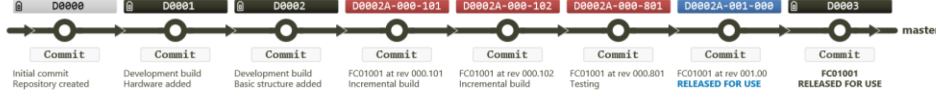


Figure 3.6 The **master** branch after the merge

Although the **master** branch now contains all the secondary commit point made on the development branch, none of them were made on the **master** branch itself (all this work happened on the development branch).

This arrangement is correct; ultimately, when the project is finished, there will only be the **master** branch left and this will contain every commit made within the project.

To more clearly understand the **master** branch, only the primary commit points (with just five characters) need be considered:

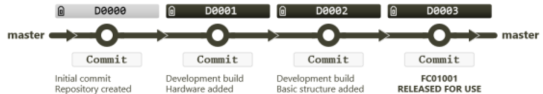


Figure 3.7 The **master** branch significant commit points

3.6 Individual module revision numbers

Every module within the PAL software has its own individual revision number, this was briefly referred to in § 3.4, when discussing the tags of a secondary commit. Each secondary commit on a development branch has a tag in the form:

`SNNNNb-nnn.amm`

Where `SNNNNb` is the first part of the branch name (before the dash), see § 3.3. This is the originating **master** branch primary commit point (the `SNNNN`) from which the development branch diverges and the branch ordinal character, `b`, (this will be `A` for the first, `B` for the second &c.).

The remaining characters (`nnn.amm`) reflect the individual revision number of the module being developed. The six digits are all decimal numerals.

The numbering of the revision `nnn.amm` is an incremental numbering system. In this system `nnn` reflects the current version of the software; typically, the first properly released software will be `001`. Previous development versions will be `000`.

The numbers after the decimal point (`amm`) reflect development and test modification to the current revision (for software modifications), in this system `a` reflects the current status of the software as follows:

FIRST DIGIT (a)	MEANING	DESCRIPTION
0	Released `mm` will be `00`	Code is released at version `nnn` (i.e. `nnn.000`)
1-7	Development	Code is under development and has not been tested
8	Proving	Proving (test) revisions of the software
9	Qualification	Software is deployed to site and is being commissioned or qualified

Table 3.2 Software revision number (first digit)

The remaining numbers (`mm`), are incremental build numbers for the current revision (this allows development tracking).

Note: *A release version of the software will have revision `001.000`, `002.000`, `003.000` &c. I.e. the numbers after the decimal point are all zero. The first development of the software at release `003` would have revision `003.101`.*

3.6.1 Recording revision numbers within a programmable block

All programmable blocks (with the exception of OB1, see § 3.7) have the current revision number stored in the first non-empty network (usually network 2, sometimes network 3 for blocks with a large, textual block descriptions) of the block.

The revision number is both hardcoded in the block and is stored (with additional information) within the network comments of that network.

Hardcoded module revision data

The hardcoded information is stored internally within the temporary area of the block as variable `revInfo`, this is of the user data type: *`UT01000_St_SysRevision`*:

DATA STRUCTURE	*UT01000_St_SysRevision*	
SIGNAL	TYPE	FUNCTION
REV_BLOCK	String[7]	Block number (of this block)
REV_NUMBER	String[20]	Revision status and by revision number (for this block)
REV_DATE	String[10]	Revision date in format YYYY-MM-DD
REV_AUTHOR	String[20]	Revision author (initial and surname) or username

Table 3.3 Data structure: UT01000_St_SysRevision

The purpose of this is to hardcode in a recoverable format the basic, necessary revision data of the particular module (hardcoded information will always be present and recoverable from the Controller, even if the code comments are lost):

- Block ID (the unique number of the block in question)
- Revision number (incorporating status information)
- Revision date
- Revision author

An example of this is shown below:

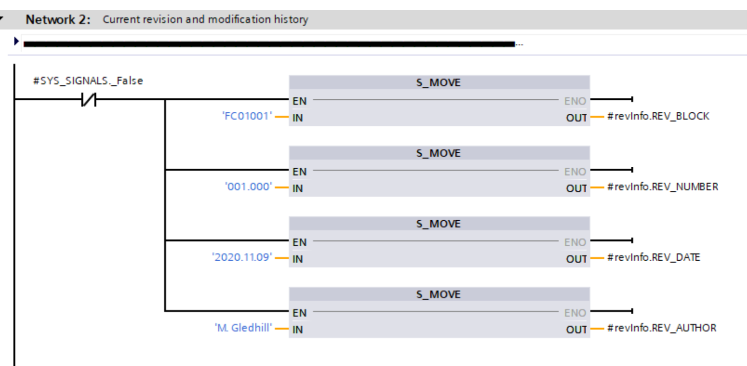
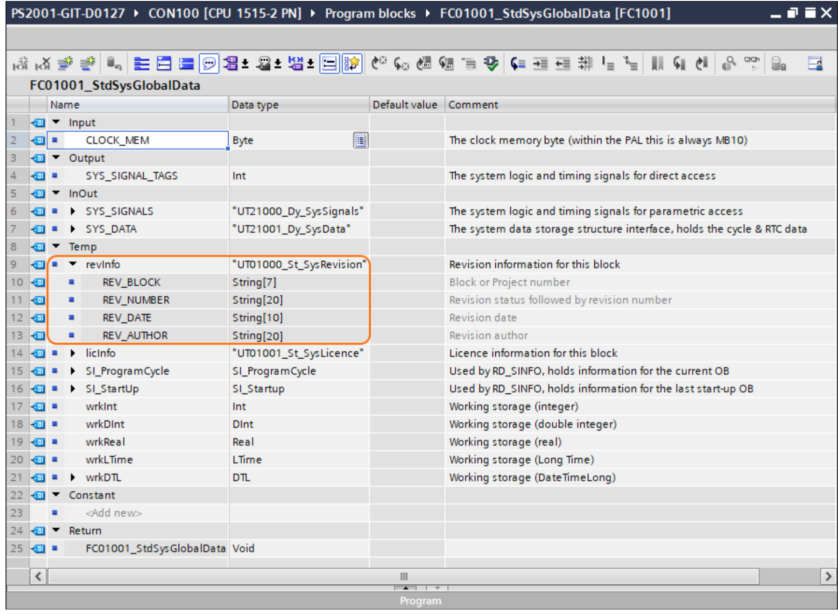


Figure 3.8 Block hardcoded revision information

The temporary variable `revInfo` is part of the block interface and is common to all PAL software modules (it must be defined and be present for all blocks within the PAL), an example is shown below:



PS2001-GIT-D0127 ▸ CON100 [CPU 1515-2 PN] ▸ Program blocks ▸ FC01001_StdSysGlobalData [FC1001]

FC01001_StdSysGlobalData

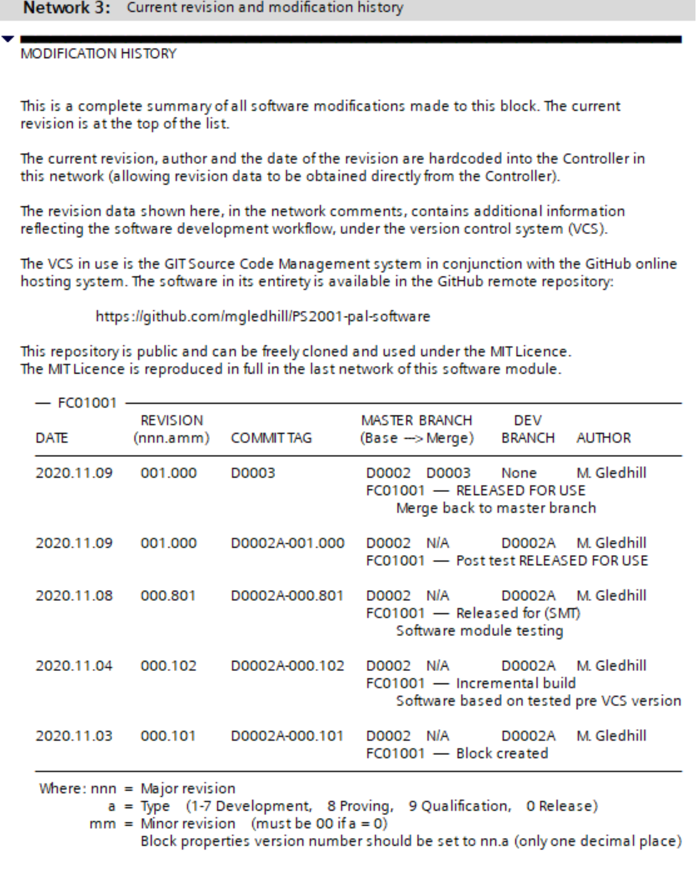
	Name	Data type	Default value	Comment
1	Input			
2	CLOCK_MEM	Byte		The clock memory byte (within the PAL this is always MB10)
3	Output			
4	SYS_SIGNAL_TAGS	Int		The system logic and timing signals for direct access
5	InOut			
6	SYS_SIGNALS	"UT21000_Dy_SysSignals"		The system logic and timing signals for parametric access
7	SYS_DATA	"UT21001_Dy_SysData"		The system data storage structure interface, holds the cycle & RTC data
8	Temp			
9	revInfo	"UT01000_St_SysRevision"		Revision information for this block
10	REV_BLOCK	String[7]		Block or Project number
11	REV_NUMBER	String[20]		Revision status followed by revision number
12	REV_DATE	String[10]		Revision date
13	REV_AUTHOR	String[20]		Revision author
14	licInfo	"UT01001_St_SysLicence"		Licence information for this block
15	SI_ProgramCycle	SI_ProgramCycle		Used by RD_SINFO, holds information for the current OB
16	SI_StartUp	SI_Startup		Used by RD_SINFO, holds information for the last start-up OB
17	wrkInt	Int		Working storage (integer)
18	wrkDInt	DInt		Working storage (double integer)
19	wrkReal	Real		Working storage (real)
20	wrkLTime	LTime		Working storage (Long Time)
21	wrkDTL	DTL		Working storage (DateTimeLong)
22	Constant			
23	<Add new>			
24	Return			
25	FC01001_StdSysGlobalData	Void		

Figure 3.9 Hardcoded module revision storage variable

Network comment module revision data

The network comments contain considerably more information about the revision and its point in the software development workflow, under the control of the VCS.

Figure 3.10 show an example of the network revision comments. These comments represent the example shown in § 3.5, reproduce in Figure 3.11 below.



Network 3: Current revision and modification history

MODIFICATION HISTORY

This is a complete summary of all software modifications made to this block. The current revision is at the top of the list.

The current revision, author and the date of the revision are hardcoded into the Controller in this network (allowing revision data to be obtained directly from the Controller).

The revision data shown here, in the network comments, contains additional information reflecting the software development workflow, under the version control system (VCS).

The VCS in use is the GIT Source Code Management system in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/mgledhill/PS2001-pal-software

This repository is public and can be freely cloned and used under the MIT Licence.
The MIT Licence is reproduced in full in the last network of this software module.

— FC01001 —

DATE	REVISION (nnn.amm)	COMMIT TAG	MASTER BRANCH (Base —> Merge)		DEV BRANCH	AUTHOR
2020.11.09	001.000	D0003	D0002	D0003	None	M. Gledhill
			FC01001 — RELEASED FOR USE Merge back to master branch			
2020.11.09	001.000	D0002A-001.000	D0002	N/A	D0002A	M. Gledhill
			FC01001 — Post test RELEASED FOR USE			
2020.11.08	000.801	D0002A-000.801	D0002	N/A	D0002A	M. Gledhill
			FC01001 — Released for (SMT) Software module testing			
2020.11.04	000.102	D0002A-000.102	D0002	N/A	D0002A	M. Gledhill
			FC01001 — Incremental build Software based on tested pre VCS version			
2020.11.03	000.101	D0002A-000.101	D0002	N/A	D0002A	M. Gledhill
			FC01001 — Block created			

Where: nnn = Major revision
a = Type (1-7 Development, 8 Proving, 9 Qualification, 0 Release)
mm = Minor revision (must be 00 if a = 0)
Block properties version number should be set to nn.a (only one decimal place)

Figure 3.10 Network comment revision information

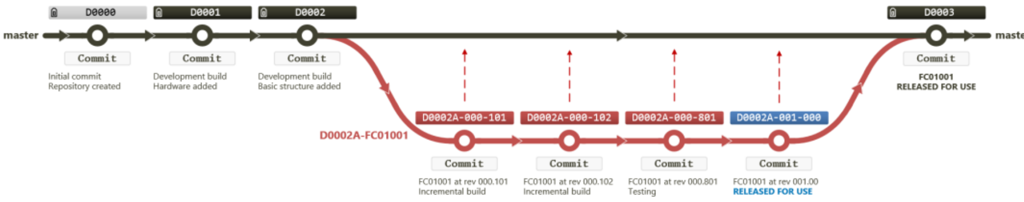


Figure 3.11 Example development branch, merge to **master**

Examining the network comments in more detail:

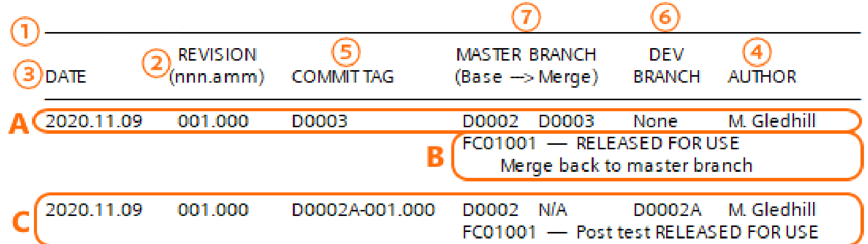


Figure 3.12 Network comment revision information details

Point ① is the start of the revision table

The information given in point ② to ④ is identical to the information hardcoded into the module:

- Revision number (incorporating status information)
- Revision date
- Revision author

Point ⑤ is the commit tag given to the commit when the software is added to the repository.

Point ⑥ identifies the development branch upon which the changes were made, only the first six characters are required (everything before the dash) to uniquely identify the branch.

Point ⑦, the MASTER BRANCH contains two entries: BASE and MERGE.

The BASE entry records the commit point on the main branch from which the development branch spurs away, in this example it is at the commit point with tag D0002:

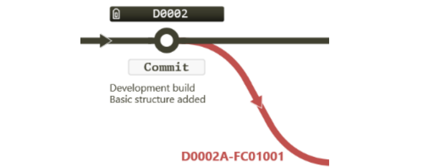


Figure 3.13 Base commit point (where a branch diverges)

The MERGE entry records the commit point tag at which the branch re-joins (*merges*) with the **master** branch. In this case it is at commit tag D0003:

Figure 3.14 Merge commit point (where a branch merges)

The BASE/MERGE entries are complete in Figure 3.12 for the final entry in the revision list (entry A), but the MERGE entry reads N/A (not applicable) in the preceding entries (entry C for example). The reason for this is that while the software is being developed on the D0002A branch, further developments may be taking placed on other branches (see § 3.9 for an explanation of this), and these branches may merge back to the **master** branch before this one (effectively occupying the next commit point tag).

It is not until the development branch is complete, and ready to be merged back to the master branch, that the final MERGE commit point tag will be known.

3.6.2 Recording revision numbers within a data block

Data blocks, both *static* and *dynamic*, like programmable blocks, have the revision information both hardcoded in the block and stored (with additional information) within the header comment area of the data block.

If the data block is being developed as part of the development of a software module, the development branch will have a label associated with the programmable block rather be directly associated with the data block (in the previous example, the branch was called D0002A-FC0100, labelled for the software module being developed: FC01001).

Data blocks are to some extent independent of the standard blocks with which they are associated, a new device may be added to a project and the associated data blocks will be modified (and their revisions changed) to accommodate it. The standard module within which the data blocks are used will not change.

If the data block were the sole focus of the development branch it would be permissible to label the branch for the data block in question (e.g. D0002A-DB21001).

Hardcoded data block revision data

The hardcoded information is stored as the first non-header variable of the data block. As with programmable blocks, the variable is called `revInfo`, and is again of the user data type: *`UT01000_St_SysRevision`*; this being the same data type used for programmable modules (see Table 3.3).

An example of this is shown below:

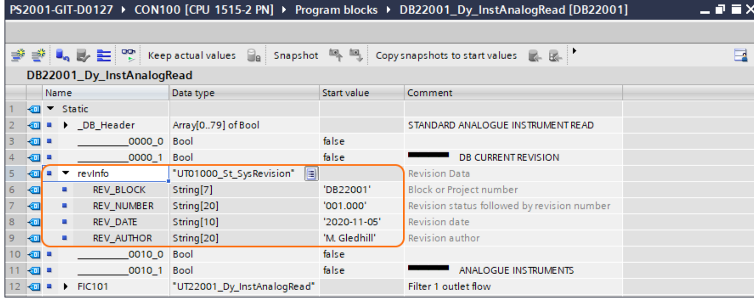


Figure 3.15 Hardcoded data block revision storage variable

Header comment data block revision data

The network comments for a DB contain the same type of information (and in the same format) as programmable blocks (see § 3.6.1).

Data blocks do not have the facility for network comments that is available to programmable blocks; however, all PAL data blocks are configured with a header array with variable name `DB_Header`, this is an array of 80 Boolean values and is used purely as a comment area for the data block. The revision information is contained within the comment area of this `DB_Header` array.

Figure 3.16 show an example of the data block header revision comments.

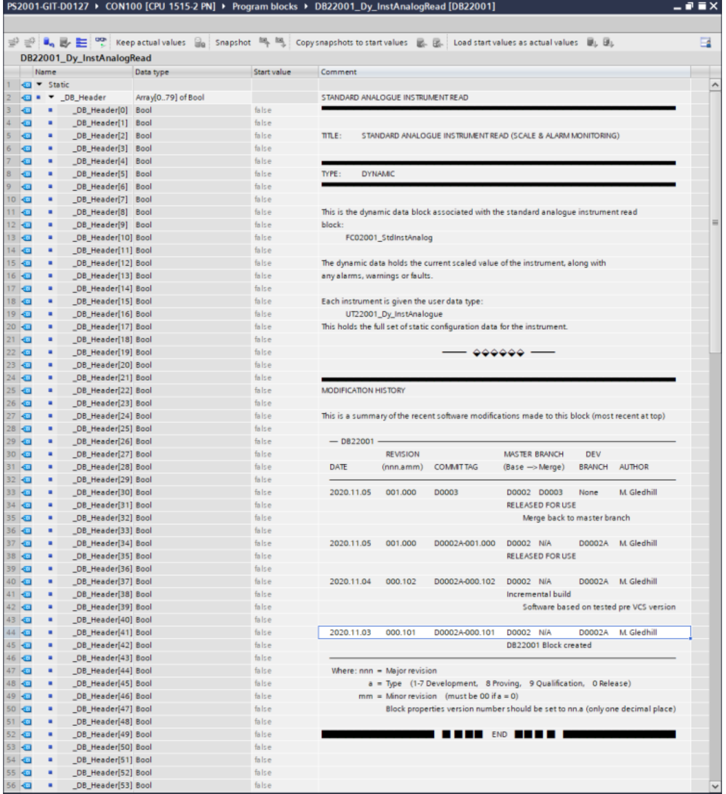


Figure 3.16 Header comment revision information

The header comments are applied in exactly the same way as the network comments of a programmable block (see page 33).

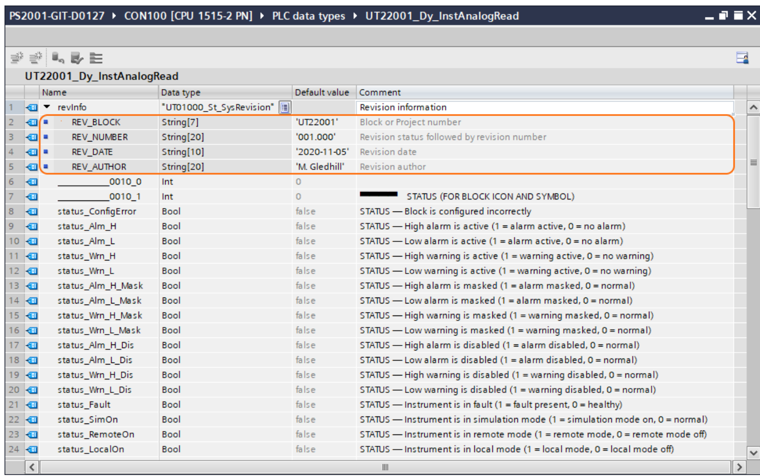
The `DB_Header` array is of a finite size and cannot accommodate unlimited comment information (unlike a programmable block), where the revision information becomes longer than the available space, the oldest revisions will be removed from the list (the revision information will still be recoverable from earlier commit points affecting that particular block).

3.6.3 Recording revision numbers within a User Data Type (UDT)

UDTs, both static and dynamic, have only hardcoded revision information and this holds only the current revision information, identical to the hardcoded data in a data block.

The hardcoded information is stored as a variable of the UDT. As with data blocks, the variable is called `revInfo`, and is again of the user data type: *`UT01000_St_SysRevision`*; this being the same data type used for data blocks and programmable modules (see Table 3.3).

An example of this is shown below:



PS2001-GIT-D0127 ▸ CON100 [CPU 1515-2 PN] ▸ PLC data types ▸ UT22001_Dy_InstAnalogRead

UT22001_Dy_InstAnalogRead

	Name	Data type	Default value	Comment
1	revInfo	"UT01000_St_SysRevision"		Revision information
2	REV_BLOCK	String[7]	'UT22001'	Block or Project number
3	REV_NUMBER	String[20]	'001.000'	Revision status followed by revision number
4	REV_DATE	String[10]	'2020-11-05'	Revision date
5	REV_AUTHOR	String[20]	'M. Gledhill'	Revision author
6	__________0010_0	Int	0	
7	__________0010_1	Int	0	STATUS (FOR BLOCK ICON AND SYMBOL)
8	status_ConfigError	Bool	false	STATUS — Block is configured incorrectly
9	status_Alm_H	Bool	false	STATUS — High alarm is active (1 = alarm active, 0 = no alarm)
10	status_Alm_L	Bool	false	STATUS — Low alarm is active (1 = alarm active, 0 = no alarm)
11	status_Wrn_H	Bool	false	STATUS — High warning is active (1 = warning active, 0 = no warning)
12	status_Wrn_L	Bool	false	STATUS — Low warning is active (1 = warning active, 0 = no warning)
13	status_Alm_H_Mask	Bool	false	STATUS — High alarm is masked (1 = alarm masked, 0 = normal)
14	status_Alm_L_Mask	Bool	false	STATUS — Low alarm is masked (1 = alarm masked, 0 = normal)
15	status_Wrn_H_Mask	Bool	false	STATUS — High warning is masked (1 = warning masked, 0 = normal)
16	status_Wrn_L_Mask	Bool	false	STATUS — Low warning is masked (1 = warning masked, 0 = normal)
17	status_Alm_H_Dis	Bool	false	STATUS — High alarm is disabled (1 = alarm disabled, 0 = normal)
18	status_Alm_L_Dis	Bool	false	STATUS — Low alarm is disabled (1 = alarm disabled, 0 = normal)
19	status_Wrn_H_Dis	Bool	false	STATUS — High warning is disabled (1 = warning disabled, 0 = normal)
20	status_Wrn_L_Dis	Bool	false	STATUS — Low warning is disabled (1 = warning disabled, 0 = normal)
21	status_Fault	Bool	false	STATUS — Instrument is in fault (1 = fault present, 0 = healthy)
22	status_SimOn	Bool	false	STATUS — Instrument is in simulation mode (1 = simulation mode on, 0 = normal)
23	status_RemoteOn	Bool	false	STATUS — Instrument is in remote mode (1 = remote mode, 0 = remote mode off)
24	status_LocalOn	Bool	false	STATUS — Instrument is in local mode (1 = local mode, 0 = local mode off)

Figure 3.17 Hardcoded UDT revision storage variable

UDTs are closely associated with a standard module, and any change to a UDT will cause a subsequent revision change within the associated module (after all, only the module can do something with the variables in the UDT). It is however, possible, and indeed common, for a change to the software module to have no effect on the UDTs associated with it.

For consistency, whenever there is a change to a UDT or to the standard module that uses that UDT, the UDT revision will be changed to match the released version of the standard module (even if there has been no change to the UDT). For example, if a standard module is changed in some way and released at revision `002.000`, all the UDTs that are associated with it will also be released at revision `002.000`.

In short, the released UDT revision should always match the revision of its parent software module.

3.6.4 Software Module Register (SMR)

A full list of all software modules is maintained in the Software Module Register (SMR) [Ref. 007].

This register contains the current revision of each module and the current revision of all its associated data blocks and UDTs.

3.7 OB1 module revision numbers

Each development branch concentrates (typically) on a single software module (usually a standard module that will form part of the PAL) with its associated data blocks and UDTs.

For development purposes, all these blocks are modifiable on a single development branch and are unlikely to be modified by work on other development (or any other) branch. In essence, the development takes place in isolation on its own branch.

The revision of the software module under development, its data blocks and UDTs are all recorded individually in each of the various blocks.

In addition to the module being developed, the main programme organisation block. OB 1 (more formally identified in the PAL as `OB00001_IntINrmMainProgram`), will also be modified, specifically to call the module under development.

OB 1 is considered a special block in the Practical Series Automation Library (and in terms of most Siemens Controller software). It is the block that executes all the rest of the controller software.

As such it contains information about the whole project rather than just a software module. The revision data is also project specific (not module specific).

OB 1 Network 2 contains the current revision of the whole *software project* (rather than of a particular block). In this regard the revision information contained in OB 1 does not follow the `nnn.amm` format specified for other programmable blocks; it simply

adopts the commit tag at the time of the commit, consider the previous example. In its final stage (at the point of merging the development back to the **master** branch), it had the following series of commit tags:

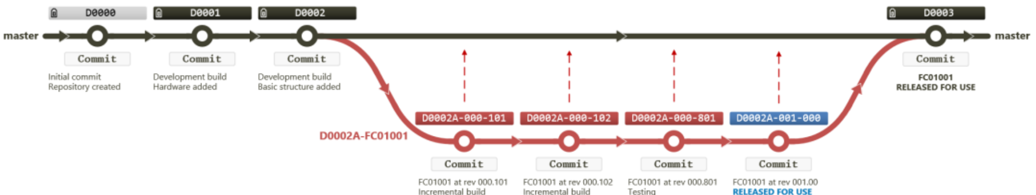
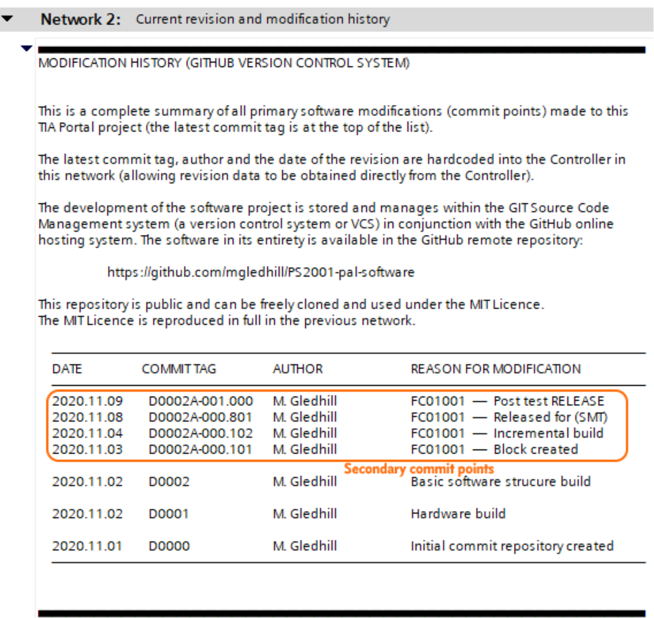


Figure 3.18 Example development branch, merge to **master**

At each commit point on the development branch, the OB 1 network comments would have recorded each commit, this can be seen below:



Network 2: Current revision and modification history

MODIFICATION HISTORY (GITHUB VERSION CONTROL SYSTEM)

This is a complete summary of all primary software modifications (commit points) made to this TIA Portal project (the latest commit tag is at the top of the list).

The latest commit tag, author and the date of the revision are hardcoded into the Controller in this network (allowing revision data to be obtained directly from the Controller).

The development of the software project is stored and manages within the GIT Source Code Management system (a version control system or VCS) in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/mgledhill/PS2001-pal-software

This repository is public and can be freely cloned and used under the MIT Licence.
The MIT Licence is reproduced in full in the previous network.

DATE	COMMIT TAG	AUTHOR	REASON FOR MODIFICATION
2020.11.09	D0002A-001.000	M. Gledhill	FC01001 — Post test RELEASE
2020.11.08	D0002A-000.801	M. Gledhill	FC01001 — Released for (SMT)
2020.11.04	D0002A-000.102	M. Gledhill	FC01001 — Incremental build
2020.11.03	D0002A-000.101	M. Gledhill	FC01001 — Block created
2020.11.02	D0002	M. Gledhill	Basic software strucure build
2020.11.02	D0001	M. Gledhill	Hardware build
2020.11.01	D0000	M. Gledhill	Initial commit repository created

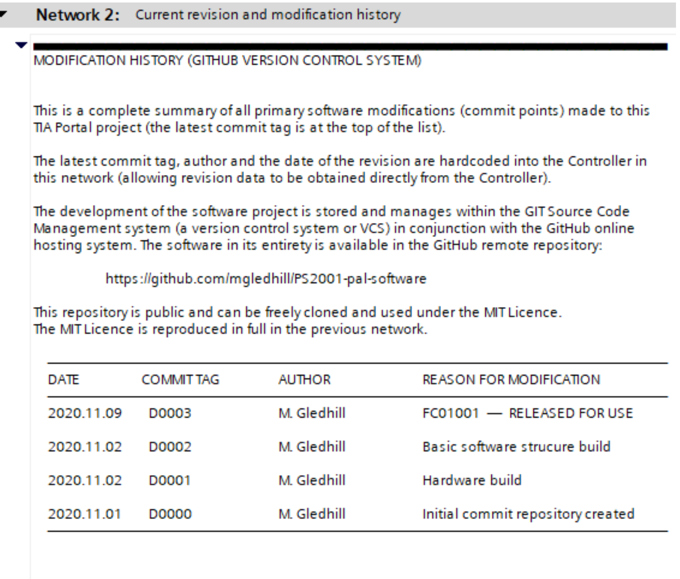
Figure 3.19 OB 1 revision history on the development branch

Here, it can be seen that the comments reflect the secondary commit points made on the development branch.

Each revision should be restricted to just one line in OB1.

Once the development branch has been merged back to the master branch, there will be an additional primary commit to reflect this; at this point, the secondary commits will be removed from OB 1.

The revision history contained in OB 1 at each primary commit point only shows the primary commit information. In this case the primary commit is `D0003` and the OB1 revision history is as follows:



Network 2: Current revision and modification history

MODIFICATION HISTORY (GITHUB VERSION CONTROL SYSTEM)

This is a complete summary of all primary software modifications (commit points) made to this TIA Portal project (the latest commit tag is at the top of the list).

The latest commit tag, author and the date of the revision are hardcoded into the Controller in this network (allowing revision data to be obtained directly from the Controller).

The development of the software project is stored and manages within the GIT Source Code Management system (a version control system or VCS) in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/mgledhill/PS2001-pal-software

This repository is public and can be freely cloned and used under the MIT Licence.
The MIT Licence is reproduced in full in the previous network.

DATE	COMMIT TAG	AUTHOR	REASON FOR MODIFICATION
2020.11.09	D0003	M. Gledhill	FC01001 — RELEASED FOR USE
2020.11.02	D0002	M. Gledhill	Basic software strucure build
2020.11.02	D0001	M. Gledhill	Hardware build
2020.11.01	D0000	M. Gledhill	Initial commit repository created

Figure 3.20 OB 1 revision history at a primary commit point

The OB 1 revision history is hardcoded in network 2, this is similar to the mechanism used for all other programmable blocks (see § 3.6.1), the difference is that the revision information is stored in a data block (all other programmable blocks store the revision information for the block in temporary storage within the block).

This can be seen here:

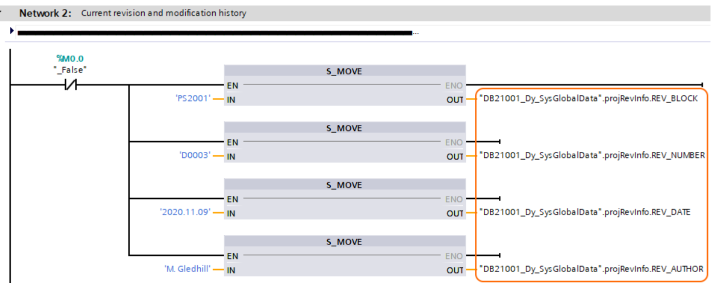


Figure 3.21 OB 1 hardcoded revision information

The block number is replaced with the project number (`PS2001` in this case), and the `S_MOVE` outputs are all passed to variables within data block `DB21001`.

OB 1 comments are slightly more complicated when multiple development branches exist, see § 3.10.2.

Note: *Where a commit is made directly on the* **master** *branch (for minor modification or to change ancillary files, see § 3.3), the revision of OB1 and the filename of the project must also change to reflect the new commit point tag.*

3.8 Commit points and filenames

The TIA Portal project, is saved at each commit point (both primary and secondary); the project is saved under a new filename at each commit point.

The filename is of the following format:

`PS2001-PAL-<commit tag>`

For example, a primary commit filename might be `PS2001-PAL-D0002` and a secondary commit file name `PS2001-PAL-D0002A-000-101`.

Note: *In the filename, any full stops (`.`) present in the commit tag field are replaced with dashes (`-`).*

The following shows the individual filenames for each of the commit points shown in the example of Figure 3.18, the filenames are shown in green:

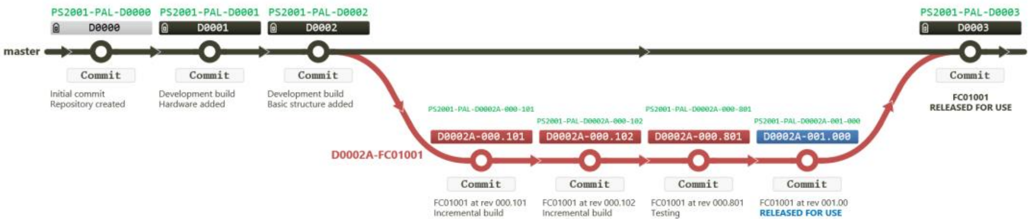


Figure 3.22 Commit point filenames

The project is saved at each commit point under the its new file name (see above), the project is also be archived at this point, using the archive facility within TIA Portal (PROJECT → ARCHIVE), this will produce a `.zap16` file with the same filename as the TIA Project. This is a compressed (zipped) file that can be used to recover the entire project. These `.zap16` files are all stored as archives on the Practical Series of

Publications network accessible storage (NAS) drives (section 5 explains the various folder structures and storage locations used by the Project).

3.8.1 OB 1 and filenames

The project filename is stored in network 1 of OB 1. This must be updated prior to each commit being made (in much the same way as the project revision, see § 3.7).

An example of the OB 1 network 1 project name is shown below:

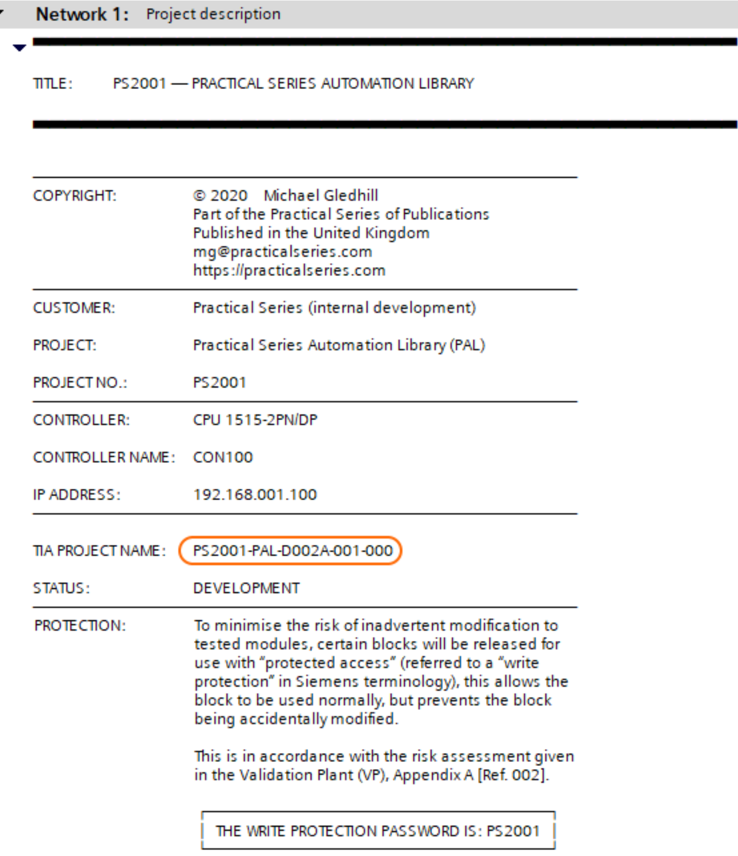


Figure 3.23 Project filename storage in OB 1

3.9 Parallel development branches

It is perfectly possible to have two (or more) simultaneous development branches:

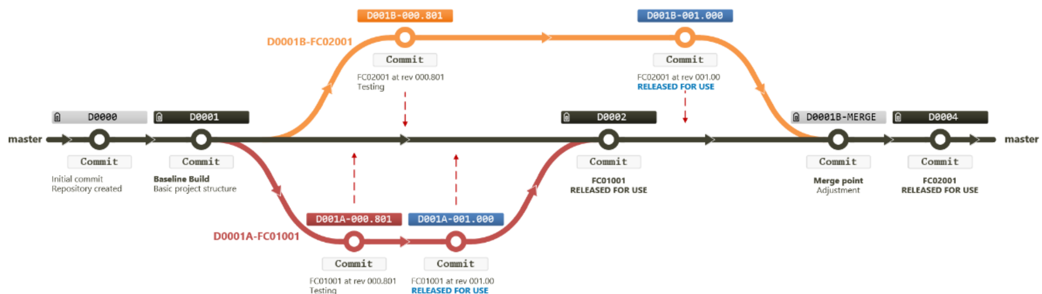


Figure 3.24 Parallel development branches

This type of arrangement can appear slightly confusing when all the branches are merged back onto the **master** branch:

Figure 3.25 Merged parallel development branches

All the commits are listed in order of the time they were applied.

Things are simplified if only the primary commits are considered:

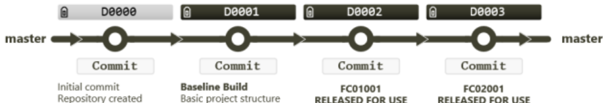


Figure 3.26 Merged parallel development branches, primary commits

From a workflow point of view, this is how the Project software should be viewed, a series of primary commits at which some part of the software was released for use.

Note: *The secondary commits are always present and can be recovered, however it is the primary commits that denote milestones in the software development.*

With parallel branches, it does not matter what order the branches are made or what order they merge back to the **master** branch. In the previous example, **D0001A** is created first, and is merged back to the **master** branch first. The following shows a similar arrangement with the first branch merging last:

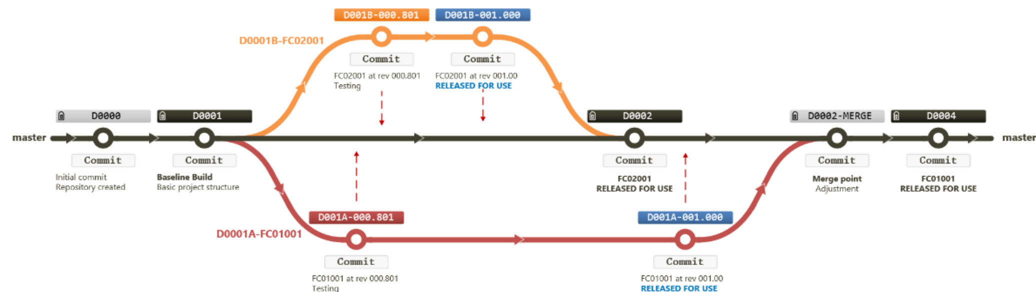


Figure 3.27 Parallel development branches with different merge order

Here, the second branch **D0001B** is created after **D0001A** but merges back before it; in this case the merged result would be:

Figure 3.28 Alternative Merged parallel development branches

And with just the primary commits:

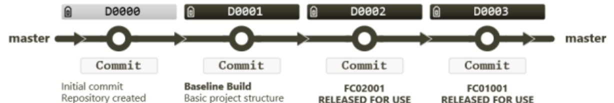


Figure 3.29 Alternative Merged parallel development branches, primary commits

3.10 OB 1 and the Merging of branches

Where software development takes place on individual development branches, this will generally involve modules that have no relation to each other, in the example of Figure 3.24, the first branch develops a particular function (`FC01001`) and the second branch a completely different module (`FC02001`), these two modules (and their associated data blocks and UDTs) could be merged back to the **master** branch without issue, all the modules developed on the first branch have no connection with the modules on the second branch and vice versa; indeed, each branch has no knowledge of the modules being developed on the other branch.

This complete independence of modules on the different development branches means that there is generally, no conflict when the branches are merged, all the modules of the first branch can be merged to the **master** branch, and when the second branch is merged, it has a completely separate set of modules that can also be merged without conflict.

There is however, one problem with this: OB 1. OB 1 contains revision information for the whole project (see § 3.7) and both branches will have a modified OB1 and both OB 1s will be different; this will not cause a problem when the first branch is merged back to the **master** branch, all the changes were on the development branch and will merge back to the **master** without any conflict (referred to as a *fast forward* merge in Git terminology). However, when the second branch is merged back, there **will be** a conflict with OB 1 (because it has been changed on both branches) and there needs to be some mechanism for reconciling the differences.

To accommodate this, the following section describe how branches should be merged together and how the primary commits are generated.

There are two types of merge, the first is where the first (or a single) development branch is merged back to the **master** branch, this is the easier merge because there will be no conflict. The second type is where additional branches are merged back, this will cause a conflict with earlier merges and this is handled slightly differently.

Examining each in turn:

3.10.1 Merging a single branch or the first branch to merge

Consider the following example from the previous section:

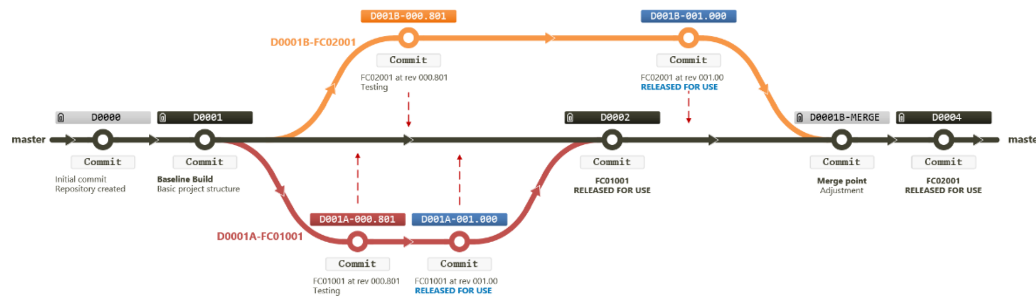
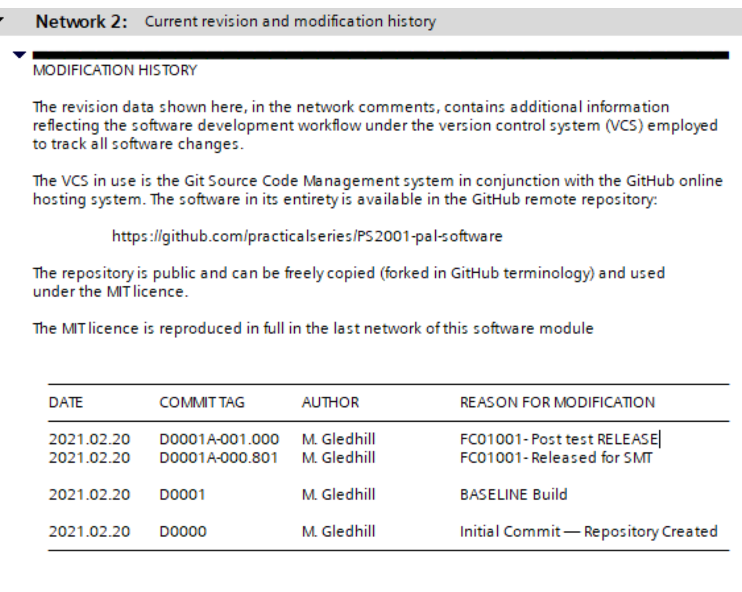


Figure 3.30 Parallel development branches

For the moment, consider only the first development branch to merge back to the **master** branch: **D0001A-FC01001**:

At the merge point, the last commit to have been made on the **D0001A-FC01001** branch was `D0001A-001.000`, at this point OB 1 had the following revision comments in it:



Network 2: Current revision and modification history

MODIFICATION HISTORY

The revision data shown here, in the network comments, contains additional information reflecting the software development workflow under the version control system (VCS) employed to track all software changes.

The VCS in use is the Git Source Code Management system in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/practicalseries/PS2001-pal-software

The repository is public and can be freely copied (forked in GitHub terminology) and used under the MIT licence.

The MIT licence is reproduced in full in the last network of this software module

DATE	COMMIT TAG	AUTHOR	REASON FOR MODIFICATION
2021.02.20	D0001A-001.000	M. Gledhill	FC01001- Post test RELEASE
2021.02.20	D0001A-000.801	M. Gledhill	FC01001- Released for SMT
2021.02.20	D0001	M. Gledhill	BASELINE Build
2021.02.20	D0000	M. Gledhill	Initial Commit — Repository Created

Figure 3.31 First branch merge OB 1 revision data

And the hardcoded revision was:

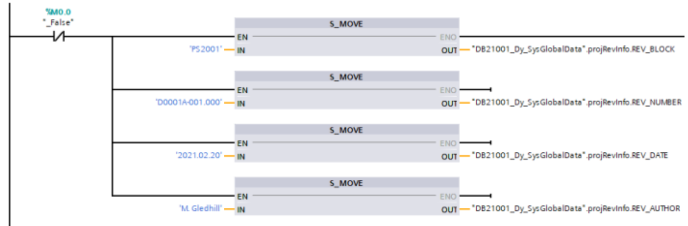


Figure 3.32 First branch merge OB 1 revision hard coded data

The merge will take place in Visual Studio Code (VSC) and will be made as a *"fast forward"* merge (this is the standard arrangement with VSC), this does not create a merge commit, it simple leaves the head at the last commit on the development branch.

After the merge, the **master** branch would be as follows:

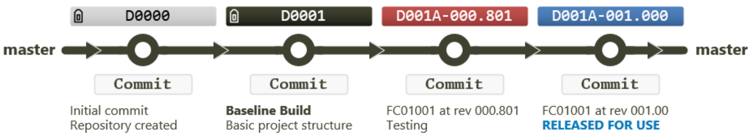


Figure 3.33 **master** branch after first merge

As yet there is no final D0002 primary commit. This commit is made directly on the **master** branch after the merge.

This may seem to contradict the "*no development work on the* **master** *branch*" rule (see § 3.3); however, adding this primary commit point is simply updating the revision status of OB 1 and cannot be considered development work.

There are three changes to be made to OB 1, the first two are changes to the revision information (both hardcoded and in the network comments), the third is to the file name (see § 3.8.1).

The changes to the hardcoded OB 1 revision (in network 2) are to update the revision to the primary commit tag (in this case D0002), as follows:

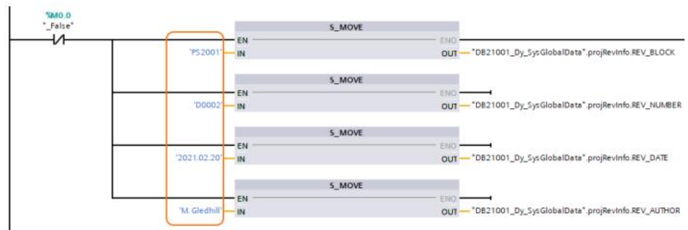


Figure 3.34 Primary commit point hardcoded update

The update to the network comment requires the removal of the secondary commit information and the addition of the primary commit revision:

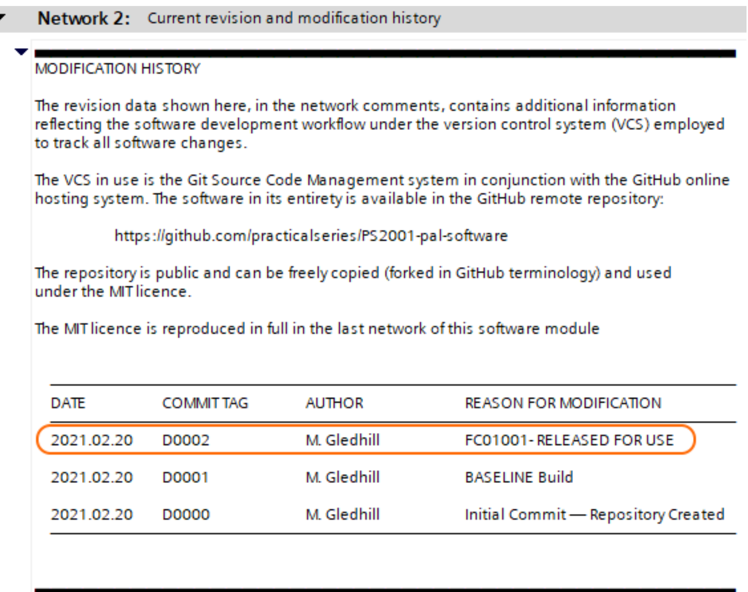


Figure 3.35 Primary commit point network comment update

The final change is to the TIA Portal project name in network 1 of OB 1:

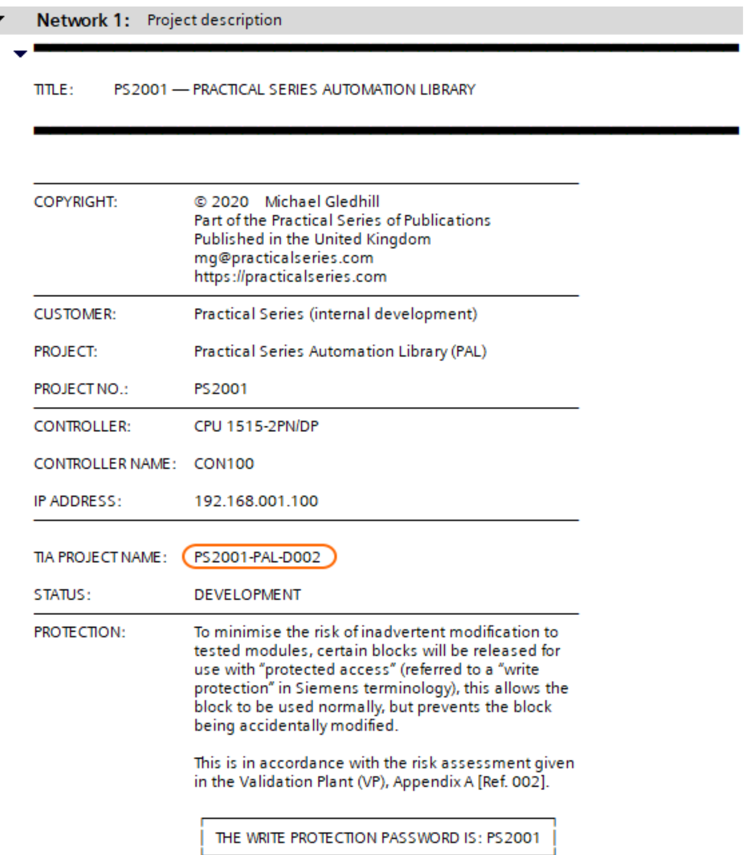


Figure 3.36 Primary commit point project filename

Section 3.8 gives details of TIA Portal project names and their association with commit points.

3.10.2 Merging additional parallel branches

Again, consider the example given in the previous section:

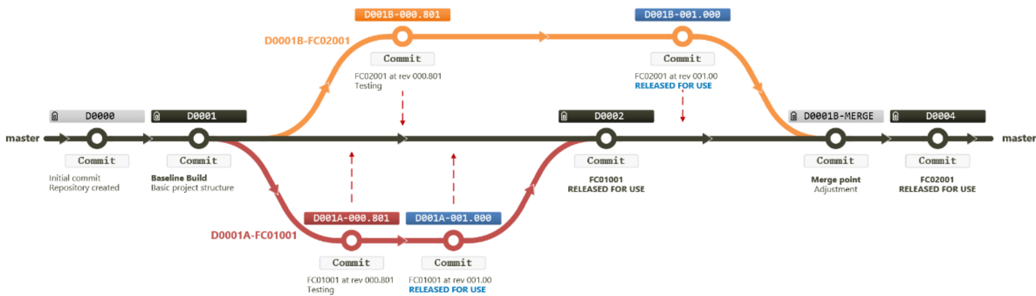


Figure 3.37 Parallel development branches

In the previous section, the first branch **D0001A-FC01001** was merged back to the master branch, leaving the overall workflow in the following state:

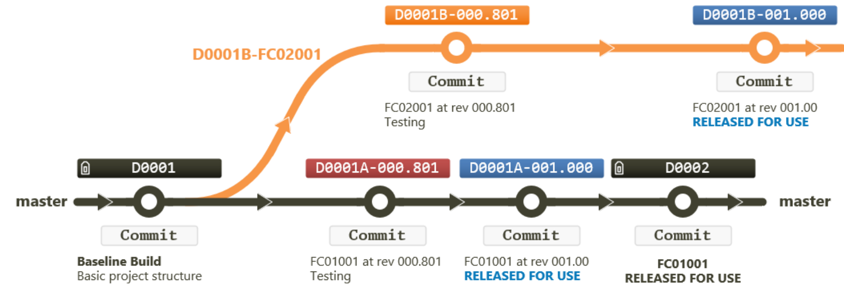
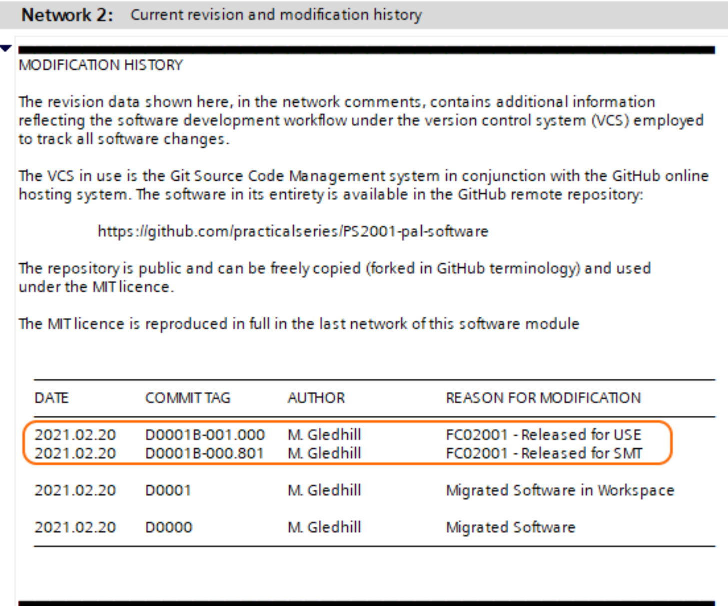


Figure 3.38 Parallel development after first branch merge

This is the point at which the second branch (**D0001B-FC02001**) is to be merged back to the **master**.

At this point, the last commit to have been made on the **D0001B-FC02001** branch was `D0001B-001.000`. Figure 3.39 shows the OB 1 revision comments at this point.

It should be noted at this point that the OB 1 comments do not contain any information about the secondary commits on the **D0001A-FC01001** branch or the D0003 primary commit point, this is because all those commits took place on other branches (either the **D0001A-FC01001** branch or the **master** branch) and are at this stage unknow to the **D0001B-FC02001** development branch.



Network 2: Current revision and modification history

MODIFICATION HISTORY

The revision data shown here, in the network comments, contains additional information reflecting the software development workflow under the version control system (VCS) employed to track all software changes.

The VCS in use is the Git Source Code Management system in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/practicalseries/PS2001-pal-software

The repository is public and can be freely copied (forked in GitHub terminology) and used under the MIT licence.

The MIT licence is reproduced in full in the last network of this software module

DATE	COMMIT TAG	AUTHOR	REASON FOR MODIFICATION
2021.02.20	D0001B-001.000	M. Gledhill	FC02001 - Released for USE
2021.02.20	D0001B-000.801	M. Gledhill	FC02001 - Released for SMT
2021.02.20	D0001	M. Gledhill	Migrated Software in Workspace
2021.02.20	D0000	M. Gledhill	Migrated Software

Figure 3.39 Second branch merge OB 1 revision data

With the hardcoded revision:

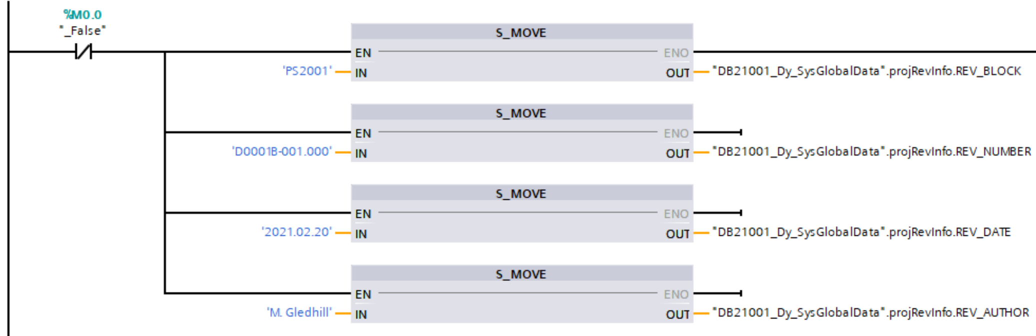


Figure 3.40 Second branch merge OB 1 revision hard coded data

Again, the merge will take place in VSC and will again be made as a fast forward merge.

This will do two things, it will merge `FC02001` on to the master branch with commit tag `D0001B-001.000`.

Secondly, it will indicate a conflict in OB 1, this is because OB 1 has been modified both on the **D0001A-FC01001** branch (now merged to the **master** branch) and on the **D0001B-FC02001** branch.

This can be seen in the SOURCE CONTROL state of Visual Studio Code (VSC):

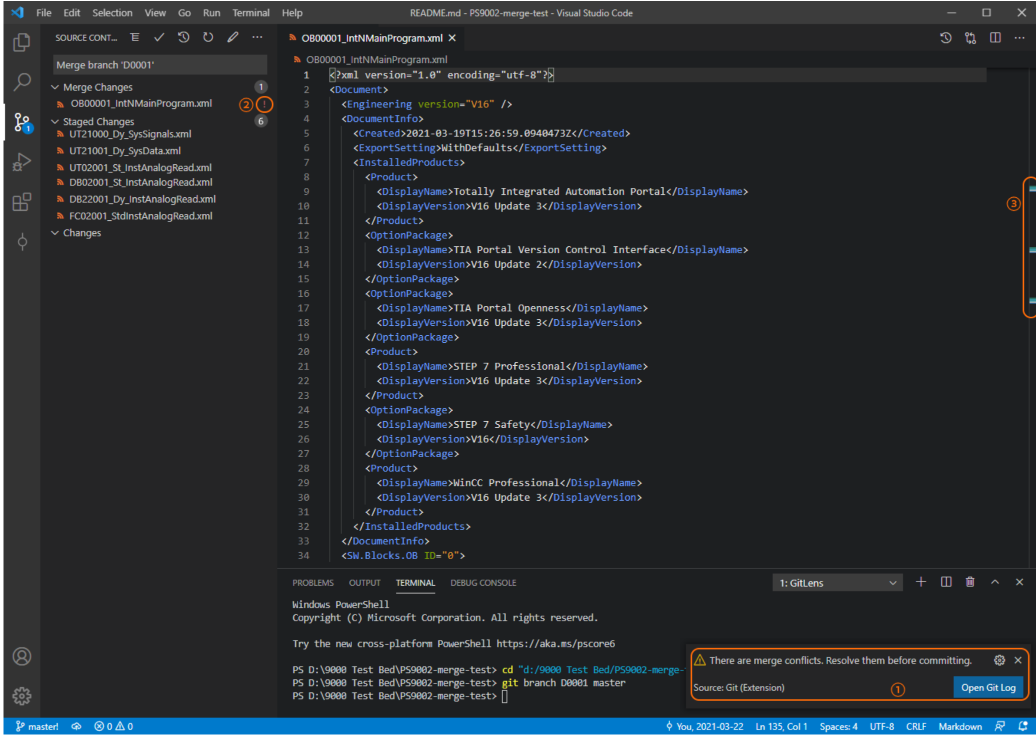


Figure 3.41 VSC merge conflict indication

The commit hasn't been made at this stage; this is because there is a conflict in one of the files.

This can be seen in point ① in Figure 3.41, the affected files are listed under MERGE CHANGES point ②, here it is just OB 1 that has a conflict (conflicted files are indicated by the red exclamation mark).

All the conflict free files (the ones that will merge without any issues) are showing as STAGED CHANGES.

To allow the commit to take place, the OB 1 modifications will be discarded, this is done by right clicking the OB 1 file in MERGE CHANGES point ② in Figure 3.41 and selecting ACCEPT ALL CURRENT in the dropdown menu (the current being the current or, in this case the **master** branch).

The merge can now be committed, in this case with commit message **D0001B-MERGE**.

This will commit all the changes from the **D0001B-FC02001** branch, but leave OB 1 as it was at the **D0002** commit point, the list of commits on the **master** branch is shown below.

All the commits are there from both branches, the master branch has the following:

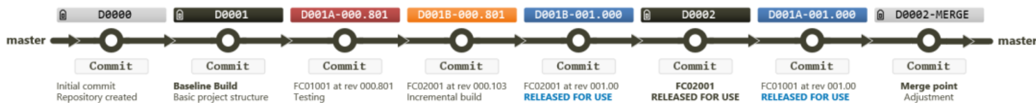


Figure 3.42 Merge intermediate commit

However, another primary commit now needs to be made on the **master** branch, this will be **D0003**, and this must include the updates made to OB 1, in the last commit on the **D0001B-FC02001** branch.

This is similar to the changes made to OB 1 for commit D0002 (see § 3.10.1); it should be noted at this point that OB 1 is currently in the same state as it was at D0002, the last primary commit on the **master** branch

Again, there are three changes to be made to OB 1, the first two are changes to the revision information (both hardcoded and in the network comments), the third is to the file name (see § 3.8.1).

The changes to the hardcoded OB 1 revision (in network 2) are to update the revision to the primary commit tag (in this case D0003), as follows:

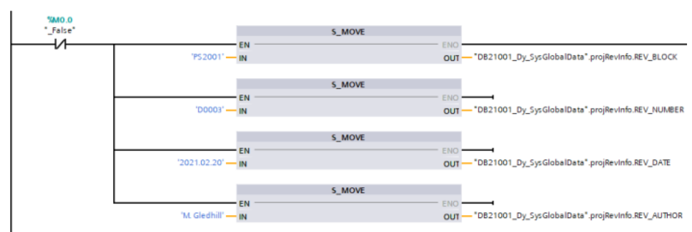
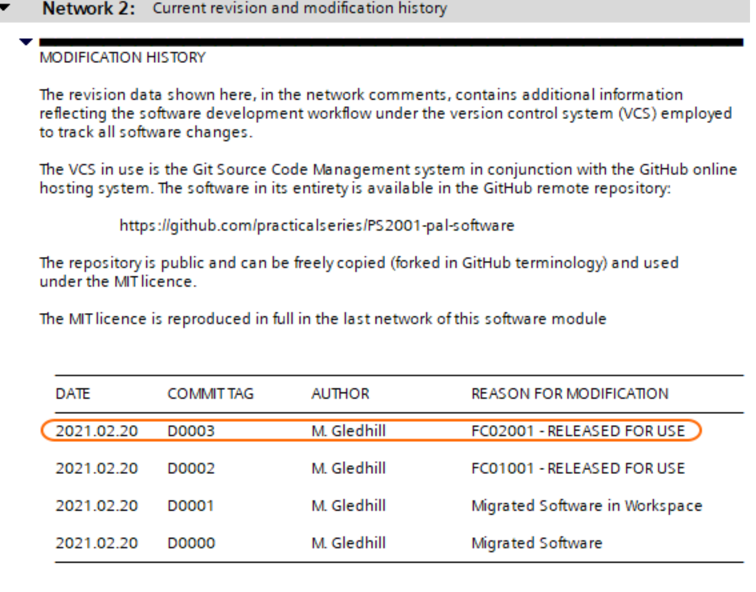


Figure 3.43 D0003 primary commit point hardcoded update

The update to the network comment requires the removal of the secondary commit information and the addition of the primary commit revision:



Network 2: Current revision and modification history

MODIFICATION HISTORY

The revision data shown here, in the network comments, contains additional information reflecting the software development workflow under the version control system (VCS) employed to track all software changes.

The VCS in use is the Git Source Code Management system in conjunction with the GitHub online hosting system. The software in its entirety is available in the GitHub remote repository:

https://github.com/practicalseries/PS2001-pal-software

The repository is public and can be freely copied (forked in GitHub terminology) and used under the MIT licence.

The MIT licence is reproduced in full in the last network of this software module

DATE	COMMIT TAG	AUTHOR	REASON FOR MODIFICATION
2021.02.20	D0003	M. Gledhill	FC02001 - RELEASED FOR USE
2021.02.20	D0002	M. Gledhill	FC01001 - RELEASED FOR USE
2021.02.20	D0001	M. Gledhill	Migrated Software in Workspace
2021.02.20	D0000	M. Gledhill	Migrated Software

Figure 3.44 D0003 primary commit point network comment update

Finally, the TIA Portal project name in network 1 of OB 1:

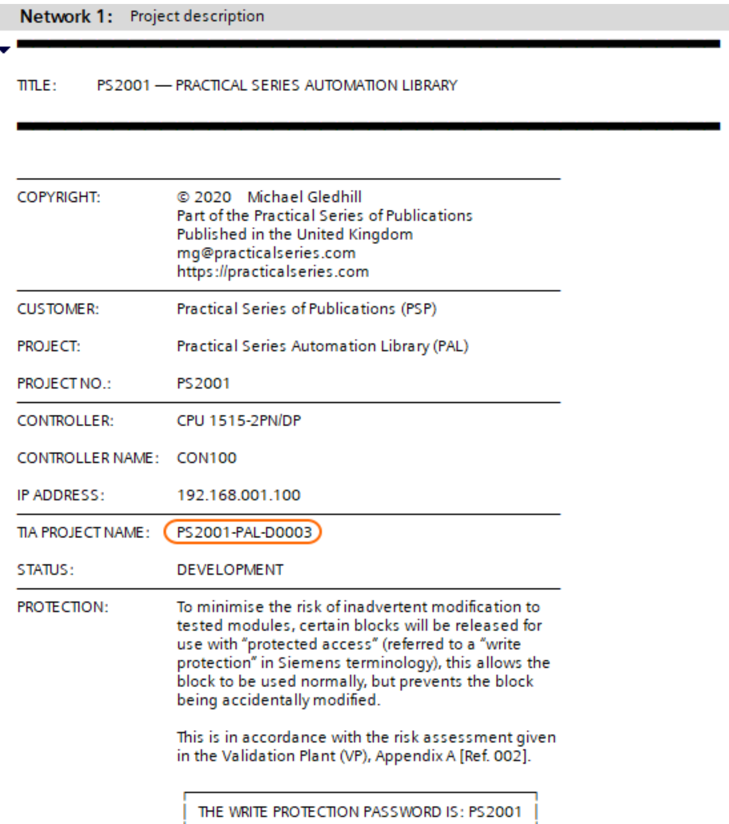


Figure 3.45 D0003 primary commit point project filename

Other changes to OB 1 may be required, if additional information is stored (such as a summary of completed modules &c.).

3.11 Nested branches

It is possible to have a development branch from another development branch (referred to as nesting). Nested branches always merge back onto their parent branch:

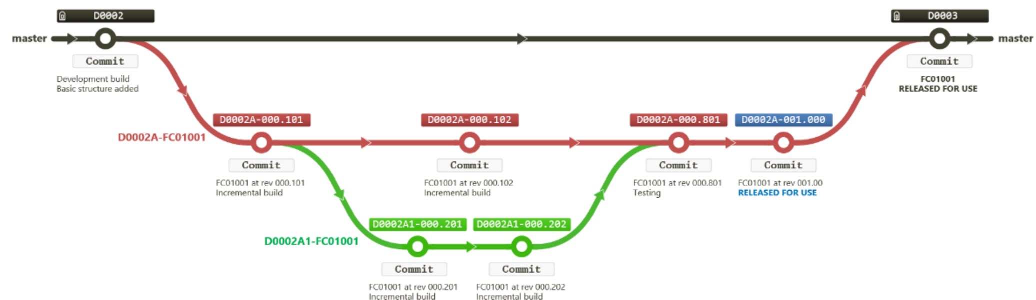


Figure 3.46 Nested development branches

The nested development branch name has an extra character before the dash, this is another ordinal number, identifying the number of the nested branch. The rest of the branch name is as § 3.3:

`SNNNNbX-MMYYYYY`

The extra character (`X`) starts at 1 for the first nested branch and incremented by 1 for each additional nested branch.

Each commit on the nested branch has the format:

`SNNNNbX-nnn.amm`

I.e. identical to the those of § 3.4, with the addition of the (`X`) character. Generally, the `a` value should be incremented by 1 to identify a separate development state.

3.12 A note on commit messages

Commit messages should have a short (less than 50 characters) first line. In Visual Studio Code (VCS), extended commits are possible (these are commits where more than one line can be entered), and the Commit Message Editor extension makes the configuration of commit messages into a standardised form-based format.

The Commit Message Editor settings should be adjusted to match the following settings:

settings.json

```
"commit-message-editor.tokens": [
    {
        "label": "Type",
        "name": "type",
        "type": "enum",
        "options": [
            {
                "label": "---",
                "value": ""
            },
            {
                "label": "PS (Mas) - Dev",
                "description": "PS master branch - development"
            },
            {
                "label": "PS (Mas) - Merge",
                "description": "PS master branch - merge point adjustment"
            },
            {
                "label": "PS (Mas) - Prove",
                "description": "PS master branch - proving (test)"
            },
            {
                "label": "PS (Mas) - Qual",
                "description": "PS master branch - qualification"
            },
            {
                "label": "PS (Mas) - Release",
                "description": "PS master branch - released for use"
            },
            {
                "label": "PS (Dev) - Dev",
                "description": "PS development branch - development"
            },
            {
```

```
                "label": "PS (Dev) - Merge",
                "description": "PS development branch - merge point adjustment"
            },
            {
                "label": "PS (Dev) - Prove",
                "description": "PS development branch - proving (test)"
            },
            {
                "label": "PS (Dev) - Qual",
                "description": "PS development branch - qualification"
            },
            {
                "label": "PS (Dev) - Release",
                "description": "PS development branch - released for use"
            },
            {
                "label": "PS (Gen) - Rev",
                "description": "PS development branch - revision update"
            },
            {
                "label": "PS (Gen) - Type",
                "description": "PS development branch - typographical changes only"
            },
        ],
        "description": "Type of changes"
    },
{
    "label": "Commit Tag",
    "name": "scope",
    "description": "The commit tag that will be applied to this commit (e.g. D0002B-0.101)",
    "type": "text",
    "multiline": false,
    "prefix": "[",
    "suffix": "]"
},
{
    "label": "Commit Title",
    "name": "description",
    "description": "Commit title line text",
    "type": "text",
    "multiline": false
},
{
    "label": "Body",
    "name": "body",
    "description": "Optional body",
    "type": "text",
    "multiline": true,
    "lines": 10,
```

```
    "maxLines": 100
},
// {
//     "label": "Footer",
//     "name": "footer",
//     "description": "Optional footer: ",
//     "type": "text",
//     "multiline": false,
// }
],
"commit-message-editor.view.defaultView": "form",
"commit-message-editor.dynamicTemplate": [
    "{scope} — {description}",
    "",
    "{type}",
    "",
    "  ________________________________________",
    "",
    "{body}",
    " ",
    "  ________________________________________",
    "Those aspects of the PAL project that have been migrated to the GitHub",
    "Version Control System (VCS) are operating under the Software Control",
    "Mechanism (SCM) specified in document PS2001-5-2302-011:",
    "   https://practicalseries.com/2001-pal/31-git/11-00-scm.html"
],
```

Code 3.1 Visual Studio Code — Commit Message Editor settings

This arrangement gives a common form that can be used to enter and edit a commit message before making the commit. It has the following appearance:

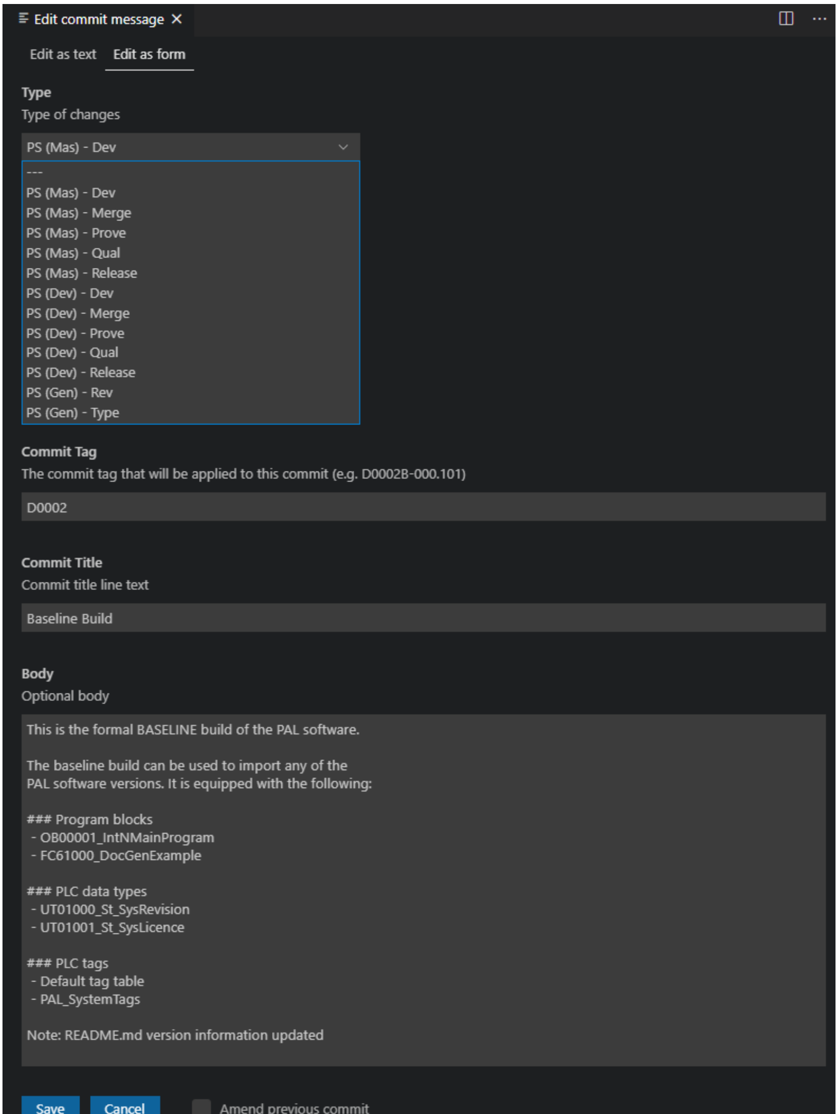


Figure 3.47 Visual Studio Code — Commit Message Editor form

The TYPE field is a dropdown selection that indicates the type of commit being made, those beginning PS(MAS) are commits directly on the master branch, those beginning PS(DEV) are on a development branch, those beginning PS(GEN) are of a general nature (not generally applicable to the software itself — e.g. a change to a README file or some file that is external to the Controller software).

The entry following the dash indicates the phase of the software as follows:

Entry	Meaning
Dev	Development commit (incremental build &c.)
Merge	Branch merge point
Prove	Proving (the software is at a test commit)
Qual	Qualification (the software is at a qualifying commit)
Release	Software (or module) is released for use
Rev	Revision change only
Type	Typographical corrections

Table 3.4 Commit message type field

The Commit tag is the commit tag that will be applied to the commit.

Commit Title is the first line of the commit, the first line is always shown in bold to indicate that it is a title.

The Body field contains the description of the changes being made at this commit point.

The completed commit has the following appearance:

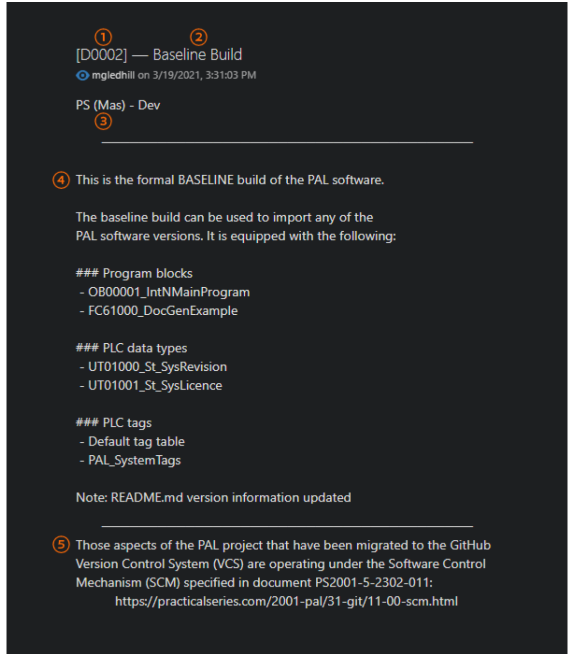


Figure 3.48 Visual Studio Code — Actual commit message

Where:

① Is the COMMIT TAG

② Is the COMMIT TITLE

③ Is the TYPE

④ Is the BODY

⑤ A common footer attached to all commit messages

4 The website revision numbering mechanism

The revision numbering mechanism and version control systems discussed in the previous sections was associated with the PAL software developed for the Siemens Controllers using TIA Portal and stored in the `PS2001-pal-software` GitHub repository (see § 5.2).

In addition to this repository, there is a second repository `PS2001-pal-website` that stores the website pages that are published in association with the PAL software (see § 5.3 for details of the website and associated GitHub repository).

This website is also developed using the same revision numbering mechanism detailed in the previous section, there are however, some minor differences, particularly in the naming of branches. These differences are discussed in the following sections.

4.1 Workflow arrangements

The website Git repository, like the software Git repository consists of a single main, **master** branch and various development branches.

The **master** branch (after some initial development work to establish the repository) will, generally, only contain released web pages

Released web pages are pages that have been built, tested and are complete and ready for live use on the website server.

As with the PAL software, development work can take place at any time and will always take place on a separate development branch and each development branch will spur from some definite commit point on the **master** branch.

With the website, each development branch has a very limited scope, usually the development of a single web page or series of webpages that are linked together (a section of the website for example). The branch may also be concerned with developing a background element of the website such as a particular JavaScript, jQuery or CSS file.

4.2 Master branch revision states

The website development progresses through various different states along the **master** branch in the same way as the PAL software (see § 3.2), where each state is a primary commit point. There are however, fewer states that can be applied to the website:

STATE	EXAMPLE	MEANING	DESCRIPTION
D	D0126	Development	The website as a whole is in the build phase and has not been fully tested. Certain web pages may have been developed and tested and released for use. This is on a page-by-page basis
P	P0001	Published	The website is published for testing. All web pages within the software are present, and the site as a whole is being tested on a live website for consistency checks, final proof reading and link integrity.
R	R0001	Released	The website is released for use

Table 4.1 **master** branch commit point tags

The **master** branch commit tags have the same format as the PAL software:

SNNNN

Where S is the state letter (Table 4.1):

- D—Development
- P—Published
- R—Released

NNNN is a number; this starts at `0001` (there is a special case for the first commit to the repository, this has value `0000`) for each particular state and is incremented by one for each subsequent issue.

E.g. D0001 → D0002 → D0003 → P0001 → P0002 → R0001 → R0002 &c.

4.3 Development branch names

The main difference between the website workflow and the PAL software workflow is in the naming of development branches.

The website development like the PAL software development branches will always diverge from the latest primary commit point on the **master** branch. The difference is the name, the website development branch name is in the format:

`SNNNNb-WW-PageName`

Where `SNNNN` is the commit point tag on the **master** branch from which the development branch diverges and `b` is the ordinal character identifying multiple branches (this is identical to the PAL software in § 3.3.

The remainder of the branch name refers to the section of the website and the web page being developed. `WW` refers to the number main folder of the website (see § 5.3.2):

`00-comres`	Common resources
`01-admin`	Various administration pages
`11-web`	The main website containing the PAL user guides and information
`21-project`	Holds all the documentation associated with the Project (validation documents)
`31-git`	Contains information used by the GitHub repositories
`81-binary`	Contains binary files (the TIA Project archive files &c.)
`91-userdocs`	The online version of the User Documentation files embedded in the TIA Project

I.e. `WW` will be `00`, `01`, `11`, `21`, `31`, `81` or `91`.

The `PageName` is the name of the HTML file for the particular page, an example being:

`11-00-scm.html`

This file (`11-00-scm.html`) is part of the `31-git` folder of the website, hence the `WW-PageName` part of the branch name would be (don't include the file extension):

`31-11-00-scm`

The `PageName` consists of a pair of number (`11-00` in the above example) followed by a short description indicating the purpose of the web page (`scm` in the example). The number pair is always in the format `cc-ss` where `cc` is the *"chapter"* number of the web page and `ss` is the *"section"*. The web pages are assigned chapter and section numbers like a document (e.g. this part of this document is chapter 4, section 3; its web page equivalent would thus be `04-03`).

The development branch is generally associated with a particular web page, and is named accordingly, however, the development branch will include all the files associated with, and required by that web page (i.e. the development branch will include the various CSS, JavaScript, jQuery, image files and binary files needed for the web page to work properly).

The `index.html` file associated with each of the main folders does not have any leading numbers in its file name (it is just called `index.html`), however, for the sake of consistency, such index files are given the number `00-00` in the branch name. E.g.:

```
31-00-00-index
```

Where the development branch is associated with a section (multiple pages) of the website, the branch name will use the chapter number only (just the `cc` part), for example, if a branch were developing the introduction pages of the website in the `11-web` folder, its branch name would be:

```
11-01-intro
```

Where the development branch is for a particular file, rather than a web page (this is usually where a common file that affects the whole website is being change or created) then the branch name will have the format:

```
SNNNNb-00-FileName
```

The common files are always stored in the `00-comres` folder (hence `WW` will always be `00`), the file name is the name of the file being modified (without the extension) for example, if the main style.css file were being modified, the branch name would be:

```
SNNNNb-00-style
```

4.4 Development branch commit tags

Development branch commit tags (secondary commits) are identical to those of the software development branches (§ 3.4); where each secondary commit is tagged and the tag has the format:

`SNNNNb-nnn.amm`

Where `SNNNNb` is the first part of the branch name (before the dash), see § 4.3. This is the originating **master** branch commit point and the branch ordinal character.

The remaining characters (`nnn.amm`) are all numerical and reflect the current revision of the web page (or file) under development, the format of this revision number is similar to that of the PAL software and is explained in § 3.6.

4.5 Merging of development branches

The merging of development branches is generally a simplified arrangement of that used for the PAL software (see §§ 3.4 and 3.5).

Development branches within the website are usually mutually exclusive and have no impact on each other (this is different to the PAL software where multiple branches usually have some degree of commonality, particularly with OB 1). This exclusiveness means that multiple branches can be merged without any conflict and the approach taken in § 3.5 for merging a single development branch or the first of multiple development branches can be adopted for all development branches within the website repository.

Exceptions exist where common table of contents (TOC) are being modified (and possibly where two simultaneous branches exist to modify a common file).

To minimise such conflicts, it is generally better to manage the workflow such that simultaneous (parallel) branches do not target common resources. Better to manage common files and common TOCs in a single development branch.

4.6 Individual page and file revision numbers

All the files associated with a web page (HTML, CSS, JavaScript and jQuery) have their own revision number. The current file revisions for all components of a web page are displayed at the top right of all web pages:

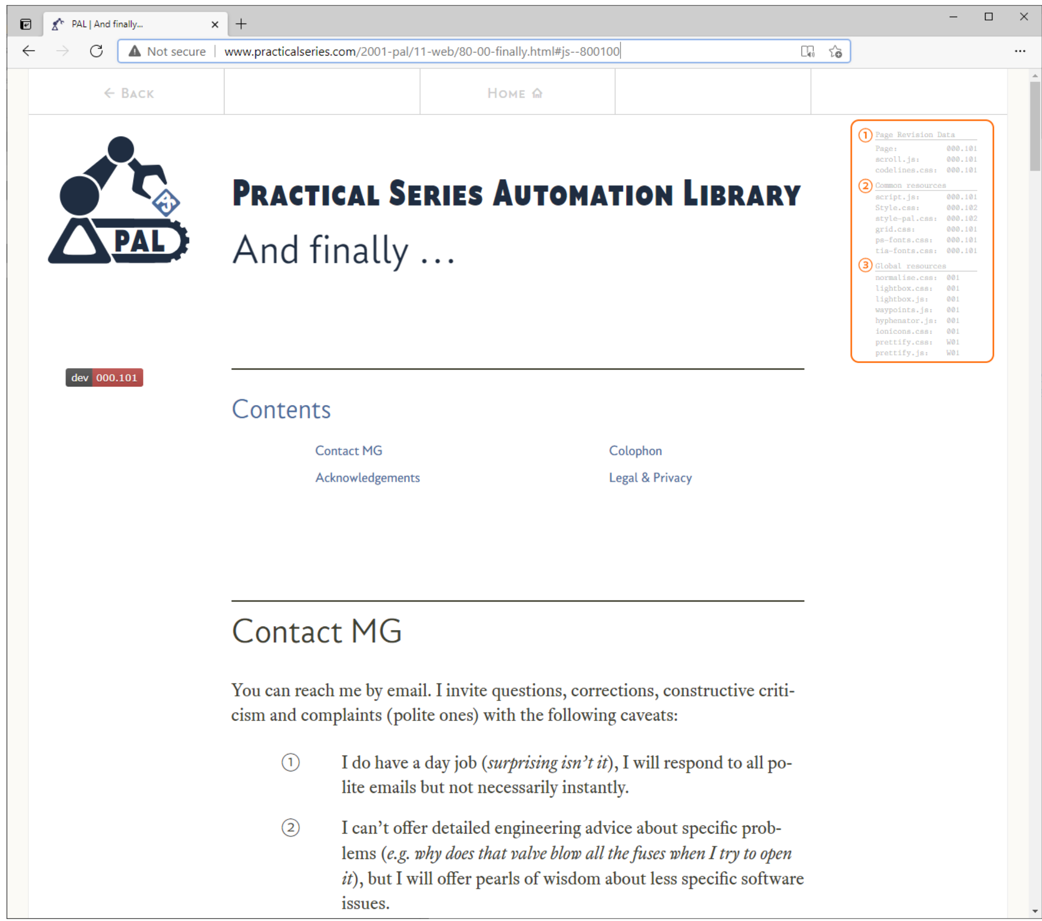


Figure 4.1 Web page component revisions

This can be seen in more detail below:

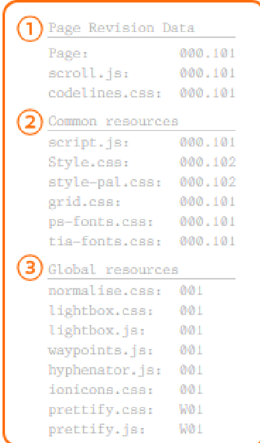


Figure 4.2 Web page component revisions (detail)

The Page revision data, point ①, shows the current revision of all the files associated with the particular web page.

The Common resources, point ②, shows the revision of the standard files that are common to all web pages.

Finally, the Global resources, point ③, shows the revision of all third-party files (these are normalised revision generated within the project, i.e. project revision `001` of `normailse.cs`s reflects the actual file build of `8.0.1`, the association between the normalised project build and the actual third-party build is listed in the revision table at the start of each file).

The numbering of the webpage (or file) revision `nnn.amm` is an incremental numbering system. In this system `nnn` reflects the current version of the software; typically, the first properly released software will be `001`. Previous development versions will be `000`.

The numbers after the decimal point (`amm`) reflect development and test modification to the current revision (for software modifications), in this system `a` reflects the current status of the software as follows:

FIRST DIGIT (`a`)	MEANING	DESCRIPTION
0	Released `mm` will be `00`	Page/file is released at version `nnn` (i.e. `nnn.000`)
1-7	development	Page/file is under development and has not been tested
8	Publication	Page/file has been published (on the webserver) for live testing

Table 4.2 Web page/file revision number (first digit)

The remaining numbers (`mm`), are incremental build numbers for the current revision (this allows development tracking).

Note: *A release version of a page or file will have revision `001.000`, `002.000`, `003.000` &c. I.e. the numbers after the decimal point are all zero. The first development of the software at release `003` would have revision `003.101`.*

The Global resources revisions, point ③, in Figure 4.2 only have the first three digits (`nnn`), third party software is always at a released version.

4.6.1 Recording revision numbers within web page files

All HTML, CSS, JavaScript and jQuery files have a modification history at the start of the file; an example for `style.css` is shown below:

```
00-comres > 11-resources > 01-css > # style.css > ...


/* PRACTICALSERIES (c) 2021

*****************************************************************************
Title:          PRACTICAL SERIES                                    STYLE.CSS
*****************************************************************************

PRACTICALSERIES: Practical Series of Publications by Michael Gledhill
                 Published in the United Kingdom

                 Email: mg@practicalseries.com
                 Web:   https://practicalseries.com

-----------------------------------------------------------------------------
DETAILS

This is the main style sheet for the website, it contains styles for the
following:

        Base settings
        Top navigation
        Fixed navigation
        Cover features (titles & picture)
        Header title bar
        Table of contents (TOC)
        Standard section formatting (titles, text styles, side bars etc.)
        Figures
        Tables
        Code fragments
        Formula
        Footer
        Footer navigation

The style sheet also contains responsive formatting for each of the areas
(responsive settings are at the end of each section).

This is one of a set of style sheets associated with the website, a full list
of style sheets and the order in which they should be called is given here:

        ps-fonts.css            - web fonts for the site
        tia-fonts.css           - Siemens TIA Portal web fonts for the site
        ionicons.min.css        - Social media font icons
        normalise.css           - standardises the rendering of the site on
                                  different browsers
        prettify.css            - styles associated with Google Prettify
        (called by script)        this is called automatically by the
                                  run_prettify.js file
        lightbox.css            - styles associated with lightbox2 viewer
        grid.css                - defines a responsive grid (Kerstner grid)
                                  for columns within the website
        style.css               - this file
        codelines.css           - Optional, adds additional line styling to
                                  Google Prettify code fragments

-----------------------------------------------------------------------------
*****************************************************************************
MODIFICATION HISTORY:

This is a complete summary of all software modifications.

Date          Issue       Author        Reason for Modification
-----------------------------------------------------------------------------
14 Feb 2021   000.102     M. Gledhill   SS02 stylistic set applied to
                                        .sans and .sans-b classes

10 Feb 2021   000.101     M. Gledhill   Development release for site build
                                        Revision control active

09 Feb 2021   000.001     M. Gledhill   .rev-badge-side added
                                        .title-contents added
                                        em changed to italic
                                        .emph bold emphasis added

08 Feb 2021   000.000     M. Gledhill   Established for PAL
                                        Based on R01 in PS1001
-------------------------------------------------------------------------- */
```

Figure 4.3 Modification history shown in file header

In addition, the revision information is also hardcoded into each file:

```
121  /* ***************************************************************************
122     REVISION
123     *************************************************************************** */
124  #rev-style:after { content: "000.102"; }
125
```

Figure 4.4 Hardcoded revision information (for `style.css`)

The various revisions for each file type are displayed by the HTML as multiple rows in a revision table, the following shows the format of each entry:

Web page .html file

```
<!--  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      TABLE - REVISION TABLE
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
                    <table class="table-rev">
<!-- Data row -->       <tr>
                            <td class="leading" style="height: 1em"></td>
                            <td class="table-cent" style="width: 30%"></td>
                            <td class="table-left" style="width: 40%">NAME:</td>
                            <td class="table-right" style="width: 30%" id="ID_NAME"></td>
                        </tr>
```

Code 4.1 Revision data displayed on a web page

The `NAME` and `ID_NAME` are as follows:

NAME	ID_NAME	DESCRIPTION
`Page`	`rev-doc`	Hardcoded revision of the web page
`scroll.js`	`rev-scroll`	`ID_NAME` is replaced by revision number in `NAME` file
`codelines.css`	`rev-codelines`	`ID_NAME` is replaced by revision number in `NAME` file
`script.js`	`rev-script`	`ID_NAME` is replaced by revision number in `NAME` file
`style.css`	`rev-style`	`ID_NAME` is replaced by revision number in `NAME` file
`style-pal.css`	`rev-style-pal`	`ID_NAME` is replaced by revision number in `NAME` file
`grid.css`	`rev-grid`	`ID_NAME` is replaced by revision number in `NAME` file
`ps-fonts.css`	`rev-ps-fonts`	`ID_NAME` is replaced by revision number in `NAME` file
`normalise.css`	`rev-normalise`	`ID_NAME` is replaced by revision number in `NAME` file
`lightbox.css`	`rev-lightbox`	`ID_NAME` is replaced by revision number in `NAME` file
`waypoints.css`	`rev-waypoints`	`ID_NAME` is replaced by revision number in `NAME` file
`hyphenator.css`	`rev-hyphenator`	`ID_NAME` is replaced by revision number in `NAME` file
`ionicons.css`	`rev-ionicons`	`ID_NAME` is replaced by revision number in `NAME` file
`prettify.css`	`rev-prettify`	`ID_NAME` is replaced by revision number in `NAME` file
`prettify.js`	`rev-run-prettify`	`ID_NAME` is replaced by revision number in `NAME` file

Each file listed in the `NAME` column appends its own revision number to the `ID_NAME` ID, in the case of `style.css`, the code is:

style.css

```
/* ****************************************************************************
   REVISION
   **************************************************************************** */
#rev-style:after { content: "000.102"; }
```

Code 4.2 Revision information for `style.css`

In the case of a `.js` file, it has the following appearance (this is for `script.js`)

script.js

```
/* ****************************************************************************
   REVISION
   **************************************************************************** */

    $('#rev-script').append (
        "<p>000.101</p>" /* LOCAL JS REVISION NUMBER */
    );
```

Code 4.3 Revision information for `script.js`

The actual HTML file has the data hardcoded in the HTML table after the `rev-doc` ID (`000.101` in this case):

Web page .html file

```
<!-- Data row -->   <tr>
                        <td class="leading" style="height: 1em"></td>
                        <td class="table-cent" style="width: 30%"></td>
                        <td class="table-left" style="width: 40%">Page:</td>
                        <td class="table-right" style="width: 30%" id="rev-doc">000.101</td>
                    </tr>
```

Code 4.4 Revision data for the HTML file

BLANK PAGE

5 Software storage and folder structures

There are several aspects to the PAL storage locations: there are the software files (the modules that form the PAL itself), there are the Git and GitHub repositories that hold those modules within the version control system. There are the Project directories that hold all the project documentation, there is a website that makes the Project documentation and PAL software available to those to whom it is of interest and finally, there is the backup storage locations for all of it.

Broadly, the software and folder structures cover the following:

① Project software storage on an Engineering Station

② Project document storage and the Project directories

③ Website folder structure and Web Development Platform

④ Local (Visual Studio Code) machine repositories

⑤ Remote (GitHub) repositories

⑥ Cloud based storage (Dropbox)

⑦ Network storage and backup facilities

This section covers each of these areas in detail.

5.1 An overview of the Project structure

Figure 5.1 shows the entire Project structure and all its components.

Physically, there are three main components:

① Engineering Stations (ES) for the development of the PAL software

② Web Development Platform (WDP) for the configuration of the web sites that accompany the Project

③ Network accessible storage (NAS) that holds all the Project documentation, drawings, schedules &c.

In addition there are remote, cloud-based storage facilities that hold the version control repositories associated with the project.

There is also *"off-site"* mirror storage of the NAS and all files and folders therein.

In terms of physical machines, there can be any number of Engineering Stations (ESs), each will have a copy of the PAL software repository. Each engineer working on the PAL software development will have a fully equipped Engineering Station

There can also be any number of Web Development Platforms (WDP) each with a copy of the website repository. In practice there will be a limited number of such machines.

The Project documentation is stored on the NAS drive and is accessible to all Project personnel. There is no special requirement for machines that can access the Project documentation (any standard office machine is suitable).

In addition to the remote repositories, cloud-based synchronisation is carried out between the NAS drive and all master engineering stations and all master web development platforms, this provides up to date repository backups on the NAS drive as well in the remote repositories.

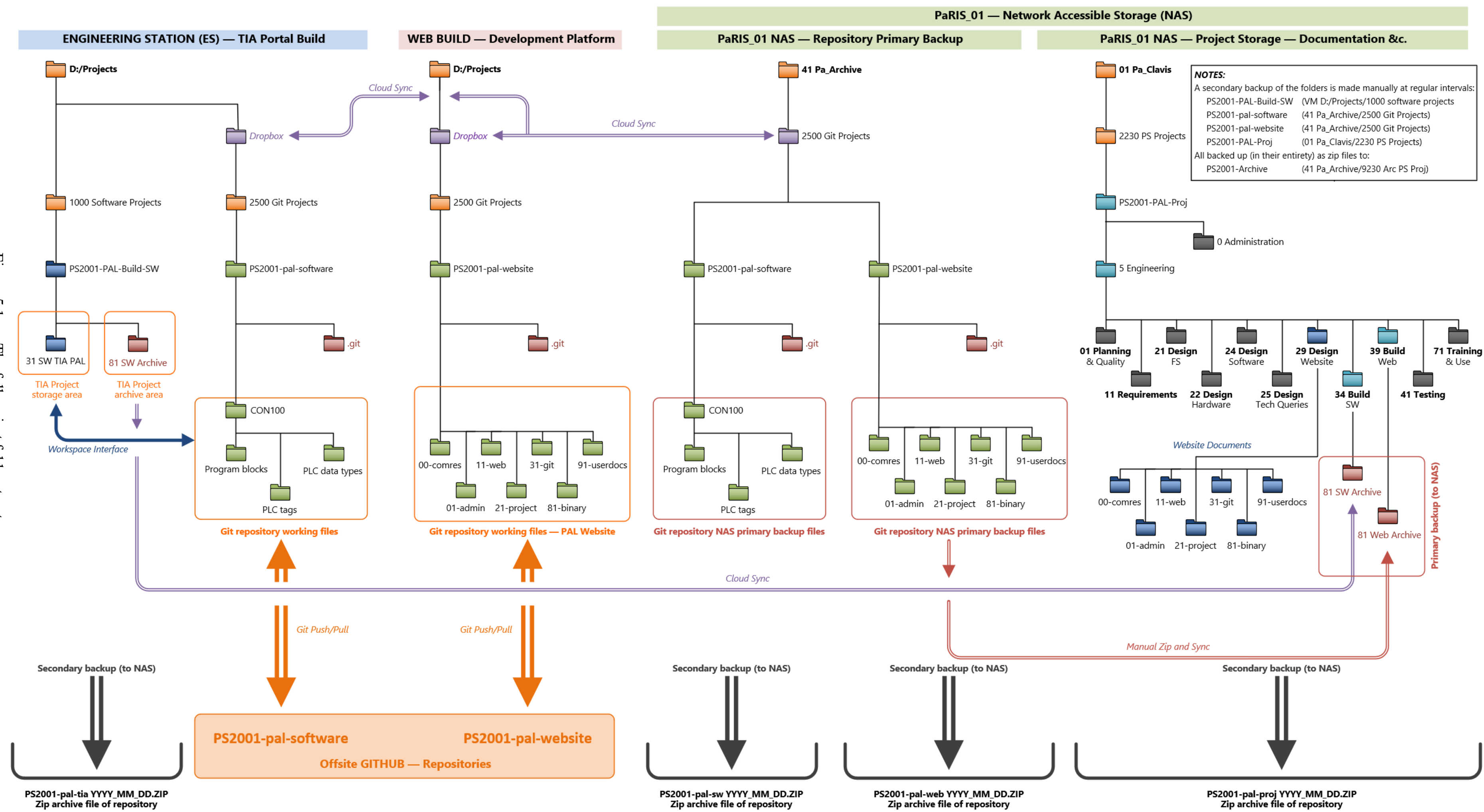


Figure 5.1 The full project folder structure

5.2 Engineering stations

Engineering stations (ES) are used to develop the PAL software, they are generally high-powered machines with at least 512 GB of hard drive storage and 16 GB of RAM. Typically with a 10th or 11th generation i7 processor or equivalent (such as an AMD Ryzen 7).

Engineering stations should be equipped with dual 27" QHD (quad high definition) screens (these have a resolution of 2560 × 1440 pixels).

The configuration of an ES, including drive allocation, device naming, software package installation &c. is explained in the ES/WDP Configuration Manual *[Ref. 006]*.

In summary, it is assumed that the ES has been configured in line with the above document and is equipped as follows:

The ES is equipped with three hard drive partitions (as a minimum):

`C:`	`OpSys`	Operating system and application files
`D:`	`Projects`	PSP project files
`E:`	`Licences`	Storage area for licences &c.

The `C:` drive (`OpSys`) holds the operating system and any installed programmes and applications (the Siemens application software and any office applications will be installed on this drive). The `C:` drive should be at least 200 GB in size.

The `D:` drive (`Projects`) holds any Controller, HMI and SCADA projects developed using TIA Portal. This consists of the source code, archives, graphical images, runtime configurations and any other files needed to develop the control system software. The `D:` drive should fill the remainder of the hard drive, excepting 1 GB that should be reserved for the `E:` drive. The `D:` drive should be at least 200 GB in size.

The `E`: drive (`Licences`) holds all the licences needed to activate the Siemens TIA Portal software and its installed options.

The `E:` drive is generally very small, it need only be a few megabytes in size (in practice, a 1 GB partition is more than adequate).

The software applications and configuration below are required on an ES:

1. TIA Portal has been installed
   - The TIA Portal settings have been set to the PAL configuration
   - The TIA Portal Git add-in has been installed and enabled
2. A GitHub user account has been setup
   - The account has been added as a contributor to the PracticalSeries organisation
3. Git SCM has been installed
   - Notepad++ is installed as the Git default editor
   - An SSH key link has been established between Git and GitHub
4. The Visual Studio Code text editor has been installed
   - The standard set of Visual Studio Code extensions have been installed

The packages above are listed in the order in which they should have been installed on the engineering station. The exact details for installing and configuring the above application is given in the ES/WDP Configuration Manual *[Ref. 006]*.

5.2.1 ES software folders

All the Controller software for the PAL is stored on the `D:` drive (`Projects`).

The `D:` drive has two primary folders:

`1000 Software Projects`	TIA Portal Projects folder
`2500 Git Projects`	Git Workspace folder

The underlying structure is:

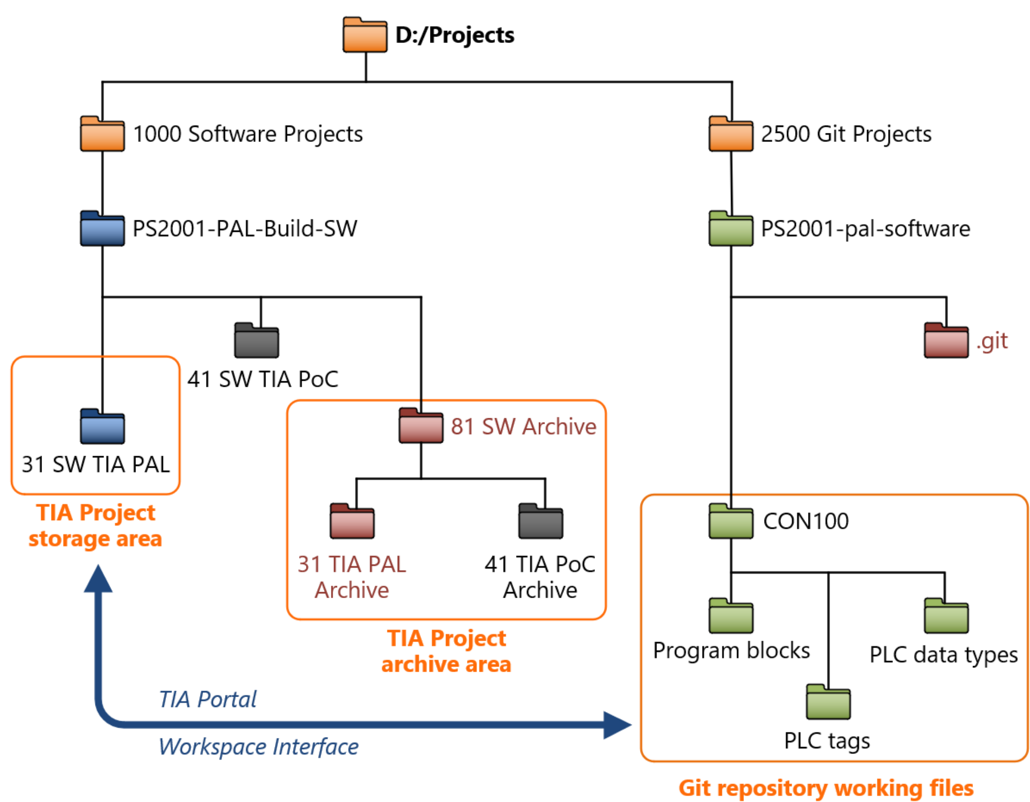


Figure 5.2 ES Project folder structure

The software projects folder:

```
1000 Software Projects
```

is used to store the individual Controller projects that are opened and developed using TIA Portal. All such projects (within the PAL) are named as follows (see § 3.8):

```
PS2001-PAL-<commit tag>
```

The alternate branch on the `D:` drive is:

```
2500 Git Projects
```

This is used to store the Git repository that is used to store the TIA Portal Project Workspace.

These two folders:

`1000 Software Projects`	TIA Portal Projects folder
`2500 Git Projects`	Git Workspace folder

are examined further in the following sections.

5.2.2 Software development area (`1000 Software Projects`)

There may be multiple projects under the

`1000 Software Projects`

Each project is identified by its project number (`PSnnnn`) followed by the name and some brief description of the project. In the case of the PAL software the project number is (`PS2001`) and the full project folder is (`PS2001-PAL-Build-SW`).

The `PS2001-PAL-Build-SW` folder holds all the TIA projects, these are the projects that can be downloaded via TIA Portal into a Controller.

The entire software development takes place in TIA Portal and is stored as a TIA project locally on the engineering station.

All TIA Projects are stored in the folder `31 SW TIA PAL`:

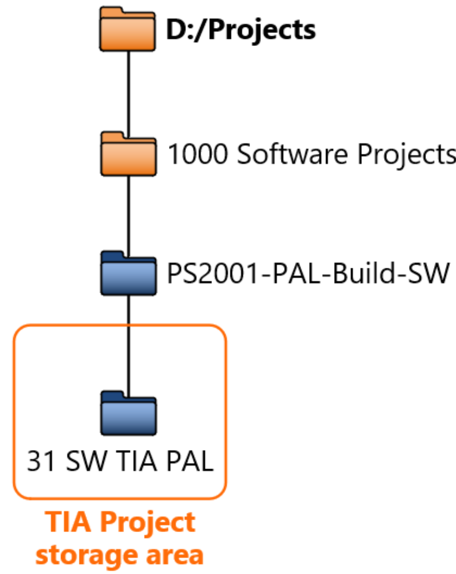


Figure 5.3 PAL local ES TIA Project folder structure

Each TIA Portal project is stored in its own sub-folder under the `31 SW TIA PAL` directory. Each project is named according to its commit point (see § 3.8 for information about file names).

An example is shown below:

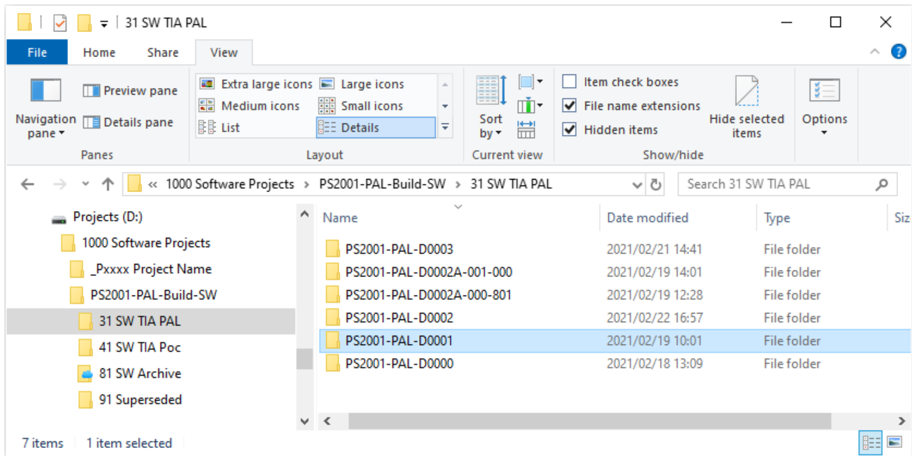


Figure 5.4 TIA Projects within the folder structure

The interior of a TIA Project folder has the following appearance:

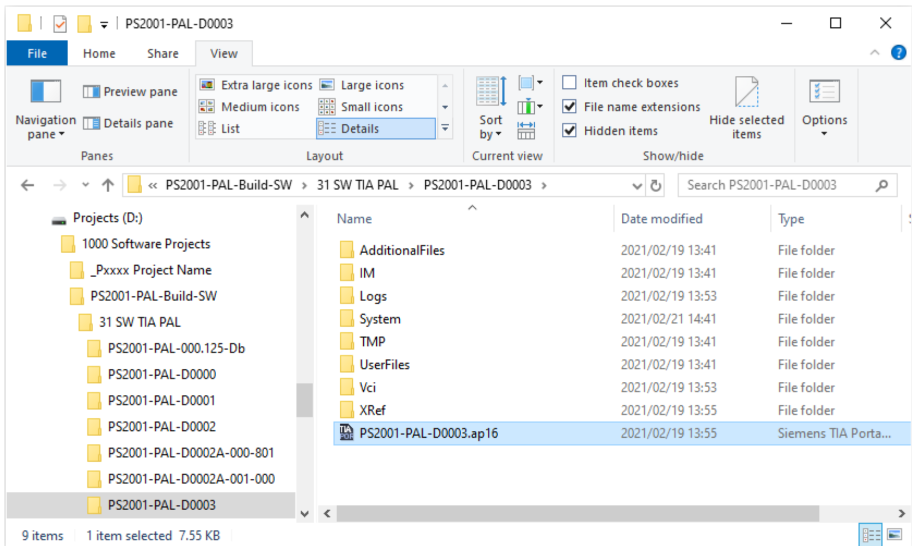


Figure 5.5 TIA Projects within the folder structure

The `.ap16` file (highlighted in blue), if clicked would open the project in TIA Portal. The folder structure above is all part of the TIA Project and independent modification of this structure or any files within it may result in the corruption of the project.

The top-level folder of a project (in this case `PS2001-PAL-D0003`) should be considered to be the TIA Portal Project in its entirety, everything below this level is best left alone.

Figure 5.2 shows an additional folder under the `PS2001-PAL-Build-SW` directory, this is the `41 TIA PoC` folder. This also holds TIA Projects; these are not formally part of the PAL software, they form *"proof of concept"* projects.

Proof of concept software is used to develop, test or demonstrate that a particular approach works within the PAL prior to that approach being formally adopted within the PAL.

Proof of concept projects can be considered a test bed or prototyping area for software.

The final folder under the `PS2001-PAL-Build-SW` directory is designated `81 SW Archive`; this is used to store *"archived"* copies of each build of the TIA Project software.

Archived copies of a project are produced by TIA Portal, they are essentially *zipped* versions of the TIA Project folder with non-essential (or re-buildable) information removed

Archive files are a convenient way of transporting projects (and indeed, each archived copy of the software is available for download from the website, see below).

Project archive files all have the extension `.zap16` (and are universally referred to as *"zap"* files), they are indeed zip files, if the extension were change from `.zap16` to `.zip`, the contents could be extracted by Windows Explorer.

There is an archive file for all commit points (see § 3.8) within the software (both primary and secondary). These are accessible from the website at the following address:

https://practicalseries.com/2001-pal/31-git/81-00-archive.html

5.2.3 The Workspace and local repository (`2500 Git Projects`)

The ES stores all its Git repositories in the directory:

`2500 Git Projects`

Specifically, the PAL repository associated with the Controller software development is the `PS2001-pal-software` folder:

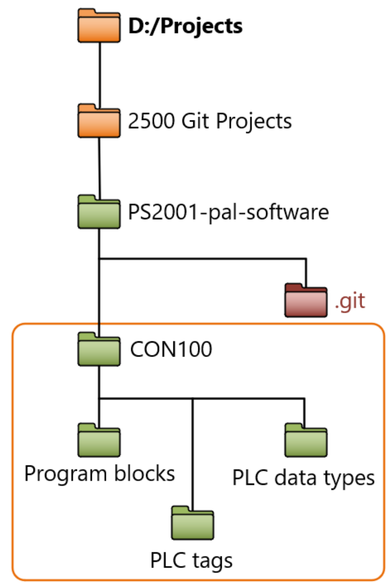


Figure 5.6 PAL local ES repository folder structure

This folder is a Git repository, this can be seen by the hidden `.git` folder, this contains all the underlying repository structures required by Git and GitHub to control and manage the folder.

The `.git` folder is similar to the TIA Project folders, in that, no changes should ever be made directly to anything that is in there. The best thing to do is to never open it or look in it; just leave it alone, it looks after itself.

The remainder of the `PS2001-pal-software` folder holds the working files for the TIA Portal Workspace.

The top-level folder (`CON100`) is the Workspace equivalent of the controller in the TIA Project, this is also called `CON100` (this is in accordance with the Siemens naming conventions discussed in the Software Design Specification *[Ref. 003]*, § 3.1.4)

The Workspace folder contains the XML versions of the TIA Project exportable objects, these objects are also stored in (pre-named) folders:

`Program blocks`	Holds the XML versions of Controller blocks: FBs, FCs, OBs and DBs
`PLC tags`	Holds any tag tables configured for the Controller
`PLC data types`	Holds the User Data Type (UDT) structures

The linkage between the controller and the Workspace folder can be seen below:

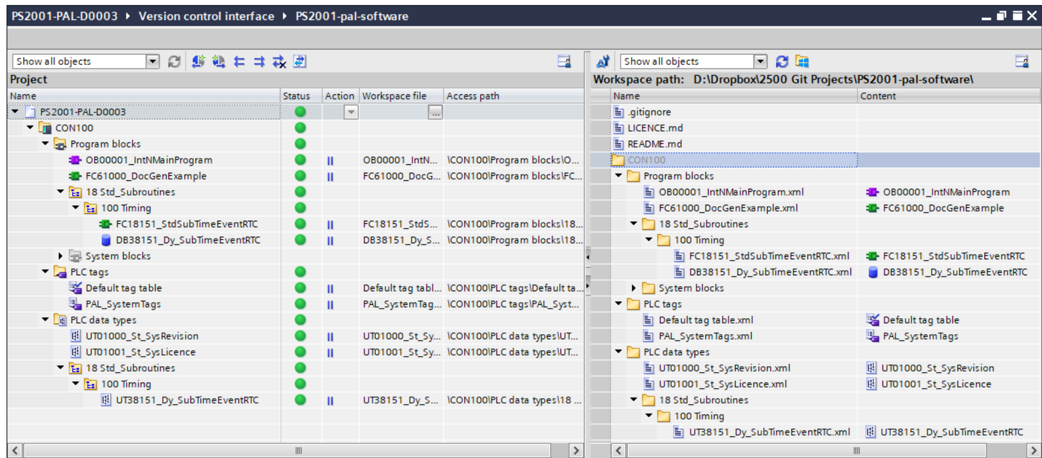


Figure 5.7 Project (left) and Workspace (right) associations

The Workspace only holds the programmable objects from the Controller software (there is no hardware configuration, watch tables or traces &c.). It can be seen that each folder and object in the TIA Project (left-hand side) has an equivalent in the Workspace (right-hand side).

It can also be seen that the Workspace also holds some additional files: `.gitinore`, `LICENCE.md` and `README.md`; these are all files associated with the Git repository itself and have no associations within the TIA Project.

These association are explained further in the following section:

5.2.4 Understanding the Simatic Workspace

A Workspace is just a Windows folder located somewhere on the ES hard drive. It can be any folder and can have any name.

The folder can be created directly through Windows or when being define via TIA Portal.

In TIA Portal terms, a Workspace is simply a folder to which it will export copies of all the following types of objects (if they exist in the Project):

① Code blocks:

- Organisation blocks (OBs)
- Functions (FCs)
- Function blocks (FBs)
- Data blocks (DBs) of any type including instance DBs

② User (PLC) data types (referred to here, as UDTs)

③ PLC tag tables (referred to here as just tag tables)

TIA Portal exports them as text files, specifically as XML files.

In addition to this, TIA Portal keeps track of the files it has exported and identifies if differences exist between its internal Project files and those files in the Workspace. If differences do exist, TIA Portal is able to synchronise those files in either direction (it can make the Workspace files match the Project files or, it can make the Project files match the modified Workspace files).

The ES/WDP Configuration Manual *[Ref. 006]* contains a full description of how to create and link a Workspace to a TIA Portal Project.

Author's note — How we got here

This isn't a new concept for Siemens, although it is new to TIA Portal. The forerunner to TIA Portal was a PLC programming package called Simatic Manager (or, more commonly, Step 7), this was used to programme earlier ranges of PLCs called S7-300 and S7-400 (TIA Portal will also programme these PLCs).

Simatic Manager had the facility to export (or import) programmable blocks in a readable format, referred to as Source Blocks, all blocks could be converted to Source Blocks and the resulting Source Blocks were all text files that held a version of the software written in a Pascal like language (actually called Structured Control Language or SCL).

The Source Files were again very useful, they were text files and so could be stored easily and could also be incorporated in a version control system — they were also readable (by humans).

When TIA Portal was introduced and Simatic Manager began to be phased out (you can still get it, but most people use TIA Portal now), the Source Block functions (or any such equivalent) was not included in TIA Portal, and this upset a lot of people — virtually everyone to whom version control was important.

Siemens were reminded of their deficiencies by those people, *"oh tut deary me"* they said, *"you seem to have forgotten this"* or its Anglo-Saxon equivalent.

Siemens have now addressed their shortcomings and have added the required features; the format is different: it exports thing as XML text files (rather than SCL text files), but it can be used in much the same way. I.e. version control systems can read the files and determine any changes that have been made. Siemens refer to the whole thing as part of their *"openness"* strategy. So, it's arrived late, but at least it's here now.

Looking once more at the Workspace in TIA Portal:

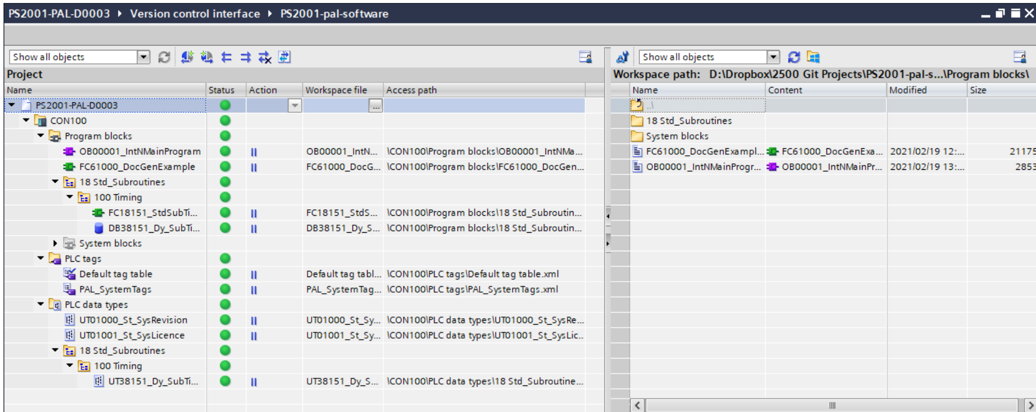


Figure 5.8 Workspace viewed in TIA Porta

The `CON100` folder (on the right) has been expanded out to show all the objects within it. On the right, the Program block folder has been opened showing the two blocks present within it. These are the XML equivalents of the blocks beginning OB00001 and FC61000 on the left.

Opening one of the XML files (in this case OB00001), gives something similar to:

OB00001_IntNMainProgram.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<Document>
  <Engineering version="V16" />
  <DocumentInfo>
    <Created>2021-01-08T13:31:28.4860482Z</Created>
    <ExportSetting>WithDefaults</ExportSetting>
    <InstalledProducts>
      <Product>
        <DisplayName>Totally Integrated Automation Portal</DisplayName>
        <DisplayVersion>V16 Update 3</DisplayVersion>
      </Product>
      <OptionPackage>
        <DisplayName>TIA Portal Version Control Interface</DisplayName>
        <DisplayVersion>V16 Update 2</DisplayVersion>
      </OptionPackage>
      <OptionPackage>
        <DisplayName>TIA Portal Openness</DisplayName>
        <DisplayVersion>V16 Update 3</DisplayVersion>
      </OptionPackage>
      <Product>
        <DisplayName>STEP 7 Professional</DisplayName>
        <DisplayVersion>V16 Update 3</DisplayVersion>
      </Product>
      <OptionPackage>
        <DisplayName>STEP 7 Safety</DisplayName>
        <DisplayVersion>V16</DisplayVersion>
      </OptionPackage>
      <Product>
        <DisplayName>WinCC Professional</DisplayName>
        <DisplayVersion>V16 Update 3</DisplayVersion>
      </Product>
    </InstalledProducts>
  </DocumentInfo>
  <SW.Blocks.OB ID="0">
    <AttributeList>
      <AutoNumber>true</AutoNumber>
      <HeaderAuthor>M_Gledhill</HeaderAuthor>
      <HeaderFamily>Interrupt</HeaderFamily>
      <HeaderName>IntNMainProgram</HeaderName>
      <HeaderVersion>0.1</HeaderVersion>
      <Interface><Sections xmlns="http://www.siemens.com/automation/Openness/SW/Interface/v4">
  <Section Name="Input">
    <Member Name="Initial_Call" Datatype="Bool" Accessibility="Public" Informative="true">
      <Comment>
        <MultiLanguageText Lang="en-US">Initial call of this OB</MultiLanguageText>
      </Comment>
    </Member>
    <Member Name="Remanence" Datatype="Bool" Accessibility="Public" Informative="true">
      <Comment>
        <MultiLanguageText Lang="en-US">=True, if remanent data are available</MultiLanguageText>
      </Comment>
    </Member>
  </Section>
  <Section Name="Temp">
    <Member Name="licInfo" Datatype="&quot;UT01001_St_SysLicence&quot;">
      <Comment>
        <MultiLanguageText Lang="en-US">Licence information for this block</MultiLanguageText>
      </Comment>
      <Sections>
        <Section Name="None">
          <Member Name="LICENCE_TYPE" Datatype="String[20]" />
          <Member Name="COPYRIGHT" Datatype="String[30]" />
        </Section>
      </Sections>
    </Member>
  </Section>
  <Section Name="Constant" />
```

Figure 5.9 The XML file for OB 1

The XML files, although not instantly comprehensible, can be read by the human eye and ultimately understood.

Understanding the Workspace symbols

Looking once more at Figure 5.8, in the left pane, there are two columns, STATUS and ACTION. The STATUS tells us the state of the object in the TIA Project compared with the state of the object in the Workspace folder. The green dot means the two version are identical (generally, this is the preferred state).

The STATUS can have the following values:

SYMBOL	MEANING	DESCRIPTION
	No differences	The compared versions of the object in the project and the Workspace are identical. If at a group level, all lower-level elements are identical in the project and in the Workspace.
	Lower-level differences	One or more lower-level elements are different in the project and the Workspace, open the group to see the affected files.
	Not in workspace	The object is only available in the project
	Project object modified	The compared versions of the object in the project and the Workspace are different. The TIA Portal object has been changed since the last synchronisation operation (Project is newer)
	Workspace object modified	The compared versions of the object in the project and the Workspace are different. The Workspace file has been changed since the last synchronisation operation (Workspace is newer)
	Both modified	The compared versions of the object in the project and the Workspace are different. Both the TIA Portal object and the Workspace file have been changed since the last synchronisation operation
	Not known	The comparison result is not known

Table 5.1 Status symbols and meaning

The Action allows us to do something with the files, the Action field can have the following values:

SYMBOL	MEANING	DESCRIPTION
Blank	Not applicable	Not applicable to this object (usually folders of groups, the folders or group must be expanded to see individual objects)
ǁ	No action	No action will be taken (do nothing)
→	Export to Workspace	The object will be exported to the Workspace (Workspace object will be made identical to Project object)
←	Import to Project	The object will be imported from the Workspace (Project object will be made identical to Workspace object)

Table 5.2 Action symbols and meaning

Synchronising the Workspace

If changes have been made to blocks within TIA Portal, these changes will be indicated within the workspace, for the sake of argument, let's assume that OB00001 and UT38151 have been changed, the Workspace will now have the following appearance:

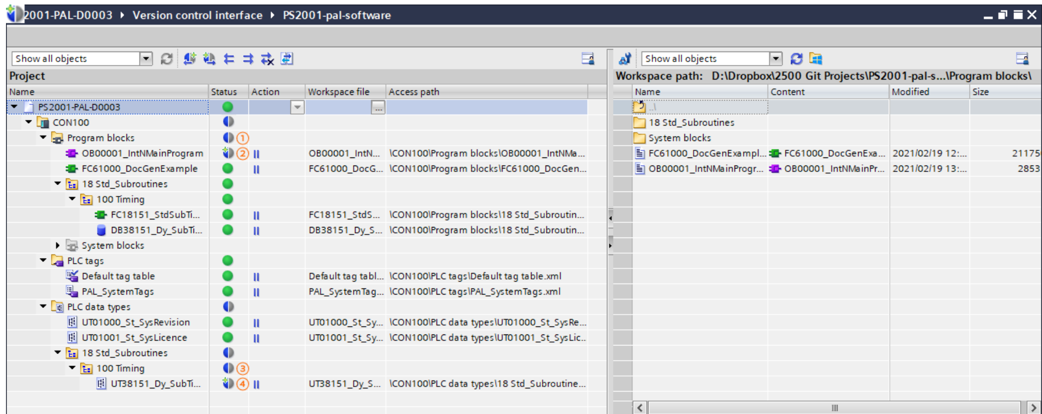


Figure 5.10 Synchronising the Workspace

Point ① is indicating that a difference exists at a lower level within the Program blocks folder. Point ② shows that OB1 has been changed in TIA Portal (c.f. Table 5.1); the Project version is newer than the Workspace version (the star is on the left in the symbol). Points ③ and ④ show similar changes for the UT38151 data type.

By using the drop down in the ACTION box next to each modified block, it is possible to select an action individually for each block.

Alternatively, the menu bar at the top of the left window:

allows the action for all the modified blocks to be selected in one go, clicking the icon (Export changes to Workspace) changes the ACTION for the two blocks to →. Nothing has happened at this point; the desired action has simple been selected *(but not yet implemented)*.

To make the changes, the Synchronize button must be clicked. This carries out the selected actions on the modified blocks.

Note: *Only complied blocks can be synchronised with the Workspace, if the blocks have not been compiled within TIA Portal, the user will be prompted to compile them as part of the synchronisation process.*

Generally, it is better to compile the blocks first and separately to the synchronisation process, this allows any errors to be more easily addressed.

Knowhow and write protection

Simatic blocks can be protected in two ways, knowhow protect (an older form of access protection) and write protection (the current form of access protection).

Blocks with knowhow protection cannot be synchronised with the Workspace.

However, blocks with Write Protection can be synchronised; some blocks within the PAL require protection (see the Validation Plan *[Ref. 001]* Appendices for an explanation), where this is used, the protection will be Write Protection rather than knowhow protection.

Common actions

In the previous example, the icon (Export changes to Workspace) was used to select the required action, the other symbols are as follows:

SYMBOL	MEANING	DESCRIPTION
	Import changes to Project	Import changes from the Workspace to the Project (Workspace has the newer version)
	Export changes to Workspace	Export changes from the Project to the Workspace (Project has newer version)
	Import all	Where an object differs in the Project and Workspace, overwrite the object in the Project with the one from the Workspace (even if the Project holds the newer version)
	Export all	Where an object differs in the Project and Workspace, overwrite the object in the Workspace with the one from the Project (even if the Workspace holds the newer version)
	Discard all actions	Discard any actions that may have been applied (set everything back to "do nothing")
	Synchronise	Implements the selected action

Table 5.3 Action toolbar commands

Note: *Actions can only be applied to an object when there is a difference between the object in the Project and the object in the Workspace*

The difference between IMPORT/EXPORT CHANGES and IMPORT/EXPORT ALL is subtle. If a block has been changed in the Project (left-hand side) then of the IMPORT/EXPORT CHANGES buttons only the EXPORT CHANGES will work, the IMPORT CHANGES will leave the action set at do nothing. The reason for this is that there has been no change to the Workspace object, so you cannot import it.

There has, however, been a change to the Project object so it can be exported as a change.

Conversely, the IMPORT ALL button *will* work, this allows the modified object in the Project to be overwritten by the unmodified (and older) object in the Workspace. The IMPORT/EXPORT ALL buttons will always work on any two objects that are different.

A note about new objects in the Project

There is a problem *(Author: I'm not sure if this is a problem or if it is intentional, it is however peculiar)* when creating a new block in TIA Portal (or indeed, creating a new block in the Workspace); if a new block were created in the Project (`Block_1` for example, shown below):

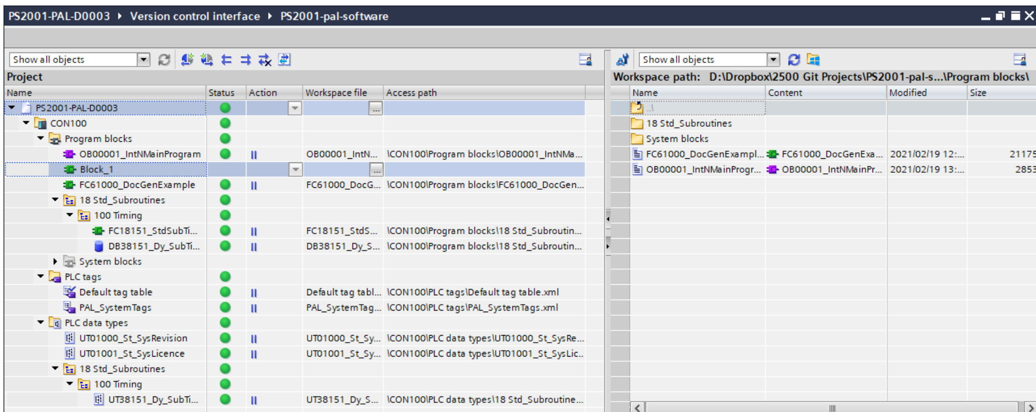


Figure 5.11 A new block in the Project

The new block (`Block_1`) is present in the Project Window, but the STATUS and ACTION columns are empty, pressing any of the action buttons or trying to assign an action directly will not work, it won't do anything.

The reason for this is that the object has not been *"linked"* to the Workspace. The way to link the object is to drag it from the Project side to the Workspace side (some care is needed; the block must be copied to the correct folder in the Workspace).

Once this is done, the `Block_1` STATUS will display the green dot, indicating the two files are synchronised and are the same.

(Author: It would seem better to me if the new block were treated as a modified block and it automatically gave the Project object modified status and it could just be synchronised, rather than having to actually drag it to a specific folder in the Workspace.)

5.2.5 Understanding the Workspace as a local repository

Section 5.2.3 established that the Workspace was also a local Git repository; this means that when files are synchronised between the Project and the Workspace, any changes are logged by the Git VCS. At some point, these changes will be *committed* to the repository and permanently stored there. At the same time an archive copy of the TIA Project will also be made (see § 5.2.6).

The Workspace is configured using the Siemens Git add-in, this is installed and activated using TIA Portal (again the instructions for doing this are given in the ES/WDP Configuration Manual *[Ref. 006])*. The Git add-in has very limited functionality compared with the Visual Studio Code functionality and is not actually used directly to maintain the Workspace repository. It is installed purely, so that TIA Portal recognises the Workspace is a repository and hides the `.git` folder, preventing it from being visible and from being inadvertently modified.

The Workspace is maintained from within TIA Portal, all changes made to the software are synchronised with the Workspace in the manner described in § 5.2.4.

The repository aspect of the Workspace is maintained via the Visual Studio Code arrangement (again the instructions for installing Visual Studio Code are given in the ES/WDP Configuration Manual *[Ref. 006])*. The Visual Studio Code text editor provides a graphical user interface to the Git repository allowing development branches to be created and used with the Workspace.

All repository actions are carried out using the Visual Studio Code application. This includes committing changes to the local repository, creating and managing development branches and the bidirectional synchronisation of the local repository with the remote GitHub repository.

5.2.6 Commit point archives

Whenever a commit is made to the repository, the TIA Project is saved and archived at that point, the archived version of the software is given the name:

`PS2001-PAL-<commit tag>`

The commit tag being the identifying tag given to the commit point (see § 3.8)

The archived copy of the software is stored in the Project folder

`D:/1000 Software Projects/PS2001-PAL-Build-SW/81 SW Archive/31 TIA PAL Archive`

All commit points (both primary and secondary, see §§ 3.2 and 3.4) are stored as archive (`.zap16`) files in this directory.

5.2.7 Maser ES — local repository backup to NAS

There can be any number of engineering stations (ESs), generally, each software development engineer will have one.

There is however, only one *Master Engineering Station* (MES), this is usually the engineering station given to the lead software engineer on the Project.

All ESs have a local repository (the Workspace) and are constantly being synchronised with the remote repository on the GitHub servers, whenever a commit is made to a local repository, on any ES, that ES must first be synchronised with the remote repository (ensuring that any changes made by the local commit do not create a conflict with the any other changes that have been stored within the remote repository).

The remote repository is essentially, the **master** repository and it is this repository that holds all the commits made by any ES.

The GitHub servers are a third party facility (ultimately owned by Microsoft), and while they are considered secure by the Practical Series of Publications, it is felt that GitHub cannot be the sole storage location for the **master** repository (Microsoft may, for example, close down the site, make it prohibitively expensive, or indeed may sell it to some other party that the Practical Series of Publications does not trust).

To this end, the Master Engineering Station, each time it is synchronised with the **master** repository on GitHub, makes a complete copy of the repository on the PSP network accessible storage (NAS) drives.

The MES should be synchronised at least once a week with the remote repository.

This is done purely as an additional backup, the PSP does not think that the GitHub website will change dramatically in the future, or be sold to the Russians — however, there is the old engineering maxim: *"better to not need a backup you have, rather than need a backup that you do not have"*, or to put it another way *"better safe than sorry"*.

Repository backup mechanism

The Master ES repository backup mechanism is slightly convoluted. This is because the Git repositories, and in particular the `.git` folder with the repository are managed (ultimately) by the Git application and this is by-and-large, a Unix based application; and while this is not a problem and the application runs perfectly well on a Windows machine, some of the filenames it uses have a Unix feel to them.

An example being the `.git` folder itself. To some extent, Windows, and certainly some Windows application do not like files that start with a full stop, they expect there to be something before the full stop and everything after it is the extension (`.pdf` or `.jpeg` for example).

This is a problem with the PSP NAS drives, these NAS drives are all supplied by Synolgy (this is the PSP standard for NAS drives).

Synology NAS drives are all equipped with an application called *Cloud Station Drive* and this allows any folder on any PC to by synchronised in real time with any corresponding folder on the NAS drive itself. The link, once created, automatically keeps the folders in sync whilever the machine is connected to the internet. It is exceptionally easy to use and just works.

The problem with this arrangement is that certain files are not synchronised (temporary files for example or files with particular extensions) and this is the problem with the `.git` folder, *Cloud Station Drive* ignores certain files that Git considers essential and this leads to a corrupted copy of the repository on the NAS drive.

To overcome this problem, a slightly different approach is taken. On the Master ES, the `2500 Git Project` folder is, itself, stored within a live Dropbox folder:

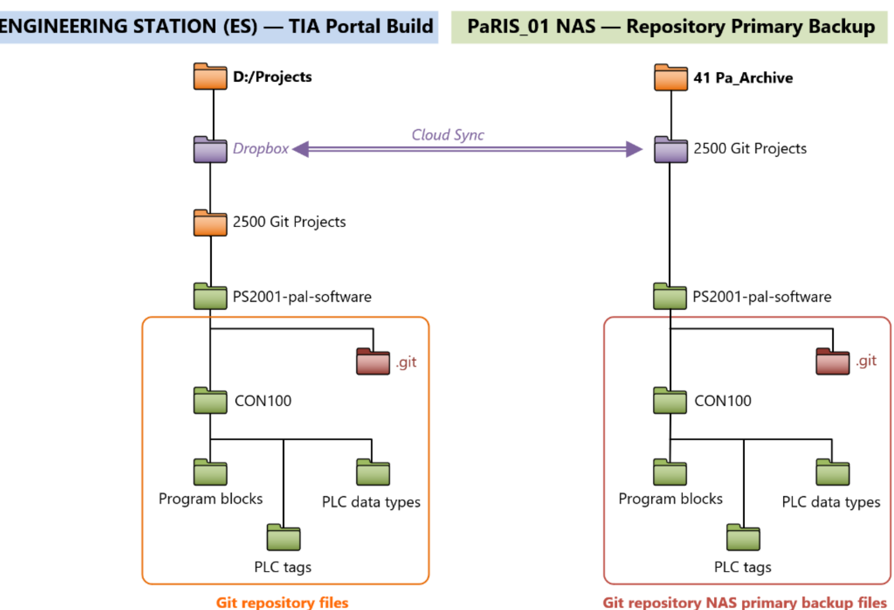


Figure 5.12 Backing up a repository to the NAS drive

The Dropbox account (in this instance) is the PSP Dropbox account, this synchronises the entire `2500 Git Projects` folder and all its content with the Dropbox cloud servers.

Synology NAS drives are equipped with a *Cloud Sync* package that allows a folder within a Dropbox account to synchronise with a partner folder on the NAS drive. In this case it is setup as follows:

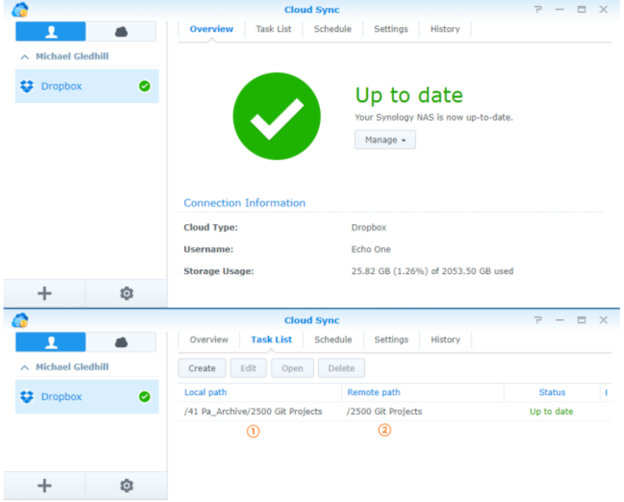


Figure 5.13 Backing up a repository to the NAS drive

The Local Path, point ①, is a directory on the Synology NAS drive, the Remote Path, point ② is the Dropbox folder. It is possible to synchronise individual folders within the `2500 Git Projects` folder if required.

This process does not suffer from the restrictions of the Cloud Station Drive application and will correctly synchronise all files within the repository without exception.

Application to access to the PSP Dropbox account should be made to:

ACCOUNT MANAGER:	Michael Gledhill
ACCOUNT DETAILS:	PSP Dropbox
CONTACT DETAILS:	mg@practicalseries.com

Table 5.4 PSP Dropbox account manager details

5.2.8 Remote repository

All ESs work with a remote repository that contains the current copy of all committed changes made on any of the ES machines. The remote repository is the **master** repository, it holds all the development branches (created by any ES) and the most up to date **master** branch.

Any development work that takes place on a development branch on an ES, will at some point be committed to the local repository (on the ES), before this can happen, the Visual Studio Code application making the commit will require that any changes that exist within the remote repository, but are not present on the local ES (i.e. changes that have been made by other users on other ESs) are *pulled* from the remote repository, before the local ES changes can be *pushed* back to the remote repository. This Pull before Push approach ensures that the user must resolve any conflicts between the user's local repository on the local ES and the remote repository before pushing the resolved changes back to the remote.

There is an explanation of this process (and indeed the whole, Git and GitHub approach to version control) on the PracticalSeries website at the following address:

https://practicalseries.com/1002-vcs/08-00-remotes.html

To use the remote repository from a local ES, the two must be linked via a secure shell key link (SSH link), the process for doing this is explained in the ES/WDP Configuration Manual *[Ref. 006]*, and again, on the website here:

https://www.practicalseries.com/1002-vcs/04-00-linking.html

To make this link, the user of the ES must have their own GitHub account and this account must be given contributor access to the remote PSP repository.

The remote repository is public repository (one that anyone with a GitHub account can read and copy) and is part of the GitHub PracticalSeries organisation. The organisation is available here:

https://github.com/practicalseries

The remote repository itself is available here:

https://github.com/practicalseries/PS2001-pal-software

Read access to the organisation and all of the repositories it contains, is available to anyone with a GitHub account.

Access for contributors requires permission from the organisation owner, applications for such access should be made to:

GITHUB ORGANISATION:	https://github.com/practicalseries
REPOSITORY NAME:	PS2001-pal-software
ORGANISATION OWNER:	Michael Gledhill
CONTACT DETAILS:	mg@practicalseries.com

Table 5.5 PracticalSeries GitHub organisation details

At the time of writing, the remote repository was in a preliminary state and had the following appearance:

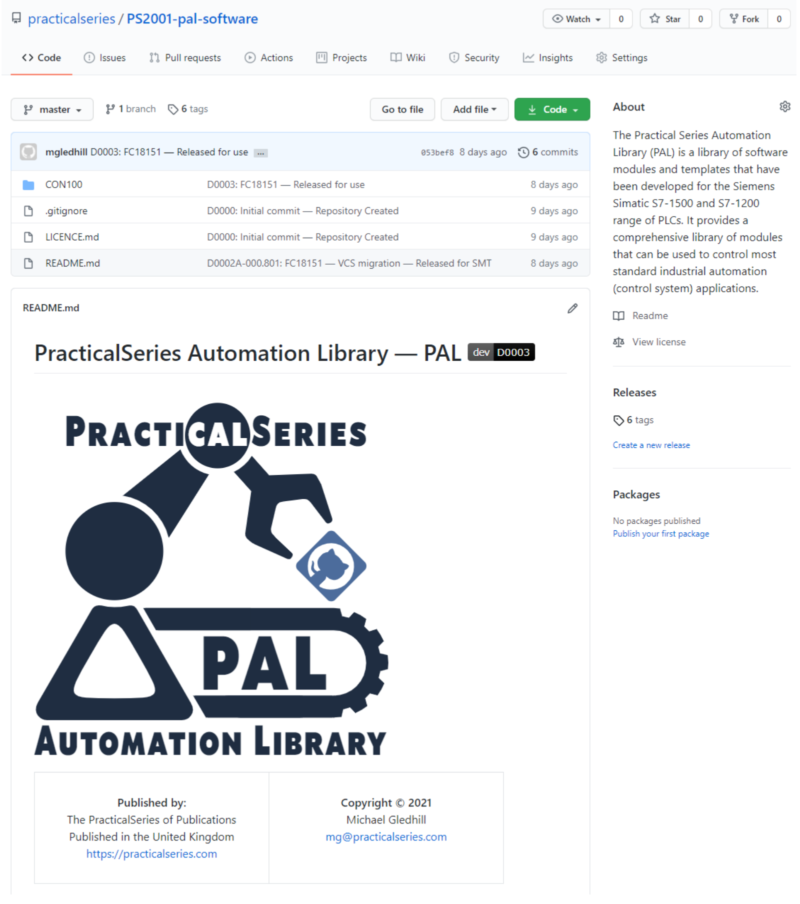


Figure 5.14 The remote PAL repository

5.3 Web development platforms

Web development platforms (WDP) are used to develop the website that supports the PAL software Project.

Web development platforms are similar to engineering stations and have a similar specification: generally high-powered machines with at least 512 GB of hard drive storage and 16 GB of RAM. Typically with a 10th or 11th generation i7 processor or equivalent (such as an AMD Ryzen 7).

WDPs should be equipped with dual 27" QHD (quad high definition) screens (these have a resolution of 2560 × 1440 pixels).

Unlike ESs, WDP machines do not need a fixed IP address and need not have TIA Portal installed.

Note: *It is perfectly possible to use an Engineering Station as a Web Development Platform and indeed, it is quite common to do so (in which case it will be referred to as an ES).*

The configuration of a WDP, including drive allocation, device naming, software package installation &c. is explained in the ES/WDP Configuration Manual *[Ref. 006]*.

In summary, it is assumed that the WDP has been configured in line with the above document. In short it is equipped as follows:

The WDP, like an ES is equipped with three hard drive partitions (as a minimum):

`C:`	`OpSys`	Operating system and application files
`D:`	`Projects`	PSP web project files
`E:`	`Licences`	Storage area for licences &c.

The `C:` drive (`OpSys`) holds the operating system and any installed programmes and applications. The `C:` drive should be at least 200 GB in size.

The `D:` drive (`Projects`) holds the website project developed for the PAL. Broadly, this is all the HTML, CSS, JS, jQuery and image files needed by a website.

The `E`: drive (`Licences`) holds is generally not used on a WDP machine, but is included to give a consistent approach to configuring both WDPs and ESs.

The `E:` drive is generally very small, it need only be a few megabytes in size (in practice, a 1 GB is more than adequate).

The software applications and configurations required by a WDP are as follows:

1. A GitHub user account has been setup
   - The account has been added to the PracticalSeries organisation
2. Git SCM has been installed
   - Notepad++ is installed as the Git default editor
   - An SSH key link has been established between Git and GitHub
3. The Visual Studio Code text editor has been installed
   - The standard set of Visual Studio Code extensions have been installed

The packages above are listed in the order in which they should have been installed on the WDP. The exact details for installing and configuring the above application is given in the ES/WDP Configuration Manual *[Ref. 006]*.

5.3.1 WDP software folders

The WDP website project is stored on the `D:` drive (`Projects`)

The `D:` drive a holds single primary folder that contains the WDP website Git repository:

`2500 Git Projects`	Git Workspace folder

The underlying structure is:

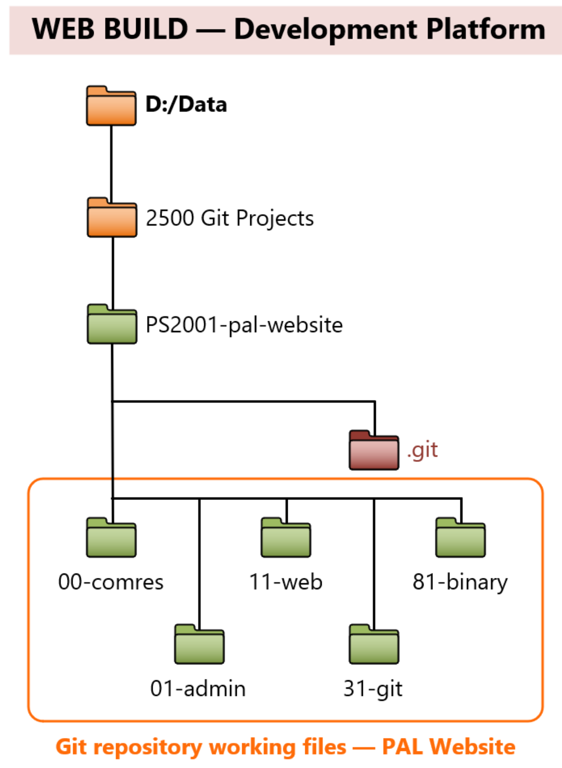


Figure 5.15 The website folder structure

This structure is examined further in the following section:

5.3.2 Understanding the website structure

The website structure of Figure 5.15 (everything below the `PS2001-pal-website`, excepting the `.git` folder) is the actual website, the offline version. Everything in these folders is copied to the live website server and can be seen at the following address:

https://practicalseries.com/2001-pal/

The relationship between the offline and online folders is as follows:

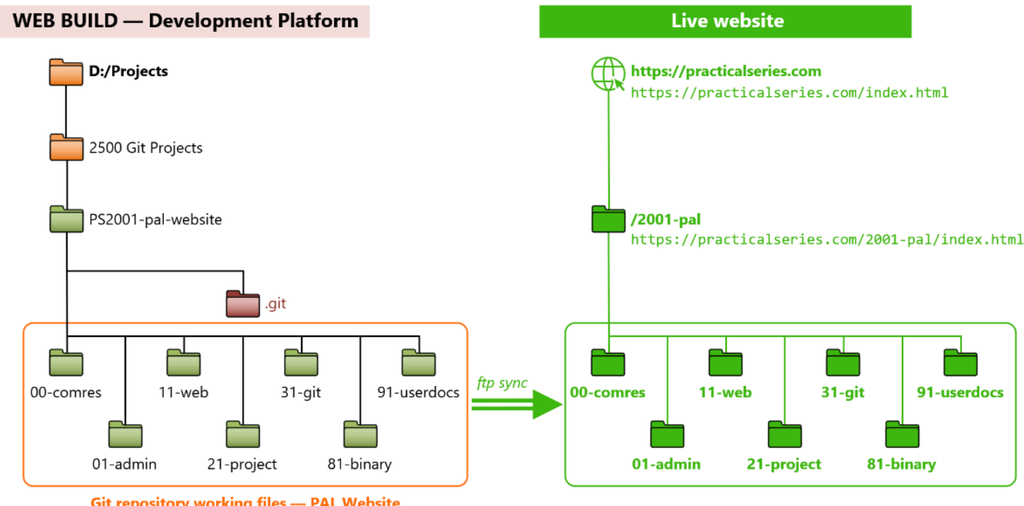


Figure 5.16 The website folder in relation to the live website

The contents of the folders (`00-comres`, `01-admin`, `11-web`, `21-protect`, `31-git`, `81-binary` and `91-userdocs`) and the root folder are copied to the live website servers using a *file transfer protocol* (FTP) package, this is a one-way sync from the WDP to the website servers. The synchronisation is made manually whenever the WDP website is updated.

The website has several folders within it:

`00-comres`	Common resources
`01-admin`	Various administration pages
`11-web`	The main website containing the PAL user guides and information
`21-project`	Holds all the documentation associated with the Project (validation documents)
`31-git`	Contains information used by the GitHub repositories
`81-binary`	Contains binary files (the TIA Project archive files &c.)
`91-userdocs`	The online version of the User Documentation files embedded in the TIA Project

The contents of these folders are shown in the Figure 5.17 and Figure 5.18 below:

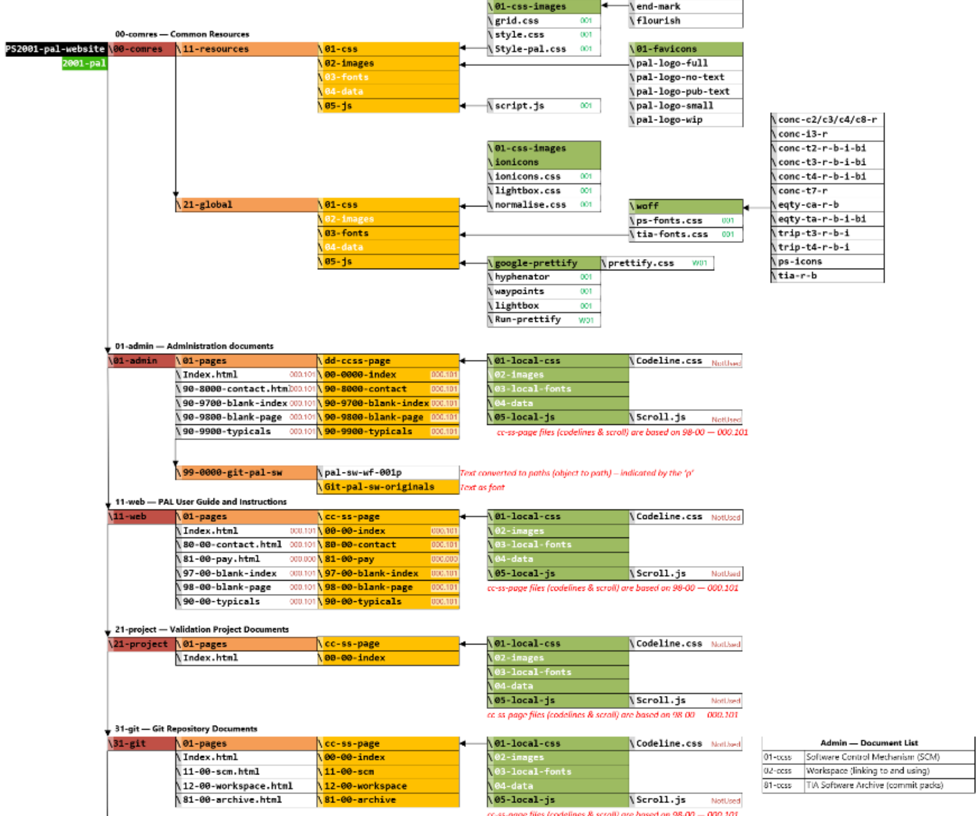


Figure 5.17 The website folder structure in detail (part 1)

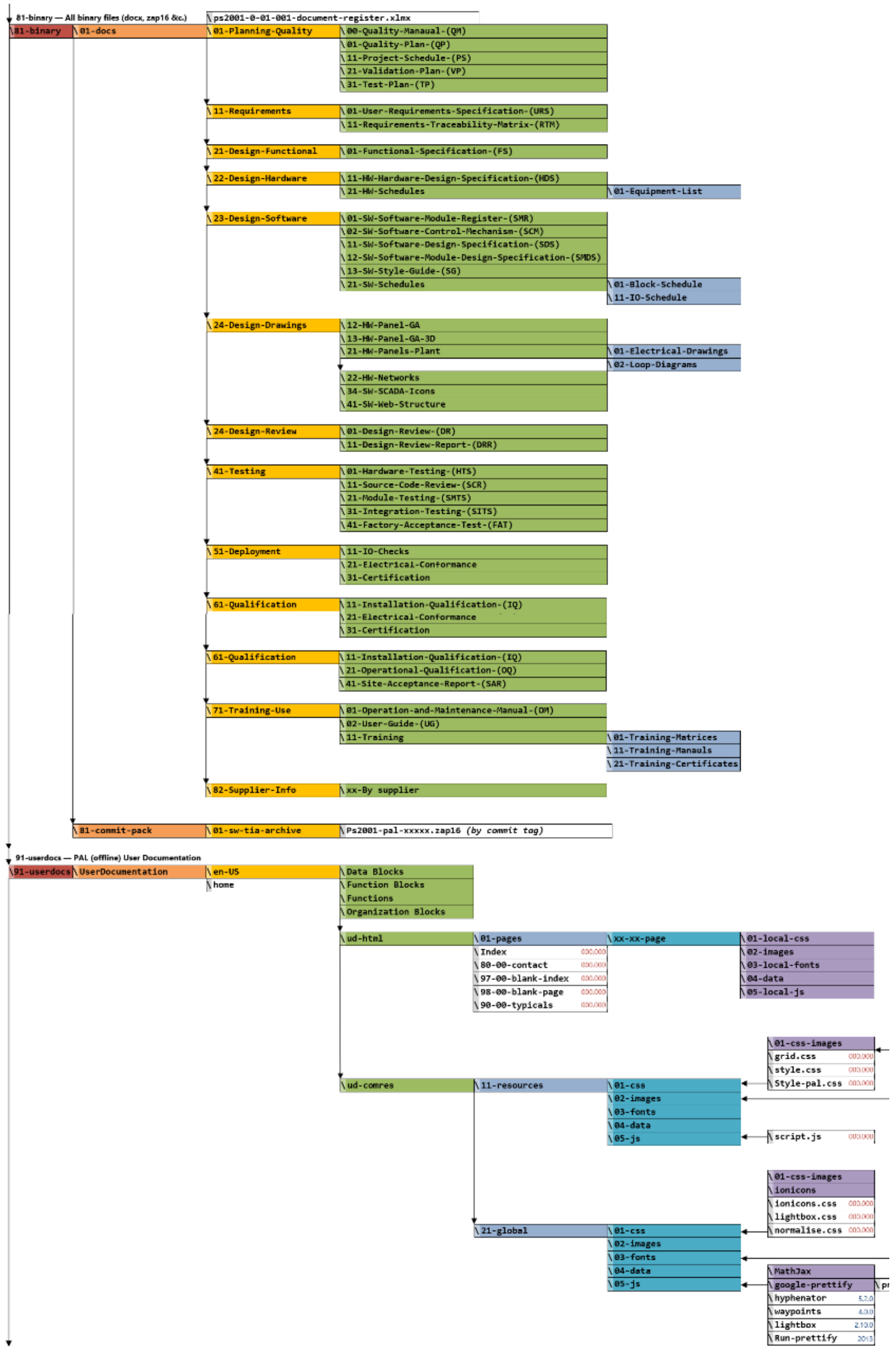


Figure 5.18 The website folder structure in detail (part 2)

The website has two primary components:

① A user guide explaining how to download and use the PAL software (contained in the `11-web` directory)

② A comprehensive guide to validated projects and all the documentation associated with the PAL project in its entirety (contained in the `21-project` directory)

The remaining folders are secondary and are used to support the primary sections. A brief description of all the folders is given below:

The PAL documentation (`11-web` directory)

This contains a full, on-line description and operating guide for the PAL software. It explains exactly how to use the PAL software, how to configure it and gives very detailed, real-world examples of how to use it.

This directory also contains individual module documentation (in an online format) that explains exactly how each module is configured, the requirements and options for that module and examples of how to use the module.

This part of the website is directly accessed at the following url:

http://practicalseries.com/2001-pal/11-web/

The Project documentation (`21-project` directory)

The PAL software is designed for use within pharmaceutical environments and as such is a *"validated"* project (see the Validation Plan *[Ref. 001]* for a detailed description of validated projects and their requirements).

Validation is the process of making sure a computerised system (such as a PLC and its software) does precisely what it was designed to do; specifically, it is the exercise of correctly and traceably documenting every requirement of the system and making sure that that requirement is formally and exhaustively tested.

The fact that the Project is validated, and the associated documents required by such projects have been deemed to be useful in their own right. This part of the website gives

a practical approach to validating a control system, it explains the "life cycle" process and the phases necessary to progressing from a requirement specification to a fully validated and deployed system.

This part of the website provides examples of all the documentation required to by a validated system and explains how they should be used. This documentation is all made available in pdf and Microsoft Office formats (Word, Excel, Visio and Projects); the documents are complete and can be downloaded and reused as a template by anyone to whom they may be useful; again under the MIT licence (see page 2)

The project documentation also includes copies of the completed test and qualification documents needed to demonstrate the PAL software has been validated

This part of the website is directly accessed at the following url:

http://practicalseries.com/2001-pal/21-project/

Common resources (`00-comres` directory)

The common resources are those components needed by every page within the website. It contains things such as the common cascading style sheets (CSS), the JavaScript (JS) files used within each page, common images and the common font-files needed to correctly render the web pages.

The `00-comres` directory is broadly split into two further directories:

`11-resources`	Contains CSS, JS and jQuery files written and produced by the PSP
`21-global`	Contains third party components needed by the website

The `11-resources` folder contains files associated with the website that have been written and developed by the PSP engineers (i.e. these are files that belong to the PSP). They fall into three categories: CSS files (to manage the appearance of the webpages), images (such as logos &c.) and JavaScript files that handle the dynamic navigation used on the web pages.

The `21-global` is primarily used to hold third party applications that are used within the website. These are categorised as follows:

`woff files`	These contain the fonts used by the website and were purchased by the PSP
`normalise.css`	A third-party file use to standardise how different browsers render a website
`lightbox.js`	Used to display images in a larger, overlay arrangement
`Waypoints.js`	Used to create dynamic navigation bars
`Hyphenator.js`	Used to correctly and dynamically hyphenate the website text
`MathJax.js`	Used to render equations on the website where required
`Google-prettify.js`	Allows sections of software (code fragments) to be displayed on the website

Administration files (`01-admin` directory)

The administration area is used internally by the PSP web development team, it contains various files that are necessary for the website management (such as revision data, workflow diagrams, change requests &c.)

The `01-admin` directory is not directly accessible by users of the website, but the contents of it can be accessed by other webpages within the website to display or reference particular information.

Git repository webpages (`31-git` directory)

The Git repositories created as part of this Project are all public repositories available to anyone with a GitHub account. These repositories all contain documentation of some form or another, usually as `README.md` files, that explain the purpose of the repository and how to use the repository.

These files often reference specific websites or pages that offer further explanation of a particular point.

The `31-git` directory provides a storage location for such webpages for the PAL repositories; this document, for example, is available as an online webpage:

https://practicalseries.com/2001-pal/31-git/11-00-scm.html

Binary file storage (`81-binary` directory)

All the downloadable aspects of the website:

- PDF documents
- Microsoft Office documents
- Software archive files
- Code examples &c.

are stored in the binary area of the website, such files are all accessed via other webpages within the website.

User Document storage (`91-userdocs` directory)

The `91-userdocs` directory is a special directory and is structured in the correct format for the TIA Portal User Documentation facilities (see the Software Design Specification *[Ref.* 003*]*, section 13 for details of the User Documentation facilities).

This is the online version of the User Documentation embedded within the PAL software TIA Projects.

The User Documentation allows additional information about a block within the PAL software to be directly accessed from within the TIA Portal environment.

5.3.3 Local repository

The website folder: `PS2001-pal-website` contains the full website in the folders listed in the previous section. This folder is also a Git repository (separate to the `PS2001-pal-software` repository that contains the software being developed for the PAL, the Controller software, see § 5.2.3)

This means that the development of the website is under the control of the Git VCS.

The website is written and developed using the Visual Studio Code text editor. The repository aspect of the website is maintained via the source code control aspects of this application (again the instructions for installing Visual Studio Code and the various Git extension are given in the ES/WDP Configuration Manual *[Ref. 006]).*

All repository actions are carried out using the Visual Studio Code application. This includes committing changes to the local repository, creating and managing development branches and the bidirectional synchronisation of the local repository with the remote GitHub repository.

5.3.4 Master WDP — local repository backup to NAS

There can be any number of web development platforms (WDPs), generally, each developer will have one.

There is however, only one Master web development platform (MWDP), this usually belongs to the lead web developer.

All WDPs have a local repository and are constantly being synchronised with the remote repository on the GitHub servers, whenever a commit is made to a local repository, on any WDP, that WDP must first be synchronised with the remote repository (ensuring that any changes made by the local commit do not create a conflict with the any other changes that have been stored within the remote repository).

The remote repository is essentially, the **master** repository and it is this repository that holds all the commits made by any WDP.

For the same reasons given in § 5.2.7, the web repository is also backed up to the PSP NAS drive.

This is done by the Master Web Development Platform; each time the MWDP is synchronised with the **master** repository on GitHub, it makes a complete copy of the repository on the PSP network accessible storage (NAS) drives.

The MWDP should be synchronised at least once a week with the remote repository.

The backup mechanism is the same as that for the Master Engineering Station, it uses Dropbox as an intermediary, the full description of how this works is give on page 97.

The Master Web Development Platform has a slightly different folder structure. Similar to the Master ES, the `2500 Git Project` folder is, stored within a live Dropbox folder on the MWDP:

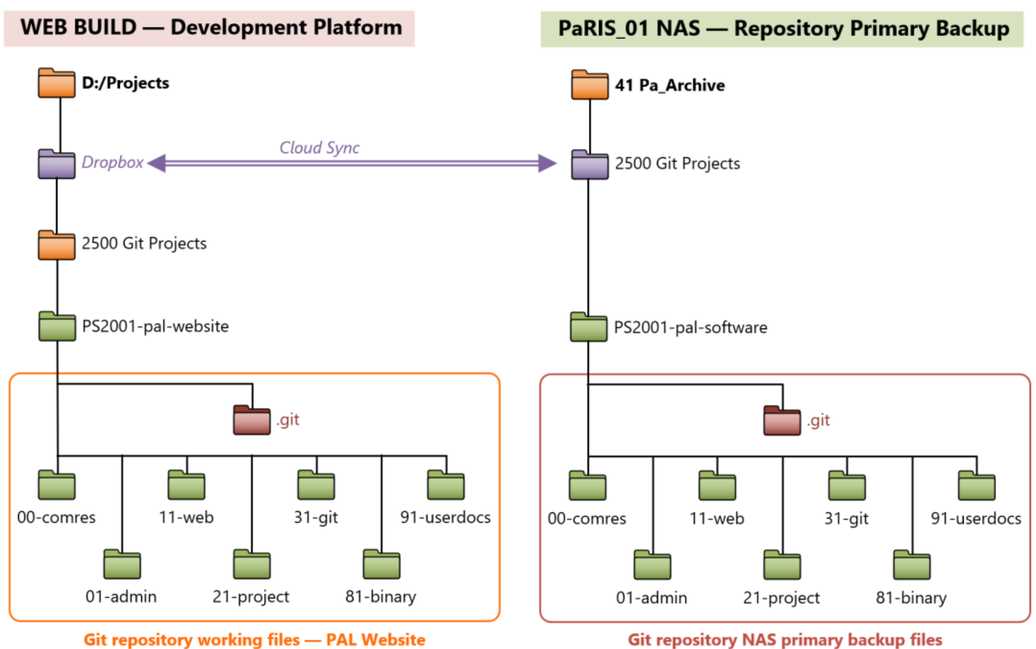


Figure 5.19 MWDP NAS backup structure

Application to access to the PSP Dropbox account should be made to:

ACCOUNT MANAGER:	Michael Gledhill
ACCOUNT DETAILS:	PSP Dropbox
CONTACT DETAILS:	mg@practicalseries.com

Table 5.6 PSP Dropbox account manager details

5.3.5 Remote repository

All WDPs work with a remote repository that contains the current copy of all committed changes made to the website on any of the WDP machines. The remote repository is the **master** repository, it holds all the development branches (created by any WDP) and the most up to date **master** branch.

Any development work that takes place on a development branch on any WDP, will at some point be committed to the local repository (on the WDP), before this can happen, the Visual Studio Code application making the commit will require that any changes that exist within the remote repository, but are not present on the local WDP (i.e. changes that have been made by other users) are *pulled* from the remote repository, before the local WDP changes can be *pushed* back to the remote repository. This Pull before Push approach ensures that the user must resolve any conflicts between user's local repository on the local WDP and the remote repository before pushing the resolved changes back to the remote.

To use the remote repository from a local WDP, the two must be linked via a secure shell key link (SSH link), the process for doing this is explained in the ES/WDP Configuration Manual *[Ref. 006].*

The remote repository is a public repository (one that anyone with a GitHub account can read and copy) and is part of the GitHub PracticalSeries organisation. It is available here:

https://github.com/practicalseries

The remote repository itself is available here:

https://github.com/practicalseries/PS2001-pal-website

Read access to the organisation and all of the repositories it contains, is available to anyone with a GitHub account.

Access for contributors requires permission from the organisation owner, applications for such access should be made to:

GITHUB ORGANISATION:	https://github.com/practicalseries
REPOSITORY NAME:	PS2001-pal-website
ORGANISATION OWNER:	Michael Gledhill
CONTACT DETAILS:	mg@practicalseries.com

Table 5.7 PracticalSeries GitHub organisation details

5.3.6 The live website

The live Practical Series of Publications website is hosted by Heart Internet in the United Kingdom.

The website has various publications (of which the PAL website is just one component), The landing page for the top level of the website is:

https://www.practicalseries.com/

And the landing page for the PAL website is:

http://www.practicalseries.com/2001-pal/

The Master Web Development Platform (MWDP) is used to maintain the live website.

The live website is an exact copy of the offline website stored in folder `PS2001-pal-wesite` on the `D:` drive of the MWDP, but without the `.git` folder.

The website is uploaded from the MWDP to the Heart Internet servers using the WinSCP application, the installation of this application is discussed in the ES/WDP Configuration Manual *[Ref. 006]*.

Logon information is required to give access to the WinSCP application (this logon information is also restricted to having the correct credentials, the website will only permit machines with specific IP addresses to upload the data).

The WinSCP application has two windows, the left-hand side is the *offline* website on the MWDP, the right-hand side is the *online* website on the Heart Internet servers:

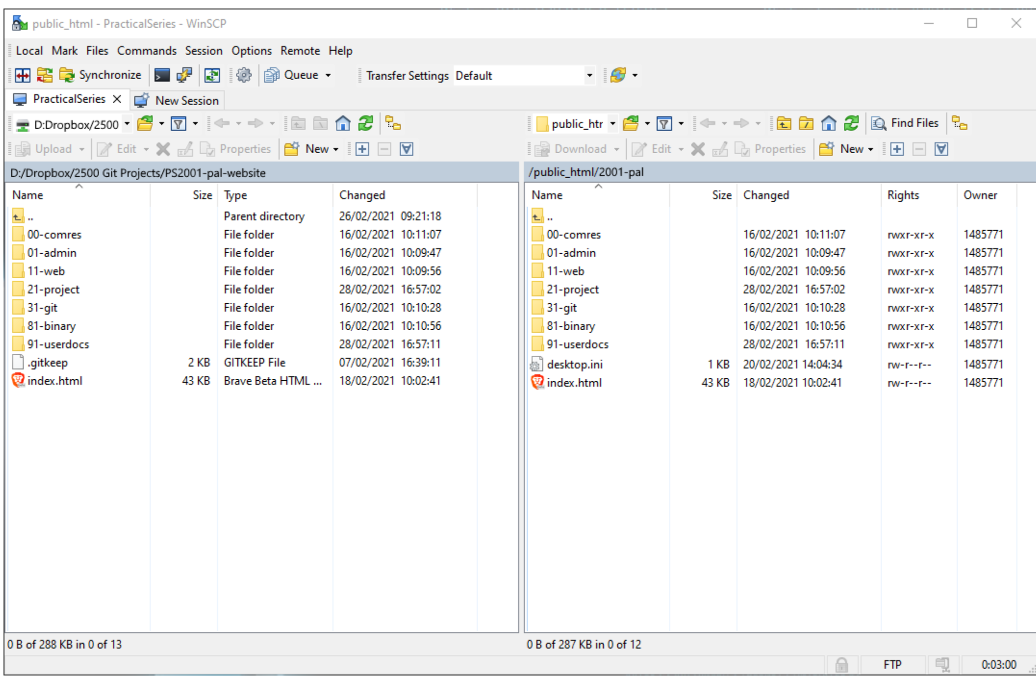


Figure 5.20 WinSCP FTP with the live website

Those requiring FTP access to the website should apply to the following:

SITE OWNER:	Michael Gledhill
ACCOUNT DETAILS:	PSP Website FTP
CONTACT DETAILS:	mg@practicalseries.com

Table 5.8 PracticalSeries FTP access details

5.4 NAS based Project documentation

The PAL Project documentation and administration files are stored on the PSP NAS drives, in the common project area. This can be accessed as a network location on any office PC as follows:

```
\\192.168.1.85\01 Pa_Clavis\2230 PS Projects\PS2001-PAL-Proj
```

Within this folder, the Project has six distinct areas:

`0-Administration`	Contains the project register (of all documents) and a set of template documents for use within the Project
`3-Project management`	Contains all the project management files: resource management, project planning, order placement, security &c.
`4-QHSE`	Quality, health, safety and environment. Contains all risk assessment and method statements and handles any health and safety incidents
`5-Engineering`	Contains the bulk of the project documentation, organised according to life cycle phases. Holds all documents, spreadsheets, drawings &c. required to design and build the Project
`6-Accounting`	Cost tracking, budget management and invoicing.
`7-Correspondence`	All project correspondence including minutes of meetings, scanned copies of paper correspondence and a full email archive

Table 5.9 Main areas within the Project directory

The full Project folder structure is shown in Figure 5.21:

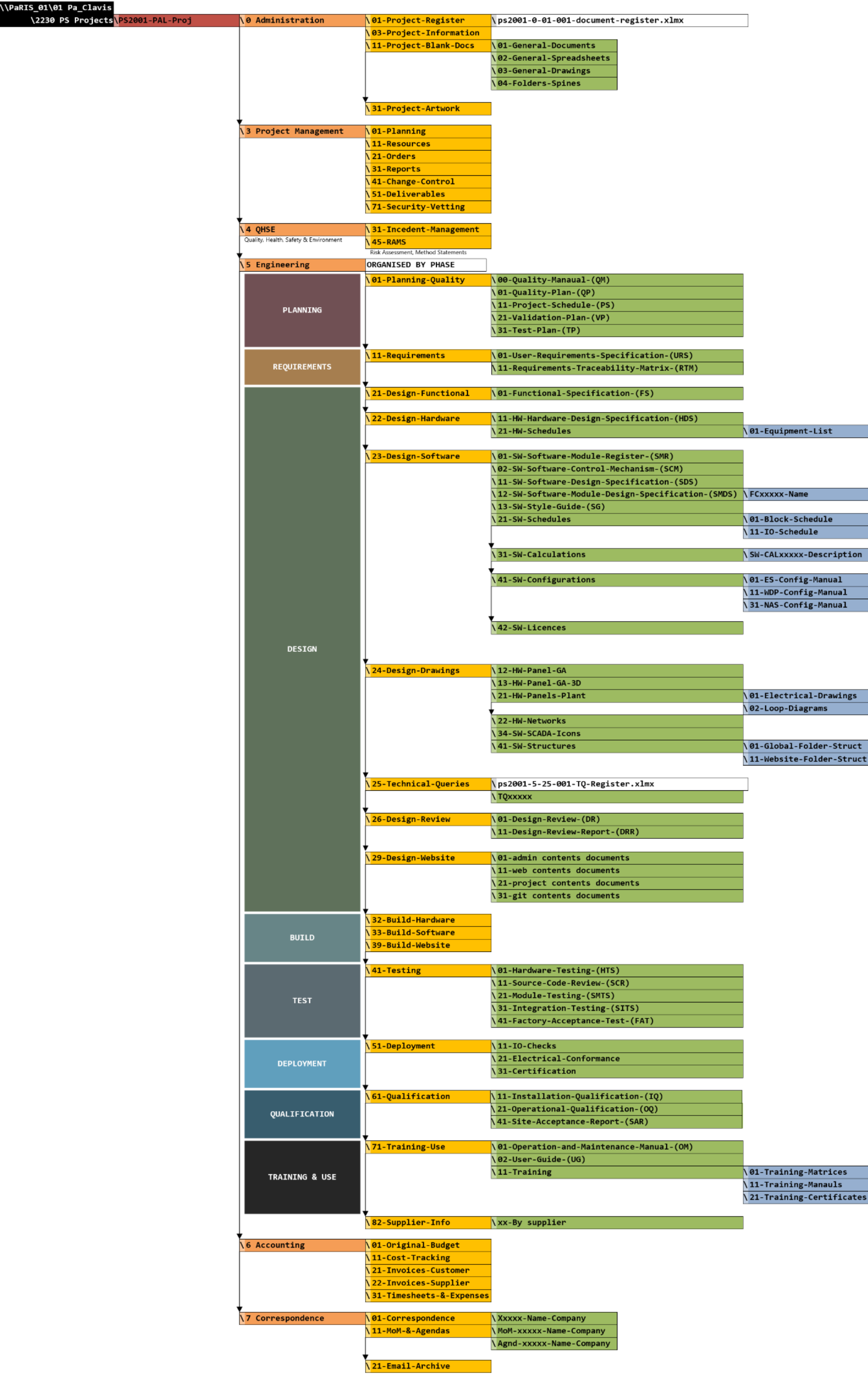


Figure 5.21 The full project folder structure

5.4.1 Understanding the Project folder structure

The Project folder structure is designed to hold all the project information. It is split, generally, according to discipline (management, engineering, financial &c.).

This is a common structure for all PSP projects, this is done to give consistency and commonality to all PSP projects.

The bulk of the information contained within the Project folder structure is documentation, the breakout of the `5-Engineering` directory is organised by phase and contains all the documents listed in the Validation Plan *[Ref. 003]*, documents such as:

① Quality Manual (QM) and Quality Plan (QP)

② Validation plan (VP)

③ User Requirement Specification (URS)

④ Requirement Traceability Matric (RTM)

⑤ Functional Specification (FS)

⑥ Hardware and Software Design Specifications (HDS, SDS)

⑦ Design Review (DR)

⑧ Test documentation (SMTS, SITS, FAT)

⑨ Qualification documentation (IQ, OQ)

⑩ Training and use documentation (UG)

The directory also contains secondary documents such as:

- Design drawings
- Schedules
- Equipment lists
- Certificates (calibration &c.)
- Manufacturer's literature

The directory also holds the data for all aspects of the system including backups of the developed software, licence information, copies of any software supplied to the project (TIA Portal media &c.) and user configuration information (user names, credentials &c.).

The Project directory contains all the live information for the Project and the entire project (including development build information) can be recreated from the information contained within this directory.

Each document within the Project, has its own folder, for example the Functional Specification is located in the folder:

`PS2001-PAL-Proj\5-Engineering\21-Design-Functional\01-Functional-Specification`

The document filename reflects this location, in this the example, the FS filename is:

`PS2001-5-2101-001 R01.00 PAL FS.docm`

All documents have this format, it can be broken down as follows:

`PSnnnn-A-BBCCDD-PPP Sxx.yy Name`

Where `PSnnnn` is the project number (2001 in this case),

`A-BBCCDD` is the leading directory numbers in the path to the document from the root of the project folder, the `A` being one of the main project areas (Table 5.9), 5 in this case.

`BBCCDD` are the remaining folder numbers, the FS is in folder:

`5-Engineering\21-Design-Functional\01-Functional-Specification`

Taking the leading number from each folder give `5`, `21`, `01` (the FS is three folders deep), hence the first part of the FS filename is:

`PS2001-5-2101`

The `PPP` is a three-digit number to ensure the document is uniquely numbered, for a single document in a particular folder, this is usually `001` (this is at the discretion of the user).

`Sxx.yy` is the revision status of the document, see § 5.4.3:

The `Name` is a meaningful name for the document and can be anything (though generally, shorter is better, the whole thing should be 50 characters or less).

Common document folders

Generally, each PAL document (and drawings, spreadsheets &c.) has its own folder within the Project folder structure. The document itself will be in the root of this folder, the document folder will also contain a common set of sub-folders:

Figure 5.22 Document common sub-folders

The purpose of these folders is as follows:

`11-submitted`	Contains the submitted documents (those with a revision status of R)
`21-Review-Comment`	Contains the marked-up documents with a P status that have been reviewed and received comments from the concerned parties
`51-Figs-images-diag`	Figures, images and diagrams used within the main document (Visio drawings are often used, the Visio file has the same number as the main document)
`52-Reference`	Any reference material pertinent to the main document
`91-Superseded`	All superseded versions of the document (including draft documents)

The following shows an example arrangement for the Functional Specification

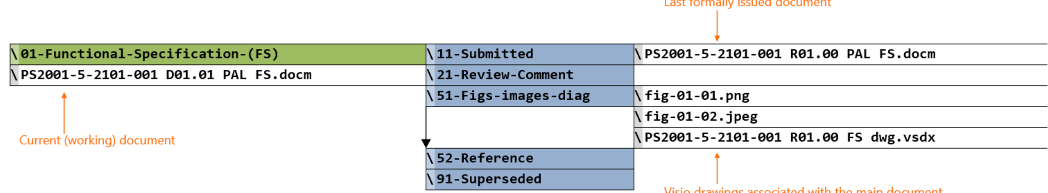


Figure 5.23 Document common sub-folders (example)

Empty folder conventions

The PAL Project folder structure is extensive with a large number of folders, many of which are pre-configured in the PSP folder template used to create the Project directories in the first place.

To make navigation around the folder structure easier, empty folders are, by convention, prefixed with the characters E#, this is the default state for all folders. This can be seen below:

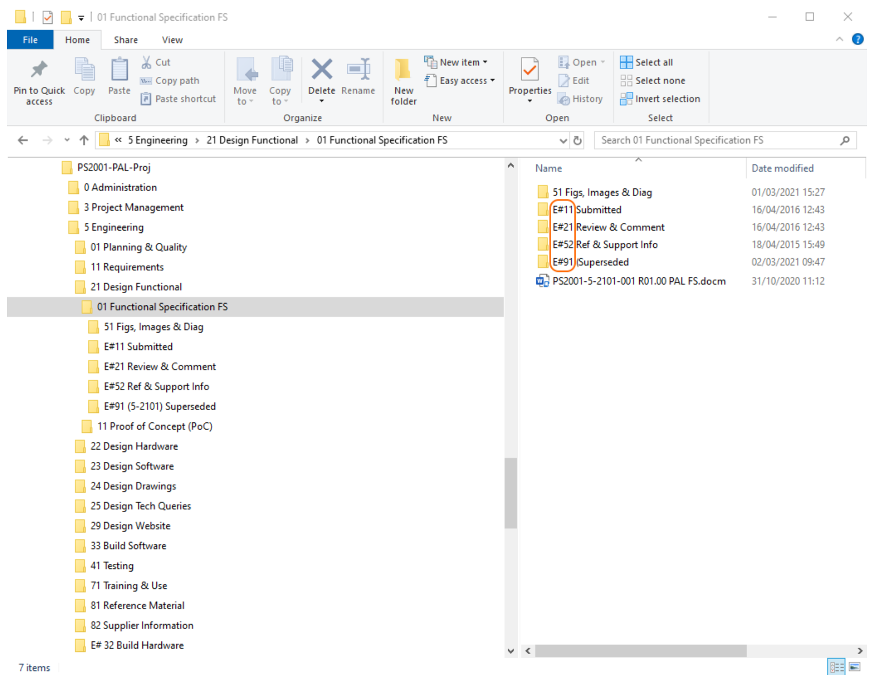


Figure 5.24 Document common sub-folders (example)

5.4.2 Project registry

The `0-Adminstration` folder contains the Project Registry *[Ref.* 005*]*, this is a registry of all the documents produced for the Project

The Project Registry is an Excel spread sheet that lists every document within the Project (usually by phase). It has the file name:

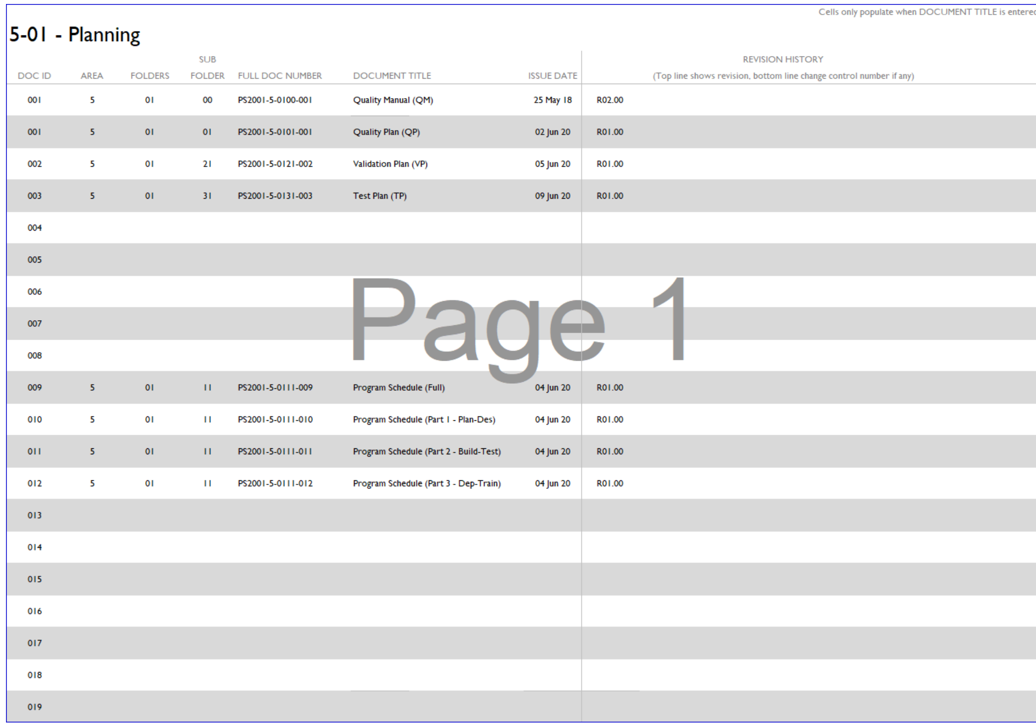
`PS2001-0-01-001 Rxx.yy Project Register.xlsx`

And is located in the following folder

`PS2001-PAL-Proj\0-Administration\01-Project-Register`

Any new document created must be entered in this Project Registry.

The following is an example of a page from the Project Registry:



Cells only populate when DOCUMENT TITLE is entered.

5-01 - Planning

DOC ID	AREA	FOLDERS	SUB FOLDER	FULL DOC NUMBER	DOCUMENT TITLE	ISSUE DATE	REVISION HISTORY (Top line shows revision, bottom line change control number if any)
001	5	01	00	PS2001-5-0100-001	Quality Manual (QM)	25 May 18	R02.00
001	5	01	01	PS2001-5-0101-001	Quality Plan (QP)	02 Jun 20	R01.00
002	5	01	21	PS2001-5-0121-002	Validation Plan (VP)	05 Jun 20	R01.00
003	5	01	31	PS2001-5-0131-003	Test Plan (TP)	09 Jun 20	R01.00
004							
005							
006							
007							
008							
009	5	01	11	PS2001-5-0111-009	Program Schedule (Full)	04 Jun 20	R01.00
010	5	01	11	PS2001-5-0111-010	Program Schedule (Part 1 - Plan-Des)	04 Jun 20	R01.00
011	5	01	11	PS2001-5-0111-011	Program Schedule (Part 2 - Build-Test)	04 Jun 20	R01.00
012	5	01	11	PS2001-5-0111-012	Program Schedule (Part 3 - Dep-Train)	04 Jun 20	R01.00
013							
014							
015							
016							
017							
018							
019							

Page 1

Figure 5.25 Document common sub-folders (example)

5.4.3 Document versions

The revision of the document is expressed in the form `Sxx.yy`, where:

`S` is the status:

`D` – Draft/development

`P` – Published for review

`R` – Released

The `xx.yy` numbers are the revision number, `xx` being the major revision and `yy` being a minor revision.

The first formal release of the document will be at 01.00, prior to this the document will have been is a draft state (e.g. `D00.01`, `D00.02`, `D00.03` &c.) at some point it will have been published for review (this takes the next logical number, e.g. `P00.04`).

Revisions after a document has been released continue with minor revisions from the released revision, consider a document at release R01.00 that is to be modified and re-released, its progression would continue as:

`R01.00` → `D01.01` → `D01.02` … `P01.09` → `R02.00`

The status letter changes to reflect the document state, the numbers always go upwards.

Document revision in document references

Where documents are referenced from within other documents, e.g.:

Validation Plan (VP) *[Ref. 003]*

The current revision of the document is not quoted, neither is it quoted in the References section of the document, this is to prevent every document having to be changed if a single document is modified (changing the revision of the SDS would require the reference section of all documents that referenced it to be change, this in turn would require all documents that referenced these documents to also be updated &c.).

To prevent this, document references quote the document number only, the latest revision of which is listed in the Project Registry *[Ref. 006]*. When using the document reference, the Project Registry must be consulted to ensure the correct revision of the referenced document is used.

At the end of the Project when no further document changes will take place (i.e. when all as-built documentation is released) all document references will be updated to include the as-built revisions of all related documents for clarity.

BLANK PAGE

6 References and glossary

6.1 Document references

The following documents are referenced in this manual:

REF	DOCUMENT NO.	AUTHOR	TITLE/DESCRIPTION
001	PS2001-5-0121-002	PSP	Validation Plan (VP)
002	PS2001-5-2101-001	PSP	Functional Specification (FS)
003	PS2001-5-2311-001	PSP	Software Design Specification (SDS)
004	PS2001-5-2302-011	PSP	Software Control Mechanism (SCM) THIS DOCUMENT
005	PS2001-0-01-001	PSP	Project Document Registry
006	PS2001-5-234101-001	PSP	ES/WDP Configuration Manual
007	PS2001-5-2301-001	PSP	Software Module Register

Table 6.1 Table of references

6.2 Glossary of terms

ABBREVIATION	DESCRIPTIONS
AMD	Advanced Micro Devices, a company that makes computer processors
CSS	Cascading Style Sheet
DB	Data Block
DR	Design Review
ES	Engineering Station
FAT	Factory Acceptance Test
FB	Function Block
FC	Function
FS	Functional Specification
FTP	File Transfer Protocol
Git	A version control system application
GitHub	The online version of Git
HDS	Hardware Design Specification
HMI	Human Machine Interface
HTML	Hypertext Mark-up Language
ID	Identifier
IP	Internet Protocol
IQ	Installation Qualification
JS/JavaScript	A web-based scripting language
jQuery	A library of JavaScript objects, commonly used in web development
MES	Master Engineering Station
MIT	Massachusetts Institute of Technology (Licence)
MWDP	Master Web Development Platform
NAS	Network Accessible Storage
OB	Organisation Block
OQ	Operational qualification
PAL	Practical Series Automation Library
PC	Personal Computer
PLC	Programmable Logic Controller (a Siemens Controller)
PoC	Proof of concept
PSP	Practical Series of Publications
QHD	Quad High Definition

ABBREVIATION	DESCRIPTIONS
QM	Quality Manual
QP	Quality Plan
RAM	Random Access Memory
RTM	Requirements Traceability Matrix
SCADA	Supervisory Control and Data Acquisition
SCL	Structured Control Language (a PLC programming language)
SCM	Software Control Mechanism
SDS	Software Design Specification
SHA-1	Software Hash Algorithm 1
SIT	Software Integration Test
SITS	Software Integration Test Specification
SMT	Software Module Test
SMTS	Software Module Test specification
SSH	Secure Shell, a secure network transfer protocol
TIA	Totally Integrated Solutions (TIA Portal, a Siemens programming tool)
TOC	Table of contents
UT/UDT	User Data Type
UG	User Guide
URS	User Requirements Specification
VCS	Version Control System
VP	Validation Plan
WDP	Web Development Platform
WinSCP	Windows Secure Copy, a file transfer program
XML	Extensible Mark-up Language
Zip	A file extension for compressed files
Zap16	A file extension for TIA Portal compressed files

Table 6.2 Glossary