National Physical Laboratory

Report

NPL REPORT
DEM-ES 023

Best Practice in Software Development - A Case Study in LabView illustrated by the UHTBB Safety System Monitor Project

J Mountford and G I Parkin

Not restricted

March 2007

FUNDED BY THE DTI

Best Practice in Software Development - A Case Study in LabView illustrated by the UHTBB Safety System Monitor Project

J Mountford and G I Parkin
Mathematics and Scientific Computing Group

March 2007

ABSTRACT

This report outlines a case study in the application of SS*f*M Best Practice Guide No. 1, "Validation of Software in Measurement Systems". The guide provides recommended techniques for those developing software for measurement systems to ensure the software is fit-for-purpose, and to meet the requirements of ISO 9000. The case study shows the practical application of the guide to the "UHTBB" project, a project to monitor various parameters indicating the state of the system.

The report also shows how the same project adheres to the NPL software procedure, NPLQP/M13, which is part of the NPL quality system. The procedure assures that software development at NPL meets the requirements of ISO 9000. This report provides an illustrated use of the guide and the NPL procedures, to assist those who plan to use them as part of their software development.

The report also indicates problems in validating LabView programs for systems requiring a Measurement Software Level of 2 or above.

ISSN 1744-0475

National Physical Laboratory
Hampton Road, Teddington, Middlesex, United Kingdom. TW11 0LW

We gratefully acknowledge the financial support of the UK Department
of Trade and Industry (National Measurement System Policy Unit)

Approved on behalf of the Managing Director, NPL
by Jonathan Williams, Knowledge Leader of the Electrical and Software team

Contents

List of Tables

List of Figures

1. Aims

This report outlines a case study in the application of SS*f*M Best Practice Guide No. 1, “Validation of Software in Measurement Systems” (BPG1) [1]. The BPG1 provides recommended techniques for those developing software for measurement systems to ensure the software is fit-for-purpose, and to meet the requirements of ISO 9000. The case study shows the practical application of the BPG1 to the *UHTBB* project, a project to monitor various parameters indicating the state of the system. The aim of the case study is to assist those developing software, explaining how the software can be made fit-for-purpose.

The report also shows how the same project adheres to the NPL software procedure, NPLQP/M13, which is part of the NPL quality system. The procedure assures that software development at NPL meets the requirements of ISO 9000. This report provides an illustrated use of the BPG1 and/or the NPL procedures, to assist those who plan to use them as part of their software development.

The report describes the *UHTBB* project in section 2, and describes the Best Practice Guide No.1 methodology in section 3. Section 0 describes how the validation techniques were applied. Section 4 explains the issues in using LabView for some types of systems. Section 5 describes the application of the NPL quality system.

2. Description of the UHTBB project and its purpose

The UTTBB software will be used to display concurrently the parameters that are monitored by the Ultra-High Temperature Blackbody (*UHTBB*) Safety System electronics. This information, which is collected and reported via a data logger unit, includes cooling water flow rates, surface temperatures and argon gas flow rate. The data displayed will provide the operators of the *UHTBB* with clear, unambiguous and concurrent information regarding the performance of the safety system.

3. Application of Best Practice Guide No. 1

This BPG1 concerns the best practices to be followed when producing software that forms part of a measurement system. It advocates a risk-based approach to software validation described by the following steps:

1. A *risk assessment,* the purpose of which is to make, an objective assessment of the likely risks associated with a software error. The assessment considers the risk factors (a) criticality of usage, (b) legal requirements, (c) the impact of complexity of control, and (d) the complexity of data processing.
2. Assigning a *measurement software level* indicated by the results of the above risk assessment.
3. Applying *software validation techniques* indicated by the assigned measurement software level.

3.1 UHTBB Risk Analysis

3.1.1 Criticality of usage

The BPG1 gives a choice of four categories on an increasing scale of criticality. These are:

1. **Critical** – if correct operation of the measurement system software is considered sufficiently important that appropriate software validation should be undertaken, without being in any of the higher categories below.
2. **Business critical** – if failure could cause serious financial loss, either directly or indirectly
3. **Potentially Life-Critical** – if failure indirectly puts human life at risk or directly puts human health at risk. Such software is considered safety-critical in the context of IEC 61508
4. **Life Critical** – if a failure directly puts human life at risk, and hence is safety-critical

The *UHTBB* software is used to monitor a system it is not part of the safety functionality of the system. The criticality of usage of the *UHTBB* software is therefore **Business critical**.

3.1.2 Legal requirements

Many measurement systems are used in contexts for which there are specific legal requirements, and where a system malfunction could have serious legal consequences. The *UHTBB* software is not used in such a context, and so does not have any legal requirements.

3.1.3 Complexity of control

This is classified in four categories of increasing complexity. The BPG1 provides examples to assist the user in selecting the appropriate category:

1. Very simple – e.g. when the system detects if a specimen is in place, either by means of a separate detector, or from the basic data measurement reading. The result of the detection is to produce a more helpful display read-out.
2. Simple – e.g. temperature control undertaken so that temperature variation cannot affect the basic measurement data.
3. Modest – e.g. if the system takes the operator through a number of stages, ensuring that each stage is satisfactorily complete before the next is started. This control can have an indirect effect upon the basic test/measurement data, or a software error could have a significant effect upon that data.
4. Complex – (This has nothing to do with complex arithmetic!) e.g. the software contributes directly to the functionality of the measurement system. For instance, if the system moves the specimen, and these movements are software controlled and have a direct bearing upon the measurement/test results.

The complexity of control of the *UHTBB* software is **Simple**.

3.1.4 Complexity of data processing

The BPG1 provides four categories of increasing complexity with helpful notes to enable the user to select the right one:

1. Very Simple – The processing is a linear transformation of the raw data only, with no adjustable calibration taking place.
2. Simple – simple non-linear correction terms can be applied here, together with the application of calibration data. A typical example is the application of a

small quadratic correction term to a nearly linear instrument which is undertaken to obtain a higher accuracy of measurement.
3. Modest – well-known published algorithms are applied to the raw data. It is assumed that the algorithms are numerically stable and there is evidence to support this.
4. Complex – anything else

The *UHTBB* software falls into the **Simple** category.

3.2 Calculating the Measurement Software Level

The Measurement Software Level, or MSL for short, is an integer in the range 0 to 4, and is calculated from the criticality of usage, the complexity of control, and the complexity of data processing. The BPG1 provides a simple table [1, Table 5.1] showing the level for each possible value of the three risk factors. Table 1 is an extract of the relevant section of that table, and clearly shows that the *UHTBB* software has an MSL of 1.

Criticality of Usage	Complexity of Processing	Impact of Complexity of control – **Simple**
Business Critical	Very Simple	1
	Simple	**1**
	Moderate	2
	Complex	2

Table 1: Measurement Software Level

3.3 Selection of Software Validation Techniques and Tools

The purpose of using software validation techniques is to mitigate the risks. The BPG1 says: “If the *supplier* can assure the *user* that appropriate techniques have been applied, then the use of the measurement system in the agreed role can be justified.”

While ISO 9001 has general requirements, the BPG1 [1] recommends specific techniques. In all, twenty techniques are discussed, with advice given on their scope, and examples of their application. A simple selection table is provided in the BPG1, in the form of a table [1, Table 11.2]. It is a simple matter to select the required techniques and read up on them. Table 2 shows all the techniques, with ticks against those relevant for MSL 1:

Guide reference	Recommended Technique	MSL 1
12.2	**Review of informal specification**	✓
12.3	Software inspection of specification	
12.4	Mathematical specification	✓
12.5	Formal specification	
12.6	Static analysis	
12.6	BoundaryValue Analysis	
12.7	**Defensive programming**	✓
12.8	**Code review**	✓
12.9	Numerical stability	
12.10	Microprocessor qualification	
12.11	Verification testing	
12.12	Statistical testing	
12.13	**Structural testing**	✓
12.13	Statement testing	
12.13	Branch testing	
12.13	Boundary value testing	
12.13	Modified Condition/Decision testing	
12.15	Accredited testing	
12.16	**System-level testing**	✓
12.17	Stress testing	
12.18	Numerical reference results	✓
12.19	Back-to-back testing	
12.20	Comparison of the source and executable	

Table 2: Software Validation Techniques

Seven software validation tools are recommended for software with an MSL of 1.

Five (set in bold in Table 2) were chosen to apply to the *UHTBB* software. These are shown in Table 3, together (where appropriate) with the tool used to apply the technique. As can be seen no tools were applied.

A *mathematical specification* [1, section 12.4] was not required as no mathematics of any significance was used. This also means that *numerical reference results* [1, section 12.18] was not required. An independent *review of the specification* [1, section 12.2] was performed.

Technique	Tool applied
Review of informal specification	-
Mathematical specification	Not applicable since only reading values.
Defensive programming	-
Code review	LabView guidelines but no tool [2].
Structural testing	-
System level testing	-
Numerical reference results	Not applicable since no numerical analysis.

Table 3: Techniques and Tools

Defensive programming [1, section 12.7] was used throughout the software coding. Although essentially a design technique, defensive programming has an impact on the validation of software. The technique involves including code to check explicitly (and dynamically) whether a pre-condition on, e.g., a program unit, is satisfied.

The developed code was subject to an independent *code review* [1, section 12.8]. Code review is applied to ensure the readability and maintainability of the source code, as well as potentially to identify software errors. Code review followed guidelines available from National Instruments [2].

Testing involved using *UHTBB* as a “black-box” known as *system testing* [11] and testing of the VIs, known as *structural testing* [1, section 12.13].

4. Issues

For *UHTBB* the MSL of 1 (see section 3) means that the code only really has to meet ISO 9000 requirements, see section 5. It is difficult to see how LabView can be used to develop systems cost effectively to a MSL of 2 or higher for the following reasons:

- Currently there are no effective tools to support testing e.g. JUnit like tools [3]. This obviously can be overcome by writing your own test harnesses.
- No means to measure the effectiveness of testing e.g. the equivalent of code coverage tools [1, section 12.3]. It is not easy to see how this might be overcome.

5. The Software Development Life-cycle

The NPL M13 procedure uses the ‘V’ model for software life-cycle this is illustrated in Figure 1.

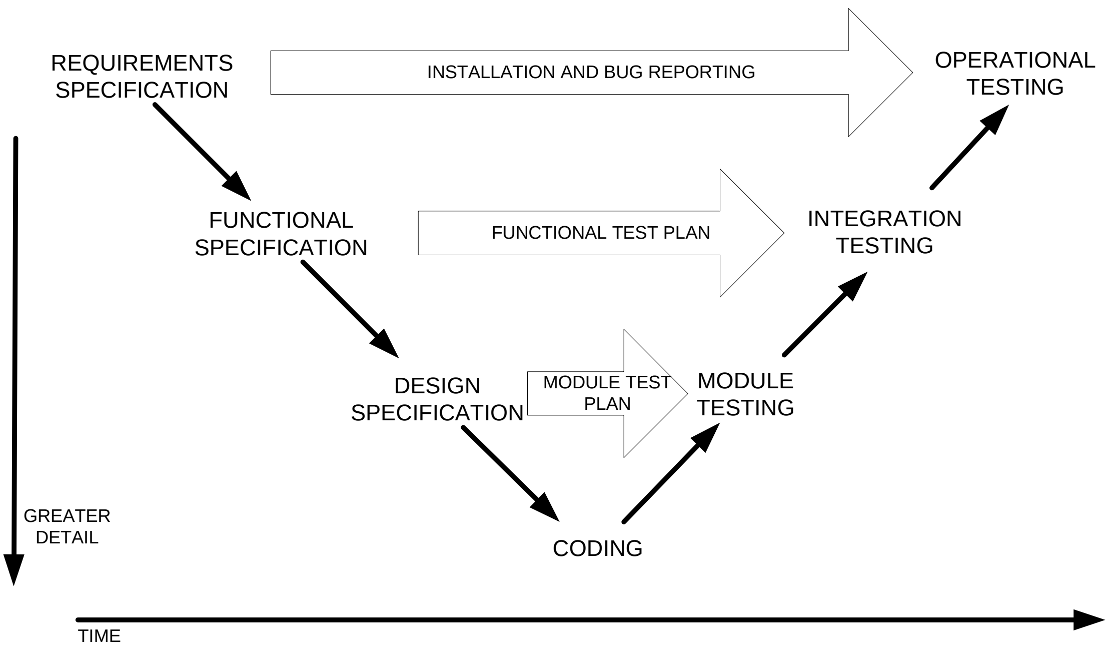

Figure 1 'V' model for software development

The *UHTBB* project is an application developed using LabView to which the full documented lifecycle has been applied.

5.1 Software Quality Plan

Before the software project is undertaken a Software Quality Plan is established, this outlines the requirements for the rest of the software life cycle. The quality plan is written using the NPL software quality plan generator application. The user answers some simple questions regarding the software to be developed then the software calculates the Software Integrity Level (SIL) and generates an electronic copy of the NPL QF59 form with most of the fields populated. The originator of the form must complete section 5 of this form describing configuration management, archiving regime, release rules and maintenance plan.
The QF59 generated for the *UHTBB* is available for examination see reference [4].

5.2 Requirements Specification

The requirements specification is the input to the design process; it must describe what the customer requires the software to do and what limitations are imposed on the developer by the target hardware and budget. It is desirable that the customer writes this document, however in practice the developer may assist in the editing of the document, guiding the customer to a practical set of requirements. This may avoid excessive revisions to the specification as the project evolves.

Some useful sections for the document are:

- why the system is needed
- problems to be solved
- an outline of functions it will carry out
- intended environment – (Target hardware and operating system)
- acceptance criteria
- estimated effort required – (For budget purposes)
- maintenance of software, including bug reporting

This document forms a contract between the customer and the developer and must be agreed by both parties before progressing with the project. The closer the match, the final product is to the requirements, the greater the success.

For the *UHTBB* system, the user requirements specification document was written by the customer see document [5].

5.3 Functional Specification

The functional specification is the developer’s reply to the customer’s requirements, it is more detailed than the requirements document and must contain enough information for a competent person to implement the project in software.
It describes how each requirement is to be met and should highlight any variance if full compliance is not practical. This can be agreed with the customer at this stage and is formalised when the document is signed.
This specification will cover details of how the end-user is to interact with the system, and therefore it is desirable that the customer is involved at this stage.

The specification is a working document and will evolve with each cycle of the development until eventually the customer agrees that all requirements are met. It is therefore very important that strict document control is applied to this document and that the customer, developer and project leader all agree on the changes made.
Typically, the document would cover:

- The software and hardware environment
- A description of the functions the software is to perform
- The data the system will use and generate
- File formats
- How the user will interface to the application
- Special constraints that apply to the system
- Software management

This document will be used as the input to the functional test plan; each function described will require a test to be written to prove compliance with the specification. This is useful to bear in mind when writing the specification.

The *UHTBB* system functional requirements specification document is shown in reference [6].

5.4 Design

The software design document is a record of how the requirements stated in the functional requirements will be implemented in software. The design document may use state diagrams, flowcharts or formal methods to describe the software design. Data structures should be defined, and input and outputs formally described. This document does describe explicitly how the software functions. Code segments and user interface prototypes may be used to further define the design.
Typically, the document would cover:

- program structure
- data structure
- mapping to functional requirements
- coding conventions
- software tools used

This document will form the input to the module test plan; each routine defined in this design will require a test written to verify compliance with the requirements.

In this example LabView is used as the development language. LabView is a rapid development system and provides useful documentation tools that were used in the development of the design document for the *UHTBB* system [7]

5.5 Implementation

When the software design is completed and agreed the coding can commence. The prototypes used in the software design can now be fully implemented and made functional. The coding conventions stated in the design must be adhered to. Constant reference to the software design during coding is desirable and will help avoid variation from the agreed design.

5.6 Module Testing

The module test plan takes the routines defined in the software design document and defines a set of tests required to demonstrate that the routine satisfies the design criteria.

Each test should include the following sections:

- Checkers Identity
- Module unique name and version
- Test Purpose
- Test Method
- Pre-Conditions
- Expected Result
- Actual Result
- Pass or Fail statement.

A good test will demonstrate that the module performs the designed function and how it deals with errors and negative results. Where available test data sets should be run through the module.

An index or test summary form is also helpful when a large amount of modules are being tested; it allows an overview of the test status to be easily seen.

Extracts from the *UHTBB* module test document can be seen in reference [8].

5.7 Functional Testing

The functional test plan takes the functions defined in the functional design specification and tests that the software satisfies the requirement. These tests are at a higher level than the module tests and will test how the modules work together. It will also test the user interfaces and output data for stability and accuracy.

The test plan format can follow the module test plan except that the functional test should reference the functions described in the functional design document.

Taking screen shots with the [Alt] + [Prnt Scrn] keys can be a good way of documenting the test output.

5.8 Delivery

After successful module and functional testing the system is built and an installation image created for distributing the software.

The installer will require checking so it is recommended that some or all the functional tests be executed again on the final installed version of the software.

The released version of the software is documented on the QF 59 in section 6 *For First Release* [5].

5.9 Maintenance

Bug reporting is performed by using a standard NPLs QF-14 form [9][1], when a bug is discovered the finder of the problem fills in the required fields on the form and forwards the report to the software development team. The subsequent releases of the software are tracked on the QF 59 in section 7 *Subsequent releases* [5]. This enables control of the software after release.

6. Conclusion

The BPG1 provides straightforward instructions for conducting an objective risk analysis of measurement software. This analysis results in a single number, known as the "Measurement Software Level" or "MSL" for short. The BPG1 shows, in table format, the validation techniques and tools that are required for each MSL, so it is quite clear how to tackle software development for measurement systems in order that it will be fit-for-purpose, and that the process will conform to ISO 9000. Similarly this case study shows how ISO 9000 procedures can be applied.

The study also shows that to use LabView at an MSL of 2 or higher requires a means to overcome the issues related to test harnesses and a method to measure the effectiveness of the testing that has taken place.

7. References

1. Wichmann, B.A., R.M. Barker, and G.I. Parkin, *Validation of Software in Measurement Systems*. 2004, NPL.
2. National Instruments. *LabVIEW Development Guidelines*. 2003; Available from: http://www.ni.com/pdf/manuals/321393d.pdf#labview_style_guide.
3. XProgramming.com. *Unit testing*. (2004); Available from: http://www.xprogramming.com/software.htm.
4. Mountford, J., *UHTBB Safety System Monitor Software: National Physical Labaratory Quality Plan, QF-59*. 2005, National Physical Laboratory.
5. Gibbs, D., *User Requirements Specification for the UHTBB Safety System Monitor Software*. 2005, National Physical Laboratory.
6. Mountford, J., *UHTBB Safety System Monitor Software Functional Requirements Specification*. 2005, National Physical Laboratory.
7. Mountford, J., *UHTBB Safety system monitor Software Design Specification*. 2006, National Physical Laboratory.
8. Mountford, J., *UHTBB Safety system monitor Module Test Plan*. 2006, National Physical Laboratory.
9. NPL, *NPLQF 49: Complaints and Errors database proforma*. 2005, National Physical Laboratory.

[1] Proquis is a database system used to record all complaints.