MEASUREMENT STANDARDS LABORATORY OF NEW ZEALAND

MEASUREMENT UNCERTAINTY GUIDELINES

A supplement to the MSL Quality Manual

June 25, 2021

Text in this colour is awaiting approval by the Quality Council.

Text in this colour is deprecated and awaiting approval to be deleted.

Measurement Quality Council
Measurement Standards Laboratory of New Zealand

Contents

1 Introduction

MSL relies heavily on software to control experiments and for acquiring, processing and presenting data. Most technical sections develop bespoke solutions that suit their needs.

The quality system imposes requirements on software used to carry out test and calibration work, but does not set expectations about software quality. To satisfy the quality system, software must demonstrably meet the requirements of the task, which will be verified during validation of a technical procedure. The quality system also requires the integrity of both software and measurement data to be strictly controlled.

These are rather modest requirements for assets that represent a very significant investment of MSL's time to develop and maintain. Perhaps unsurprisingly, problems arising from insufficient maintenance of software are rife within the Laboratory.

Much MSL software consists of 'legacy' programs that are used occasionally. Specification of requirements, documentation and software testing are often lacking, making maintenance difficult. Some sections have invested heavily in types of software that are inherently difficult to maintain at the level required for calibration work: spreadsheets being a good example. Spreadsheets are convenient for quick work, but notorious for hiding errors and renowned for allowing inadvertent changes to creep in.

Another problem is the platform required to run legacy software. IT systems evolve much faster than MSL would wish, so older hardware, and older software tools (e.g., compliers) need to be operated after they would have retired elsewhere. In many cases, these older systems are retained because not enough is known to allow migration.

This document has been prepared to present techniques that can improve the reliability and maintainability of software. While recognising that MSL has substantially different requirements to most commercial and even scientific organisations, and that a variety of software development systems are in use, there are general practices that can improve software quality. There are also useful tools available.

This guide recommends a structured approach to software development. The objective is not merely to produce an executable application, it is also to provide: a clear statement of the requirements, an explanation of the design to meet those requirements and a suite of tests that verifies the software's performance against requirements. It is desirable that any part of a software project can be examined and understood by someone other than the author. This facilitates re-use of software and maintenance.

We recommend that software development be undertaken in active consultation among members of the technical section, or wider MSL. As in most quality system activities, the quality of one individual's work will benefit from a review by someone else. Professional developers use regular reviews to both detect errors and improve the quality of their work. These reviews occur throughout the development cycle, not just at the end. MSL should do this too.

2 Style and Structure

"If you don't know where you're going, any road will take you there."
– George Harrison

Software should be developed with end-users and future owners in mind – it should also be designed to be tested and maintained easily. This requires consideration of how software is written.

2.1 Style

When writing, consideration should always be given to stylistic conventions: be consistent in your use of a simple, clear, style. A worthy goal is to write software that is *self-explanatory*. This can be achieved by appropriate use of: names, data structures, programming structures and idioms, layout and comments.

Names Careful naming is important. Names for variables, functions and classes, etc, should be

- easily remembered
- concise
- descriptive (for readers)
- consistent (with names of similar entities)
- easy to chose

Languages such as Visual BASIC, Python, C++, etc, and some end-user tools (e.g., MathCAD) allow considerable freedom to choose names.

Data structures All programming languages support a few basic data structures (e.g., arrays), some allow *ad hoc* structures to be defined for particular uses (e.g., object-oriented structures). Good design of data structures is important, because it can make instructions that manipulate data easier to read, understand and maintain.

Programming structures and idioms Authors in languages like English, etc, use syntactical rules and common idioms to write statements that can be easily understood. The same applies to programming. Each language has its own syntax, features and quirks, but there will always be good, widely-recognised, ways of describing common tasks, which can be regarded as 'best-practice'.

All programming languages have a few basic control structures (e.g., `FOR-NEXT` loops, `IF-THEN` statements, etc). While the logic of these structures is simple and universal, the way they are deployed is usually language-dependent. There are subtleties in how the language works that make certain idioms preferable. Using recognised language idioms both improves code quality and makes it easier to read and maintain.

Layout The way that code appears on the page (screen) is important. Often source code will work fine no matter how it is laid out, but consistent layout makes a huge difference to readability. Once again, languages tend to have different conventions. Python even incorporates layout in the syntax of its block statements, in a deliberate effort to make code easier to read.

Comments Software comments are something of a necessary evil: they should be used sparingly. It is better to revise code, considering appropriate naming, layout and idioms, because doing so may make a comment unnecessary.

While changes are made to code over time, comments are often overlooked. So, comments can drift with respect to the corresponding code base. Later, it may become unclear just how accurate a comment is. Writing code that does not need many comments is a better strategy. Only details that cannot be inferred from a clear coding style should be added as (preferably) short local comments.

3 Maintainable code

"Maintainable code is more important than clever code"
– Guido van Rossum

4 Data management

No problem is too small or too trivial if we can really do something about it.
– Richard Feynman

Digital data and associated software are critical MSL assets. Like any other assets, they should be cared for and maintained. This section is about data management planning.

4.1 Data management planning

MSL should identify its requirements for data management so that implementations (specific procedures as well as supporting IT services) can be evaluated.

A Data Management Plan (DMP) describes how data is managed, from initial production, through data processing to, finally, long-term storage. It should cover how data is handled, who is responsible, and how adherence to the plan will be monitored.

The DMP should contain sufficient detail to enable stakeholders (MSL staff, managers, and IT support) to understand the requirements for data management.

4.1.1 Points to be considered in a data management plan

1. How is data generated? What software is used?
2. Data formats (e.g., raw text, .csv, .xlsx, XML, etc).
3. Is data annotated (with metadata)? Are standards used (which)?
4. Where is data stored? How is it organised (common file structure, database, etc)?
5. Size of data (storage and networking requirements)?
6. How is initial (raw) data secured? Level of protection required?
7. How is data processed? What software is used?
8. How is integrity maintained? Is there an audit trail?
9. Do policies limit access to, and sharing of, (raw or processed) data? How are these policies implemented?
10. How are final (published) reports and data related (records of processing, work flows, etc)?
11. How is data, records of work flows, etc, archived?
12. How is software maintained (over time)?

13. How long must data and software be retained?
14. Responsibilities for data (stewardship, custody)?[1]
15. Who is responsible for the DMP (audits, reviews, etc)?

4.1.2 17025 requirements

All commercial calibration data, unless explicitly noted otherwise, will be confidential to MSL (17025 clause 4.2).

When technical records are in the form of electronic data, they shall be maintained according to 17025 clauses 7.1.

Electronic data management systems used by MSL shall comply with clauses 7.11 of the standard, which deals with control of data and information management.

The control of all records shall comply with clauses 8.4 of the standard, which deal with the control of records.

4.1.3 Example: A DMP for RF standards

Raw data is produced by bespoke data acquisition software written in Python. A github repository is used to store this data on the corporate file server, which is cloned on lab and office machines.

Data acquisition software is developed under version control using a cloud-based repository (github). A record of the version(s) used during a job is logged when raw data is collected.

The data formats used are mainly: plain text files, spreadsheet files (.xls and.xlsx) and Python source (text files with extension .py). The text and Python files are largely for storing configuration information and metadata; spreadsheet files contain measurement data. Results of data processing are also generated in plain text and in spreadsheet files; there is also one proprietary binary file format used for some data (the GTC archive).

Note, there are no spreadsheet software requirements. Spreadsheets containing data may used for *ad hoc* calculations but the main data analysis is done in Python.

No metadata standards are followed.

[1] A ‘custodian‘ is usually a manger with responsibility for the business function of data; they will have responsibility for planning and policy-making on data management. A ‘steward‘ is a subject-matter expert with responsibility for how data is used; they perform the work to manage data.

Data is organised in a hierarchy of sub-folders under a job-specific root folder; sub-folder names are generally composed of a time-stamp added to a descriptive root (e.g., Fig. 1 shows sub-folders where the 'sp' stands for S-parameters). The results of data processing are stored in a similar way, using folder names that begin with 'dp' followed by a date and time stamp (see Fig. 2)

The storage requirement for a completed job is typically several megabytes.

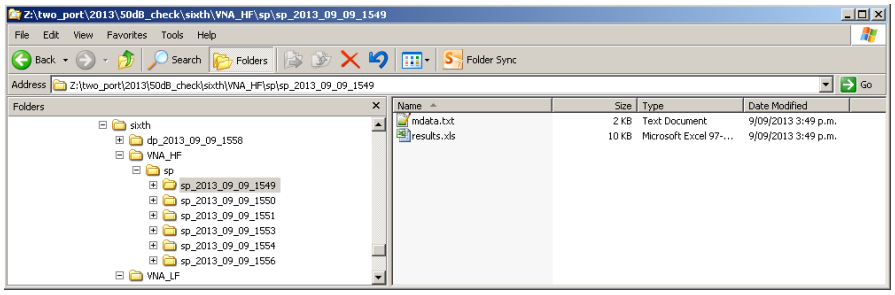

Figure 1: Measurement runs (sub-folders) inside an 'sp' measurement folder (pane on the left) and the contents of one run folder (pane on the right).

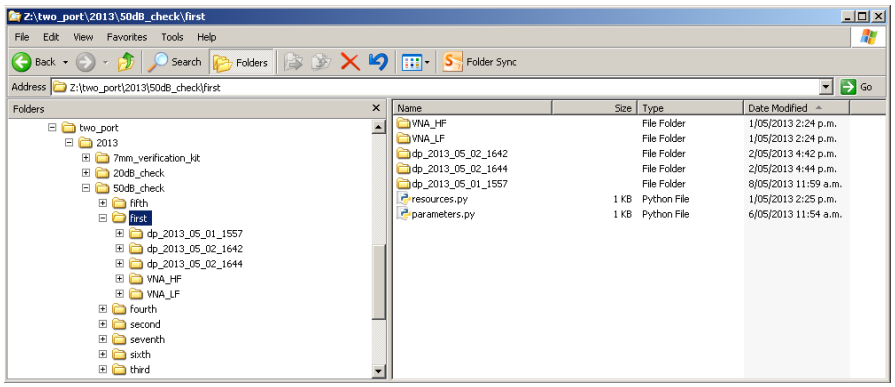

Figure 2: View of a root folder structure for one job (the root is 'first', organised below 'two_port', the year, and '50dB_check'), containing three data processing sub-folders as well as VNA sub-folders in which raw data and configuration information is stored.

Data integrity is maintained by a git repository for each job. The initial job set-up creates a bare repository on the corporate file server (I-drive), which is immediately cloned to the lab computer. After raw data has been collected, it is committed to the local repository by the metrologist and a copy is pushed to the I-drive.

This produces a (complete and independent) lab copy of the data and a remote copy on the I-drive; the I-drive is also routinely backed up by IT service (24-hour cycle).

Using a repository guards against unintended accidental changes to data and also guards against intermittent network problems, by allowing work to be carried out locally until network access is restored.

All data collected in commercial jobs is confidential to MSL by default. So, the 'private' I-drive allocated to MSL is used, which restricts access to MSL staff. No other privacy protections are necessary.

Data is processed by bespoke Python software (also version-controlled) and the results stored in the same folder structure (see Fig. 2). This allows the git repository to capture all stages of analysis. Using a git repository also facilitates review of processing details at a later date, or independent processing on the data without loss of information obtained earlier.

When final client-specific analysis steps are required, the Python script for these last steps will be stored in the section file system commercial job file, which is separately located on the I-drive. The final report is also saved in this commercial job file.

On completion of a job, the repository is left undisturbed on local computers and on the corporate file server. There is no specific 'archiving' process. The amount of data does not warrant compression and any eventual future analysis will be facilitated by maintaining the local file structure.

Data and software must be retained indefinitely.

The reliance on open-source Python software will help to provide on-going access to the functionality of our software. Python, of course, evolves and regular upgrades are expected. A suite of unit-test cases are used to facilitate migration and acceptance-testing of new versions.

Responsibilities for data relate to custody and stewardship. The MSL Director is the nominally the custodian of data, but this is delegated to the Team Manager in charge of the section. The data stewards are MSL staff qualified for the measurement procedure (MSL Competency Matrix).

Adherence to the DMP will be reviewed annually and a record of review kept in the section files (lab book).

A References

[1] Brian W. Kernighan and Rob Pike, *The Practice of Programming*, (Addison-Wesley Professional Computer Series, 1999).

[2] Steve Maguire, *Writing Solid Code*, (Mircosoft Press, 1993).

[3] Brian W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, (McGraw-Hill, 1974).

[4] Diomidis Spinellis, *Code reading: the open source perspective*, (Addison-Wesley, 2003); (additional information available on the books website)

[5] Diomidis Spinellis, *Code quality: the open source perspective*, (Addison-Wesley, 2006); (additional information available on the books website)

[6] Richard Blaustein, *Eliminating Spreadsheet Risks: A Guide for Internal Auditors to Regain Control and Limit Exposure*, Internal Auditor Magazine, 2009; online

[7] Greg Wilson, D A Aruliah, *et al*, *Best Practices for Scientific Computing*, PLOS Biology, 12(1), 2014; DOI: 10.1371/journal.pbio.1001745

[8] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, *Good enough practices in scientific computing*, PLoS Comput Biol, **13**(6), June 2017, DOI: 10.1371/journal.pcbi.1005510.

[9] Marian Petre and Greg Wilson, *Code Review For and By Scientists*, in: Katz D, editor., Proc. Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE 2), 2014; ArXiv:1407.5648, September 7, 2014; online

[10] Working Towards Sustainable Software for Science: Practice and Experiences, workshop presentations available online