MEASUREMENT STANDARDS LABORATORY OF NEW ZEALAND

SOFTWARE DEVELOPMENT AND MAINTENANCE GUIDELINES

A supplement to the MSL Quality Manual

April 3, 2020

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

Every section in MSL relies on a suite of software tools for acquiring data, processing data and presenting data. No two sections use the same set of tools: most sections have developed bespoke solutions that suit their needs; there have been a few cases where software was developed externally under contract.

The quality system imposes a common set of requirements on the software used to carry out test and calibration work: it must be fit-for-purpose, it must demonstrably meet the design requirements of the task to be performed, and the integrity of software must be maintained.

There is also a requirement to maintain the integrity of all measurement data.

A few relatively simple practices can be followed to improve software quality and help meet the expectations of MSL. There are also useful software tools available.

When writing new software in a modern programming language, these tools and practices can be incorporated without much effort. However, much of our software is in the form of ‘legacy’ programs and spreadsheets: software that may be modified from time to time, but which is unlikely to be redeveloped from scratch.

Another difficulty is that some sections have invested heavily in types of software intended for quick and easy programming by the end-user. This inherent flexibility makes them harder to maintain: spreadsheets being the best (or worst) example. Spreadsheets are convenient for quick work, but hard to validate and notorious for hiding errors.

Given the wide variety of software in use at MSL, the most useful guidance is likely to be a collection of simple ideas, rules, or principles that can be adapted to suit a wide range of applications. The purpose of this document is to provide such general guidance.

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 modified easily. All of which requires consideration about how software will be structured and written.

2.1 Style and idiom

Writing software, writing prose, or even writing mathematical formulae, has much in common.

Any language (English, French, mathematics, etc) has syntactical rules and common idioms. We work with these to create statements that others can readily understand. A programming language is much the same. Although each language has its features and quirks, there will be good ways of describing operations and there will be awful alternatives; there will be 'best-practice' and there will be ghastly 'hacks'. When writing software, consideration should always be given to another reader (or even the author at some distant point in time): be consistent, express yourself using a simple, clear style.

A worthy goal is to write software that is self-explanatory.

Names Procedural languages (such as Visual BASIC, Python, C++, etc) and some end-user tools (e.g., MathCAD) allow considerable freedom when naming things. Use this freedom to make software readable. Names should be concise but descriptive: a reader should easily keep track of the meanings of different variables, functions and classes, etc.

Format and layout The appearance of source code and high-level programming environments, like LabView and spreadsheets, is important. Adopt clear formatting conventions and layout that reflect the structure of a program.

Comments Somewhat counter-intuitively, comments in software are really a necessary evil: they are needed, but should be used as sparingly as possible. It is always better to re-phrase the code itself, using appropriate naming, layout and idioms, if doing so will make a comment unnecessary.

There will often be stylistic conventions in a language. Use them, because readers are more likely to understand your code. If there are alternatives styles, choose one and stick to it. Be consistent: don't swap back and forth between one style and another.

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

[5] Greg Wilson, D A Aruliah, *et al*, *Best Practices for Scientific Computing*, PLOS Biology, **12**(1), 2014; online.