

3 October, 2002

**Best Practice Guide No.  
12:**

**Guide for Test and  
Measurement Software**



Adelard, Drysdale Building, Northampton Square, London EC1V 0HB  
Tel: +44 (0)20 7490 9450, Fax: +44 (0)20 7490 9451

© Adelard and Crown Copyright 2002

**Document details**

**Document control**

Adelard's reference: D/209/7103/1  
Status: Definitive

Version	Review no./issued	Date
v0.1G	R/866/7103/1	30 July, 2002
v0.2C	issued in Draft	1 October, 2002
v1.0	R/872/7103/2	2002-10-03

**Authors**

Tim Clement  
Luke Emmet  
Peter Froome  
Sofia Guerra

## Preamble

This document is a best practice guide on the development of test and measurement software. It provides practical general guidance covering the lifecycle phases of test and measurement software development. It also contains language-specific guidance for several of the most important languages for the development of test and measurement software—LabVIEW, Visual Basic, C/C++, Delphi and Java—as well as guidance on mixed-language programming covering calling subroutines in Fortran and C from these development languages and MATLAB.

Because of the amount of material in the guide, it is supplied as an executable program that installs a version configured to include only those languages of interest to the particular reader.

The guide has been developed for the DTI by Adelard as part of Phase 2 the Software Support for Metrology programme (SSfM-2).



## Contents

Part 1 Overview.....	9
1.1 Scope of the guidance.....	9
1.1.1 Virtual instruments.....	10
1.2 Users and audience.....	11
1.2.1 Target audience.....	11
1.2.2 Signposting.....	11
1.2.3 Levels of guidance.....	12
1.3 Other documents and resources.....	12
1.3.1 Use of Internet references.....	12
1.3.2 Books and other references.....	12
1.4 Worked example.....	12
1.5 Glossary.....	13
Part 2 General guidance.....	15
2.1 Introduction.....	15
2.2 Lifecycle of test and measurement software development.....	15
2.2.1 Overview.....	15
2.2.2 Rapid application development.....	17
2.2.3 Dynamic Systems Development Method.....	18
2.3 Requirements description.....	19
2.3.1 Requirements review.....	22
2.4 Design.....	23
2.4.1 Design of numerical algorithms.....	25
2.4.2 Timing.....	25
2.4.3 Design modelling.....	26
2.4.4 Design reviews.....	32
2.4.5 Fault tolerance and fail safety.....	33
2.5 Coding.....	34
2.5.1 Introduction.....	34
2.5.2 Coding standards.....	35
2.5.3 Coding and software documentation.....	36
2.6 Verification and validation (V&V).....	39
2.6.1 Testing.....	39
2.6.2 Code reviews.....	43
2.6.3 Static analysis.....	43
2.6.4 System tests.....	44
2.7 Maintenance.....	44
2.8 Configuration management.....	45
2.9 Metrics.....	46
2.10 Software and component reuse.....	46
2.10.1 Software libraries and in-line code reuse.....	47
2.10.2 Reusable software components.....	47
2.11 Mixed language programming.....	49
2.12 T&M software assurance.....	50
2.12.1 Overview.....	50
2.12.2 Risk assessment and mitigation.....	51
2.12.3 Process design.....	52
2.12.4 Assurance of software packages and components.....	55
2.13 Human factors in T&M software development.....	56

2.13.1	General human factors issues .....	56
2.13.2	Human Computer Interface (HCI) design .....	57
2.14	Organisational support and leverage .....	62
2.15	Operation .....	64
Part 3	Technology-specific guidance .....	65
3.1	Introduction .....	65
3.2	How to select appropriate tools .....	65
3.2.1	Initial selection .....	65
3.2.2	Upgrading existing tools.....	66
3.3	LabVIEW .....	67
3.3.1	Introduction .....	67
3.3.2	Requirements description .....	67
3.3.3	Design.....	67
3.3.4	Coding .....	72
3.3.5	Verification and validation.....	74
3.3.6	Maintenance .....	75
3.3.7	Configuration management .....	75
3.3.8	Metrics.....	76
3.3.9	Mixed language programming with LabVIEW .....	76
3.4	Visual Basic.....	77
3.4.1	Introduction .....	77
3.4.2	Requirements description .....	78
3.4.3	Design.....	78
3.4.4	Coding .....	81
3.4.5	Verification and validation .....	81
3.4.6	Maintenance .....	81
3.4.7	Configuration management .....	82
3.4.8	Metrics.....	82
3.4.9	Further resources .....	83
3.4.10	Mixed language programming with Visual Basic .....	83
3.5	C/C++ .....	85
3.5.1	Introduction .....	85
3.5.2	Benefits and pitfalls.....	86
3.5.3	Further help .....	87
3.5.4	Mixed language programming with C/C++.....	87
3.6	Java.....	88
3.6.1	Introduction .....	88
3.6.2	Requirements description .....	89
3.6.3	Design.....	89
3.6.4	Coding .....	90
3.6.5	Verification and validation .....	91
3.6.6	Maintenance .....	91
3.6.7	Configuration management .....	91
3.6.8	Metrics.....	92
3.6.9	Further resources .....	92
3.6.10	Extending Java with code from other languages .....	92
3.7	Delphi .....	93
3.7.1	Introduction .....	93
3.7.2	Requirements description .....	93
3.7.3	Design.....	93

---

3.7.4 Coding .....	94
3.7.5 Verification and validation .....	95
3.7.6 Maintenance .....	95
3.7.7 Configuration management .....	95
3.7.8 Metrics .....	96
3.7.9 Further resources .....	96
3.7.10 Extending Delphi with code from other languages .....	96
3.8 MATLAB .....	98
Appendix A Internet resources .....	99
A.1 User interface design .....	99
A.2 Component libraries on the Internet .....	99
A.3 C/C++ resources .....	100
A.4 Visual Basic resources .....	100
A.4.1 General sites for VB developers .....	100
A.4.2 Visual Basic coding standards .....	101
A.5 LabVIEW resources .....	101
A.6 Java resources .....	101
A.6.1 General sites for Java developers .....	101
A.6.2 Java coding standards .....	101
A.6.3 Java tools .....	102
A.7 Delphi resources .....	102
A.7.1 General sites for Delphi developers .....	102
A.7.2 Delphi coding standards .....	103
A.7.3 Delphi tools .....	103
A.8 Other T&M software technologies .....	103
A.9 General resources .....	104
Appendix B Books and references .....	105
B.1 Books .....	105
B.1.1 LabVIEW .....	105
B.1.2 Visual Basic .....	105
B.1.3 C/C++ .....	105
B.1.4 Java .....	106
B.1.5 MATLAB .....	106
B.1.6 General .....	106
B.1.7 Related guides .....	107
B.2 Other references .....	107
Appendix C Visual Basic example .....	109
Appendix D LabVIEW example .....	111
Appendix E Examples of mixed language programming .....	113
E.1 Calling a Fortran DLL from Microsoft Visual C++ .....	113
E.2 Calling a Fortran Subroutine from Delphi .....	114
E.2.1 Multi-Dimensional Array .....	115
E.2.2 Passing Functions and Procedures .....	115
E.2.3 NAG Library Routine D03PCF Example Program Coded in Delphi .....	115
E.2.4 String Handling and Passing .....	119

---

E.2.5 NAG Library Routine G02EEF Example Program Coded in Delphi .....	119
E.3 Calling C procedures from Visual Basic .....	123
E.3.1 Multi-Dimensional Array .....	123
E.4 Visual Basic calling Fortran components .....	124
E.5 Calling Fortran Subroutines from LabVIEW .....	132
E.6 Incorporation of a Fortran Subroutine into MATLAB .....	132

## Figures

Figure 1: T&M software lifecycle .....	16
Figure 2: RAD lifecycle .....	17
Figure 3: Requirements description .....	21
Figure 4: Requirements block diagram .....	23
Figure 5: Top-down design .....	24
Figure 6: Data-flow diagram .....	27
Figure 7: State transition diagram .....	28
Figure 8: Object-oriented design .....	29
Figure 9: Sequence diagram .....	31
Figure 10: Statechart diagram .....	32
Figure 11: Test specification .....	41
Figure 12: Use standard controls to invite standard user behaviours .....	58
Figure 13: Label controls to indicate their function .....	59
Figure 14: How to confuse with use of colour .....	60
Figure 15: Use select boxes for enumerated options .....	61
Figure 16: Employ input validation routines to check user data .....	61
Figure 17: Check interface controls for unexpected behaviours .....	62
Figure 18: State transition diagram implemented in LabVIEW .....	70
Figure 19: Error handling .....	73
Figure 20: Coverage measurement for LabVIEW .....	75
Figure 21: LabVIEW metrics .....	76
Figure 22: Standard palette of VB interface controls for Visual Basic (Professional version) ...	80

## Tables

Table 1: Comparison of documentation styles .....	37
Table 2: Techniques for measurement software levels .....	54
Table 3: Data dictionary for LabVIEW program .....	71
Table 4: Data dictionary for Visual Basic program .....	79



## Part 1 Overview

This guide aims to provide practical assistance for the developers of test and measurement (T&M) software, based on current best practice. It presents a simple lifecycle for T&M software development and gives examples of techniques and methods that can be applied at each lifecycle phase.

The guide has been produced as part of Phase 2 of the DTI's Software Support for Metrology (SSfM-2) programme, details of which are given at <http://www.npl.co.uk/ssfm/>. Several other guides produced under this programme are relevant to T&M software.

The guide is based on three best practice guides produced under the first SSfM programme: those covering the development of software for virtual instruments [39], the development of software for metrology [40], and mixed language programming [41]. This guide addresses the comments made on these earlier guides. A key point is the importance of developing and using T&M software in a manner commensurate with the risk from errors in the measurement results, which can be achieved by determining level of assurance for the T&M software (Section 2.12) and applying appropriate development methods and techniques from this guide. The guide is intended to contribute to an organisational environment that favours good practice and develops the competencies of its staff—this is elaborated in Section 2.14.

Because of the amount of material in the guide, it is supplied as an executable program that installs a version configured to include only those languages of interest to the particular reader.

### 1.1 Scope of the guidance

The guide provides guidance on designing, building, testing and documenting T&M software. Both general guidance is provided, applicable to a range of computer platforms and development environments, and also some specific guidance for LabVIEW, Visual Basic, C/C++, Java and Delphi. In the consultation and feedback during the SSfM programme, these were identified as the most widely used T&M software development languages. The guide focuses on the PC because this is the most popular platform; however, the majority of the guidance is applicable to other platforms.

In a guide of this size it is impossible to cover detailed technical issues such as driver design, and indeed it would be unnecessary since there is a wealth of resources already available (see Appendix A and Appendix B) addressing detailed technical matters. The guide meets an identified need in providing a formal structure to T&M software development and maintenance, into which these resources fit.

The aim of the guide is to present the good practices that have evolved in the software development community in a way that is appropriate to the needs of the T&M instrument engineer, who may be working alone, and developing such software only as one aspect of their work.

The guide concentrates on the *development* of T&M software. However, guidance for T&M software users who buy off-the-shelf software components is contained in Section 2.10 on component reuse, Section 2.12.4 on the assurance of software packages, and in Section 3.2 on the selection of appropriate tools.

It should also be noted that by T&M software we mean relatively small systems developed by an individual or small team, and not large-scale data processing systems (e.g. Laboratory Information Management Systems—LIMS) or major process control systems (e.g. for chemical or nuclear plant).

For convenience, the guide uses the term “T&M instrument” to describe the integrated T&M software and hardware.

### 1.1.1 *Virtual instruments*

One particular class of T&M software is that which is part of so-called “virtual instruments” (VIs). VI software can be distinguished from more specialised T&M software because it makes use of a *general-purpose* computer to provide the processing, and uses a computer screen to provide the visual interface to the instrument. The general-purpose computer can also carry out a variety of other tasks by loading other software.

The most widely used computing platform is the IBM-compatible PC, although there are development communities for other platforms such as the Macintosh, VME-based and Unix-based systems. The VI makes use of the services and architecture provided by these computing platforms, in particular:

- standard I/O to hardware (such as serial ports, GPIB and other hardware standards)
- the user interface, in particular a Graphical User Interface (GUI)
- the general-purpose software execution platform, such as an operating system and various run-time libraries
- networking to other computers and devices
- the Internet, for users with remote measuring and monitoring requirements

Various software technologies and development environments exist for developing VIs, such as:

- National Instruments’ LabVIEW
- various versions of Basic (including Visual Basic, QuickBasic and so on)
- other software languages and environments (including C, C++, ActiveX, Java and so on)

Many of these VI attributes may also apply to special-purpose T&M software, and the contents of this guide apply equally to VIs and T&M software in general, unless specifically stated otherwise.

## 1.2 Users and audience

### 1.2.1 Target audience

This guide is intended to be of use to the full range of T&M software developers, including:

- specialist measuring instrument manufacturers
- manufacturers of reference sources
- measurement system integrators
- research workers
- in-house metrology system developers
- test and validation engineers
- auditors, reviewers and calibration signatories

### 1.2.2 Signposting

The main categories of readers of the guide are new users, developer-users, experienced developers, and system or software development managers. We suggest the following starting points for each:

- *New user*—If you are a new user, your main need may be for guidance on getting started, finding resources and avoiding common pitfalls. You are likely to have specialist skills in your measurement domain, yet may come to software development relatively late in terms of education and professional training. We suggest that you begin with the T&M software lifecycle overview in [Section 2.2.1](#), and the section on selection of tools ([Section 3.2](#)) if you have not already chosen them. Then, using [Figure 1](#) as a directory, you can read the parts of Part 2 not marked with “full-scale meter” symbols, and the corresponding language-specific guidance in Part 3.
- *Developer-user*—Within a research context, it is typical for the developer of T&M software to be the same person as the end user. If you are a developer-user, your main need may be for guidance on quality management, configuration management and finding resources. As with the new user, your primary training is likely to be in the measurement domain not software engineering. We suggest you check your current development process against the general guidance in Part 2—the rapid application development lifecycle shown in [Figure 2](#) is likely to fit best. You should pay particular attention to the guidance on assurance in [Section 2.12](#) and check your process is applicable for the software integrity level of your T&M software. You should also make sure you follow the guidance on documentation.
- *Experienced developer*—If you are an experienced developer, you will probably be engaged in T&M software product development for internal customers within your own organisation, or for external customers. You may also provide a T&M software

help desk for your organisation. Your main need may be for guidance on improving the T&M software development lifecycle, sharing complex tool knowledge, user interface design and integration. We suggest that you check your development process against the guidance on assurance in [Section 2.12](#). You may be involved in larger or more critical developments, and so you may need to study the guidance marked with “full-scale meter” symbols. We recommend you also look at the guidance on human factors in T&M software development and HCI design ([Section 2.13](#)), and organisational support and leverage ([Section 2.14](#)).

- *System or software development manager*—If you are a manager concerned with T&M software development, your main need may be for guidance on tool selection, quality management, software assurance and maintenance. As for the experienced developer, we suggest that you check your development process against the guidance on assurance in [Section 2.12](#). All the guidance is relevant to internal quality procedures for T&M software development. We recommend you also look at the guidance on human factors in T&M software development ([Section 2.13](#)) and organisational support and leverage ([Section 2.14](#)). You are likely to be concerned about specific problem areas, and the guidance marked with “full-scale meter” symbols may be of particular interest.

### 1.2.3 Levels of guidance



Three levels of guidance are provided. The majority of the guidance applies to all T&M software. Some additional, more advanced, guidance is provided that is applicable to larger T&M software projects and/or T&M software with higher software integrity requirements; this is indicated in Part 2 by a “full-scale meter” symbol in the left margin, as here.



A small amount of guidance is very advanced, and mainly deals with the highest software integrity requirements. This is indicated in Part 2 by a double “full-scale meter” symbol.

## 1.3 Other documents and resources

### 1.3.1 Use of Internet references

We have gathered a number of key Internet references to help all categories of developer find useful information regarding T&M software technologies such as LabVIEW and Visual Basic, and also some component libraries that are available. Some of these are presented in the narrative of the report, and a complete list is given in [Appendix A](#).

### 1.3.2 Books and other references

There are a number of useful books, newsletters, leaflets etc. available from a variety of sources, and these are listed in [Appendix B](#).

## 1.4 Worked example

The guidance is illustrated with a worked example. In a guide of this size, it is impossible to develop a realistically large example, and one that illustrates the range of T&M software that can be constructed in all the measurement areas. Therefore we have selected a simple example

of a site entry system. This illustrates the basic principles without going into hardware-specific or measurement-specific detail. It also has a control element, since control of actuators is important in many measurement systems.

## 1.5 Glossary

The following special terms and abbreviations are used in the guide.

API	Application Programming Interface.
B	A formal method.
CASE	Computer-Aided Software Engineering.
CIN	Code Interface Node. A LabVIEW feature allowing code in another language to be called.
COTS	Commercial-off-the-Shelf hardware or software.
CSP	Communicating Sequential Processes. A formal method.
DAQ	Data Acquisition.
DDE	Dynamic Data Exchange. A Windows protocol for data exchange.
DSDM	Dynamic Systems Development Method.
FFT	Fast Fourier Transform.
G	The underlying graphical, object-oriented language for LabVIEW.
GUI	Graphical User Interface.
HCI	Human-computer interface.
IDE	Integrated development environment.
IPR	Intellectual property rights.
kloc	Thousand lines of code.
MSL	Measurement Software Level.
PLC	Programmable Logic Controller.
RAD	Rapid Application Development.
SIL	Safety Integrity Level.
SMV	Symbolic Model Verifier. A model checker.

STeP	Stanford Temporal Prover. A model checker.
T&M	Test and measurement.
T&M instrument	The integrated T&M software and hardware.
TCP/IP	A communication protocol.
UML	Unified Modelling Language. See <a href="http://www.uml.org/">http://www.uml.org/</a> .
URL	Uniform Resource Locator. An Internet “address”, e.g. <a href="http://www.adelard.com/">http://www.adelard.com/</a> .
VB	Visual Basic.
VBA	Visual Basic for Applications.
VDM	The Vienna Development Method. A formal method.
VI	Virtual Instrument.
Z	A formal method.

## Part 2 General guidance

### 2.1 Introduction

This part of the guide contains general guidance that applies to T&M software developed using a range of software packages and languages. Language-specific guidance is contained in Part 3. More general information on software engineering methods and techniques is contained in [24] and [25].

This part is structured according to a simple lifecycle, illustrated in [Figure 1](#). This figure also provides a directory to the rest of Part 2.

### 2.2 Lifecycle of test and measurement software development

#### 2.2.1 Overview

Software development processes are often described in terms of a lifecycle. A simple lifecycle for T&M software is illustrated in [Figure 1](#), which also shows where the relevant guidance is located in this document.

The first phase is about understanding the measurement requirements, including the accuracy and dependability needed, and procuring hardware and software that will be able to deliver the requirements. The output from this phase is a requirements description. The next phase presents this more formally as a requirements specification that can either be the basis for in-house development, or for a statement of requirements if the development is to be undertaken by another organisation.

Once the requirements are understood, we can determine the importance of meeting those requirements. This determines the amount of assurance needed in developing the software, and hence some aspects of the process to be adopted, the configuration control to be used, and so on.

The next phase is where the software is designed and developed to meet the requirements. It includes designing-in the appropriate level of assurance that the T&M software will be accurate and reliable enough, for example by making provision for diverse checks on its results. Design and development has to be undertaken within a configuration management system that will enable traceability of the results to the hardware and the software version used.

During the operation phase, the output from the T&M software includes both the basic results and the evidence of traceability and accuracy.

In fact, it is likely that the lifecycle will not be followed in a linear fashion, and there will be some iteration between the phases, for example between risk assessment and requirements gathering.

Finally, most T&M software will be modified during its life. This involves repeating the relevant parts of the requirements analysis and design and development phases. To ease maintenance and modification, the possibility of changes should be taken into account from the inception of the software, by specifying the likely changes in the requirements description and by designing the software to make it easy to change.

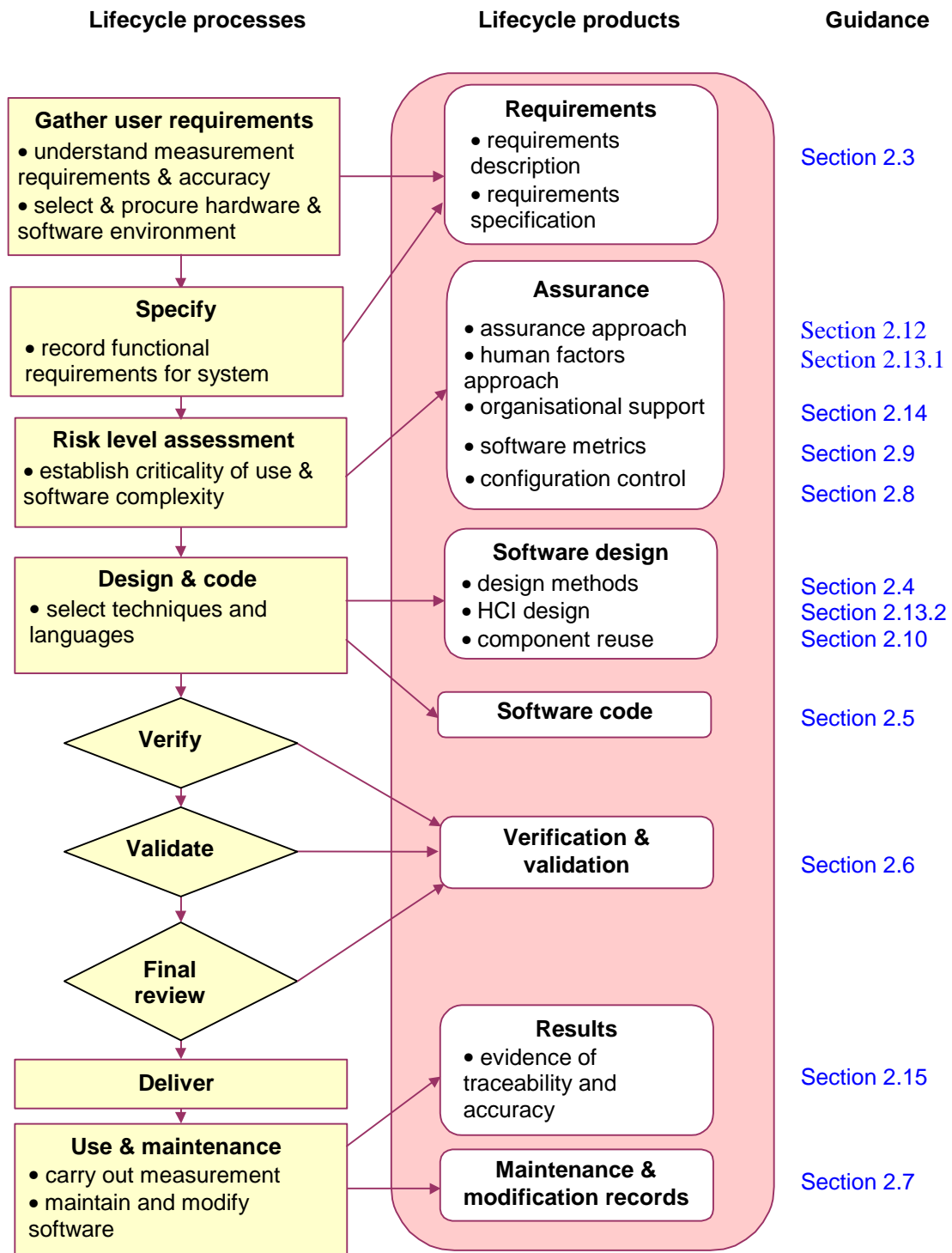


Figure 1: T&M software lifecycle

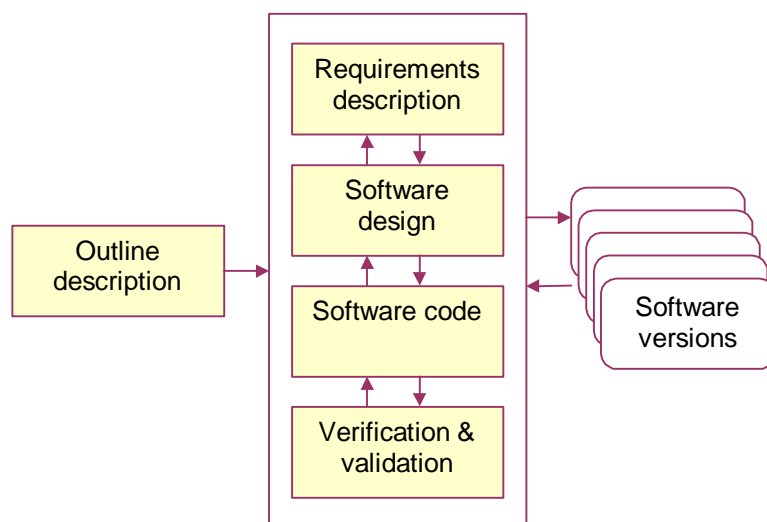
When selecting the actual methods and techniques to use through the lifecycle, it is important that they can demonstrate an adequate level of software integrity. This is discussed in [Section 2.12](#).



Most organisations operate some form of quality management system, within which T&M software development will take place. This is likely to follow the ISO 9000 series of standards. A discussion of quality management in general is outside the scope of this guide, but all the techniques described below can be fitted into a formal quality management system. NPL has a compact quality manual tailored for T&M software projects, and the lifecycle diagram in [Figure 1](#) is consistent with this. This quality manual is available for download from NPL's Website (see [Appendix A.9](#)). A tool for producing a software quality plan based on a risk assessment is also available (risk assessment is discussed in [Section 2.12.2](#) below).

### 2.2.2 Rapid application development

The type of lifecycle described above is often characterised as a “waterfall” development process, in which each lifecycle stage proceeds primarily on the basis of a completed previous stage, although there may be some iteration between phases. Much T&M software is actually developed in a much more iterative way, often in what is known as a *rapid application development* (RAD) lifecycle [30]. RAD is a more exploratory way of developing software in which requirements and design emerge together through the development process, leading to the modified lifecycle illustrated in [Figure 2](#) below. Eventually, of course, the decision is made to deploy and maintain the software, and then the RAD lifecycle reverts to [Figure 1](#). Guidance for the RAD lifecycle phases is located in the same sections of this document as for the equivalent phases of the waterfall lifecycle.



**Figure 2: RAD lifecycle**

The outline description should be written in terms of functions or benefits to be delivered or problems to be solved. From this, a more detailed requirements description is produced for each iteration. The software is then produced and/or modified and reviewed against the outline description and the requirements.

A RAD approach may be applicable to your T&M software development if:

- There is not a definitive statement of the requirements at the start of the project, and you need to evolve the requirements through a series of working prototypes.

- You are also the end-user, or you want the end-user to be closely involved throughout the development process.
- There is uncertainty over the potential pitfalls of the software implementation approach you have adopted, and you need these to be brought to light early in the development process.
- You need to confirm that all the technology to build new measurement sub-systems is available by: establishing the system architecture; demonstrating the performance of algorithms; demonstrating the reliability of control sequences; or demonstrating the correct operation of measurement sub-systems.

Software components (for example based on ActiveX or Corba) are often used as building blocks in a RAD-style process. This is discussed further in Part 3 of this guide.

However, the adoption of a RAD-style lifecycle should not be seen as an excuse for poor documentation or design. The requirements and design documentation described later in this guide should eventually be produced to ensure the software can be tested, maintained and understood by other people—and by the original developer after the passage of time. It is also important to verify and validate the design and code and provide evidence for accuracy and reliability. Good configuration control is also needed so that the developers can revert to the previous version if necessary.

### *2.2.3 Dynamic Systems Development Method*

A more formal way of managing a RAD-style development is as “mini-projects” as described in the dynamic systems development method (DSDM) [33][34][35]. See [Appendix A.9](#) for a link to more information. Each mini-project is essentially a single pass through the RAD lifecycle shown in [Figure 2](#), and has the following general characteristics:

- a fixed time limit for completion—to prevent the mini-project growing into a project in its own right
- high-level prioritised objectives—the priorities allow less important objectives to be dropped in order to meet the “time box” for the mini-project
- early review and verification points
- formal issue of the software from each mini-project as a configuration item
- complete but not over-elaborate documentation

The DSDM method is insistent that work is done properly, so the development products will still be documented, reviewed, tested and released in the normal way. However, elapsed time savings may come from developing functionality in close contact with technical users, doing tasks in the correct order and starting tasks as soon as possible, e.g. review teams should be managed so that they start work on documents as soon as they are available.

## 2.3 Requirements description

Developing a good description of the requirements for the T&M software is a key objective. Errors in the requirements tend to propagate through the development and are only detected during final testing, when they are very time-consuming and expensive to correct. For more information on the process of *requirements capture* see [19].

It may be desirable to categorise requirements, for example identify those that are essential, desirable etc.

On large projects, it is a good idea to separate the *requirements definition*, which is a customer-oriented description of what the T&M software should do, from the *requirements specification*, which is a more precise description aimed at the software designers. However, since most T&M software developments are relatively small projects, and much is developed in-house by user-developers, a single description should suffice for in-house use or as a procurement specification.

There are several special notations for expressing requirements, but for the majority of T&M software development projects natural language is sufficient. However, it will help to avoid ambiguity and loosely worded requirements if a standard format is adopted. The following example (Figure 3) of part of the requirements description for our simple site entry barrier system illustrates a possible format. The key requirements are in bold, and explanatory notes are indicated by the *rationale* keyword.

There are a number of things you should include in the statement of requirements:

1. The functions that the T&M software is to perform, described in natural language, mathematics, diagrams etc. as appropriate. Include a mathematical specification of output data values as a function of the input data values for the simpler processing of basic measurement values (the implementation may compute the results differently, using an efficient algorithm), or, for more complex processing, state or refer to the algorithm. Include sufficient detail so that a user may calculate the uncertainty of the measurement if necessary.
2. The interfaces to the T&M software from sensors and actuators, including specific interface standards that apply.
3. Other inputs and outputs and their formats, e.g. to and from computer files and to printed reports.
4. The human-computer interface (HCI) requirements, covering displays and controls, customisation options, etc.
5. The performance requirements for the T&M software, including:
  - the measurement accuracy required, expressed as the uncertainty of measurement if appropriate
  - timing constraints, identifying the real-time requirements and non-time-critical functions

- reliability, availability and safety requirements, expressed where possible in numerical terms (e.g. *the software shall return the correct value for 99.9% of measurements*) and/or measurement software levels or safety integrity levels (see [Section 2.12](#) and [38])
  - maintainability requirements, identifying those requirements likely to change over the software's lifetime that should be easy to accommodate, such as possible future hardware changes or likely extensions to the functionality
6. Constraints due to the computer used, memory and disk storage limits, etc.
  7. The requirements for built-in assurance features, such as error handling, self-checks, sanity checks, diverse algorithms etc. (this area is discussed in more detail in [Section 2.4.5](#)).
  8. Security requirements, to prevent unauthorised changes to data and programs.
  9. Any applicable standards that apply to the hardware or software (international, national or in-house).
  10. Any special terms used.

## 1. Overview

1.1 **The requirement is for a replacement site entry control system to enable operation by security personnel of an existing barrier and traffic light.** The existing system also provides a sensor that indicates when a vehicle is in the barrier area.

1.2 A plan of the site entry area is given in Figure X.

## 2. Safety features

2.1 **The system shall not allow the barrier to be closed when a vehicle is beneath it.**

2.2 **The traffic light shall be interlocked with the barrier to show red when the barrier is not fully open.**

## 3. HCI

3.1 **The operator's control panel shall provide buttons to open and close the barrier, a mimic of the traffic light, a light showing when the barrier is open, and a warning light showing when a vehicle is beneath the barrier.** The control panel shall be designed in accordance with good human factors principles.

*Rationale:* The traffic light is partially concealed from the security room and the mimic is to give assurance of its correct operation.

3.2 The barrier shall start to move within 0.5 seconds of the appropriate button being pressed.

## 4. Traffic light

4.1 **When the barrier has been fully opened, the traffic light shall change to green. Before the barrier starts to close, the traffic light shall change to red.** The system shall implement a standard UK traffic light sequence.

## 5. Interfaces

5.1 The barrier motor is controlled via an RS-232 interface.

5.2 The traffic light is composed of red, amber and green bulbs, independently controlled via a RS-232 interface.

5.3 The vehicle sensor provides a 0–20mA signal, a value below 10mA meaning a vehicle is present, and otherwise meaning no vehicle is present.

5.4 Detailed interface descriptions are given in Document Y.

## 6. Reliability

6.1 The system shall be implemented to Measurement Software Level 2 as defined in SSfM Best Practice Guide No 1.

Figure 3: Requirements description

### 2.3.1 Requirements review

The requirements should be checked by someone other than the author before development is commenced. Preferably, a formal review should be carried out. This should check that the requirements document is:

1. *valid*, i.e. it correctly describes the functionality and other properties of the software
2. *consistent*, i.e. it is internally consistent and does not contain conflicts
3. *complete*, i.e. it contains all the requirements
4. *realistic*, i.e. it is attainable in the required time using the available hardware and software, and the number and competence level of the development team
5. *comprehensible*, i.e. it can be understood by the development and maintenance teams, and the customer

Make sure that the proposed or required hardware is capable of meeting the performance requirements (accuracy, throughput, etc.). For example, a manufacturer may quote the performance of their data acquisition card as being that of the chip used at its heart, whereas the supporting hardware may compromise performance to the extent that only a fraction of the stated performance is achieved in any particular real application. You may need to produce one or more test programs to check the actual performance of the hardware. Several utility programs are available for checking performance, for example the timing VIs included with [1].

If your T&M software is safety-related or of the higher measurement software levels (see [Section 2.12](#)), you should review the following aspects of the requirements where relevant:

1. the overall architecture
2. the breakdown of component sub-systems, to make sure that the interfaces are clean and modular
3. the boundary between Commercial-off-the-Shelf (COTS) products and any specially developed software
4. external interfaces
5. data or object modelling (see also [Section 2.4.3](#))
6. data requirements and physical database specification
7. non-functional requirement constraints (e.g. reuse, availability, reliability, portability and maintainability)
8. how special features, such as safety or security, are handled
9. adequacy of performance, including the results of performance models where available
10. any specific options or alternatives to solve the T&M problem

11. the impact of advances in equipment capability
12. whether the correct balance has been struck between specially developed versus COTS solutions
13. any limitations imposed by current technology on technical and design options
14. the technical risk
15. compliance with any higher-level requirements document if one exists (e.g. a customer's specification), which can be illustrated by producing a compliance matrix

For complex T&M software, you should consider developing a *system model* as part of the requirements documentation. One possibility is a block diagram that shows the data-flow between system components. A block diagram for the access control system is shown below.

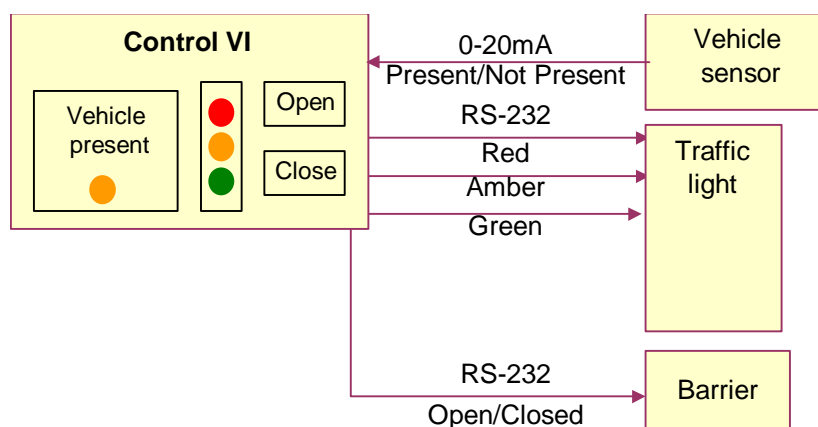


Figure 4: Requirements block diagram

You can also produce a system model using object-oriented techniques. The use of object models is illustrated for the design stage in [Figure 8](#) below. An object-oriented system model would resemble this object-oriented design but be at a higher, more abstract level.

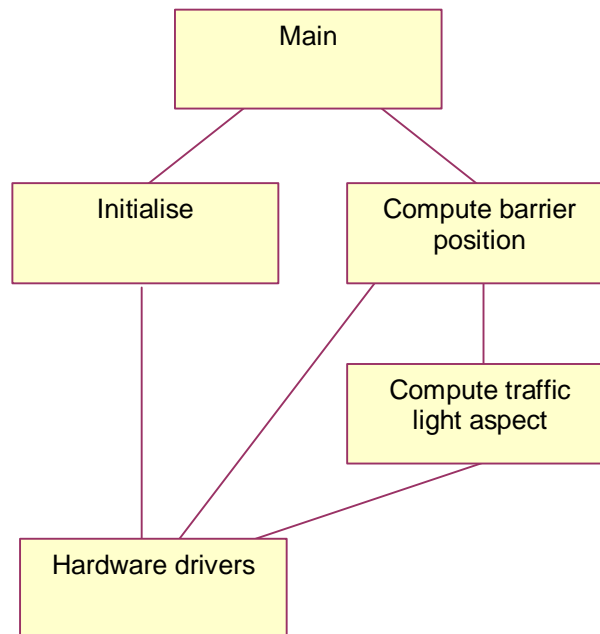
## 2.4 Design

Software design is the phase in which you decide how a system that meets the requirements will be constructed. In complex projects, design can be divided into stages such as architectural design, software specification (or functional specification), high-level design and low-level design. For most T&M software projects, however, a single design step will be sufficient, although you may find it useful to use more than one type of diagram or notation to capture all the design detail.

When creating the design, you should give some thought to how you can demonstrate that the design meets the requirements. This is particularly important for the higher measurement software levels, and is discussed further in [Section 2.12](#) below.

You should also make sure that your design clearly maps onto the implementation technology (e.g. language and COTS items) you have chosen.

Design can be carried out *top-down* or *bottom-up*, or a combination of the two. Top-down design is generally best for big T&M software projects. The approach is to break the software into a number of high-level modules, with a general specification of what they should do. Each high-level module is then broken down some more, and so on, until there is enough detail to enable coding to start. A top-down design for our example is show in [Figure 5](#).



**Figure 5: Top-down design**

Bottom-up design is more appropriate to low-level problems such as driver design or implementation of complex algorithms. Here complete modules are written and tested to carry out specific tasks, and then assembled to provide the full functionality.

In the remainder of this section, we discuss:

- the design of numerical algorithms
- how to deal with timing in design
- notations for design, including data flow diagrams, state machines, data dictionaries, object oriented design, UML and formal methods
- design reviews
- fault tolerance and fail safety

The design of the user interface is discussed in [Section 2.13](#).



### 2.4.1 Design of numerical algorithms

Computer arithmetic is only approximate and the approximations can have a large impact on the accuracy of the results. The design of numerical algorithms is outside the scope of this guide, but needs to be carried out by someone with suitable training and experience. If you choose to use numerical software libraries, the design will need to show the interfaces to the library. See [Section 2.12.4](#) for a discussion of numerical algorithm libraries. This subject is also covered in other SSfM themes; see <http://www.npl.co.uk/ssfm/ssfm1/>.

You should be aware that the basic mathematical functions supplied with programming language implementations can have unexpected properties and are not always as well designed as they could be. As an example of the former, languages may implement exponentiation using logarithms. This will tend not to give the expected integral answer when both base and exponent are integers. As an example of the latter, trigonometric function implementations usually start by reducing the argument to an angle in a  $2\pi$  range around zero. Naïve approaches can result in a large loss in accuracy for large arguments.

### 2.4.2 Timing

T&M instruments are almost always real-time systems, that is to say that the time at which they produce results or carry out actions is important to their correct operation. They may be *hard real-time systems*, in which case out of specification timing is a definite failure (e.g. where a measurement depends on application of power to an actuator for a precise time); or they can be *soft real-time systems*, when poor timing leads to degraded performance (e.g. it causes unnecessary cycling of a temperature-controlled vessel).

You should choose a design method and language for the T&M software that is capable of meeting the timing requirements. Aspects that you need to consider include:

- *Language*—Programs in some languages, such as C (and assembler) have an obvious relationship to the operations that will be executed in the hardware, making the run time of the programs easier to predict. This helps in checking that deadlines can be met. The costs of operations may be less obvious in other languages, particularly graphical ones. One particular complicating factor is the use of automatic garbage collection, which removes the necessity for the programmer to manage storage explicitly (with the attendant hazards of premature release or memory leaks) but may impose a delay at unpredictable moments in the execution.
- *Language implementation*—Implementations that compile to the native machine language of the underlying hardware (which includes most implementations of C and C++) will run faster than implementations based on interpretation of intermediate code (such as Java and most graphical languages). This speed may be necessary to meet processing deadlines. Some implementations will offer different levels of optimisation in the compilation, trading extra compilation time for reduced run time, but you should be aware that the optimising analysis in compilers is complex and compiler bugs are often revealed in these areas.
- *Operating system*—Hard real-time T&M software may require more precise timing than can be guaranteed by operating systems such as Windows. There are real time operating systems for Intel processors, and some of these implement parts of the Windows API, but most of the T&M development tools will require completely

Windows (or similar) operating system environments. This can be resolved by a design based on two or more processors, one running a non real-time operating system and providing the user interface and the other(s) providing real time data gathering and reduction, and implementing any time-critical control. The extra processors may be in independent hardware, possibly in the form of a *programmable logic controller* (PLC), or on a plug-in PC card.

- *External libraries and components*—T&M software often uses software timers for timing or sampling. However, these internal software timers can be disrupted by any external libraries (e.g. DLLs) or components that are called from the application. This problem occurs when the external libraries take the control from the application and do not hand back the control until the operation is finished, preventing the software timers from incrementing during the external call. For example, a sample period of 1 minute could be offset by around 10 seconds each iteration if an external library takes this much time to complete. You should check that the overall timing requirements for the application are not adversely affected by the use of any external libraries or components.
- *Design method*—Fast programs are normally achieved by careful choice of algorithms at the design stage, which may in turn affect decisions on data gathering. For example, the Fast Fourier Transform algorithm is much faster than a naïve implementation of the mathematical definition of Fourier Transforms, but depends on the number of samples being a power of 2. In a few cases, there may be a conflict between modular design and speed, because the information hiding of the modules means that calculations may need to be repeated. This can usually be handled by a careful choice of interfaces between modules.
- *Network latencies*—Increasingly T&M software uses a network to connect different components of the software. You should check that any network latencies or timeouts do not break any timing requirements for the T&M software.

### 2.4.3 Design modelling

There are a number of ways of modelling the design of T&M software. In this section we describe three of them: data-flow diagrams, state transition diagrams and object models. Textual languages such as *pseudo-code* can also be used. You should choose one or more notations that capture the key design aspects of the data structures and procedures in your T&M software. The way these design notations map onto the entry barrier example is shown in Part 3.

It is a good idea to check that the design addresses key properties, e.g. safety properties. For example, the state machine model ([Figure 7](#)) preserves the safety properties that the barrier cannot be closed unless the vehicle sensor is false (i.e. no vehicle is present), and that the barrier is open unless the light is red.



On small projects, these notations can be produced just with a drawing package. For larger projects, you could consider using one of the *Case* (Computer Aided Software Engineering) tools that support these notations.

### Data-flow diagrams

Data flow diagrams model the way in which data (shown as arrows) pass between the different entities in the system. The entities are

- Processes, shown as round-cornered rectangles, which transform data from one form to another.
- Data stores, shown as rectangles, where data reside when they are not flowing. They often represent bulk stores such as files and databases, but in T&M applications we may want to represent simple data representing the system state.
- External entities, such as users or external data stores and processes, or in T&M systems, sensors and actuators, which act as sources or sinks for data flows. These are shown as shadowed boxes.

Figure 6 shows a possible data flow model of the barrier system.

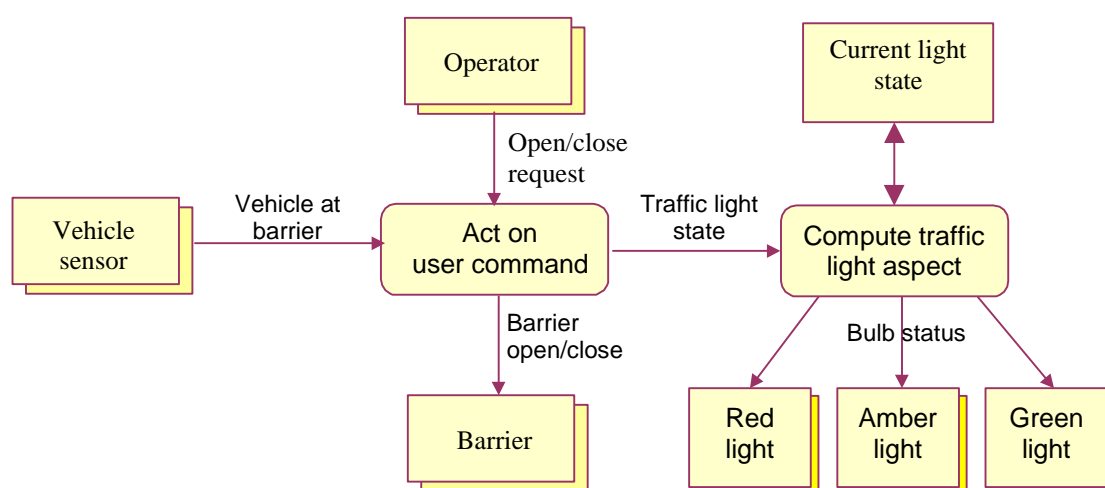


Figure 6: Data-flow diagram

To learn more about data-flow diagrams, see [24] and [25].

### State machine modelling

A suitable way of describing T&M software whose behaviour depends on previous history is *state machine modelling*. The following diagram shows the main part of the state machine for the entry barrier example using the notation for *sequential function charts* defined in IEC 1131 [21].

The diagram shows a number of steps, with associated actions in the box to the right. For example, Step 0 carries out the action “Read Open button”, and sets the value of the feedback variable OPEN accordingly. If OPEN is true (a request to open the barrier has been received),

the system moves to Step 1 (“Open barrier”); if it is false (“¬” means “not”), the system moves to Step 3 (“Read sensor”).

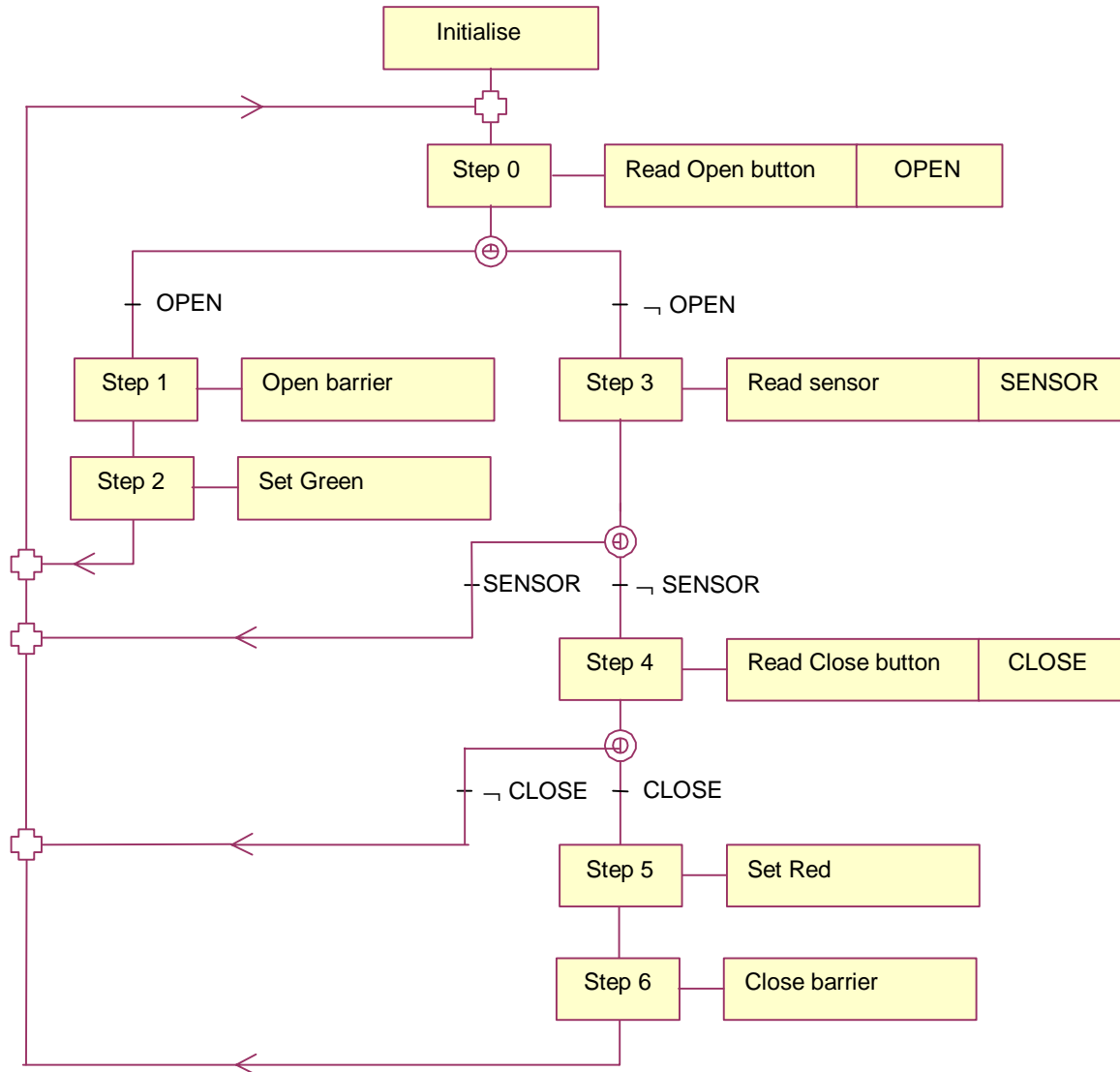


Figure 7: State transition diagram

To learn more about state transition diagrams, see [24] and [25].

#### Data dictionaries

Another very useful design tool is the *data dictionary*. This is basically an alphabetical list of the names used in the design, together with a description of each named item. It can be maintained as a word processor document, hypertext document, spreadsheet, database or within a Case tool.

### Object-oriented design

An alternative design is shown below in object-oriented notation. Each rectangular box represents a *hardware control object*. The top compartment in each box is the object's name, the next lists the *attributes* of the object, and the bottom compartment lists the operations that the object is to implement. The straight lines between the objects show the association between them. The direction of the arrows shows that messages can only be sent in the direction of the arrow. The lines with diamonds at the end show *aggregations*, in this case the way the complete traffic light is composed of three individual bulbs (the figures show that one traffic light is associated with one red light, etc.).

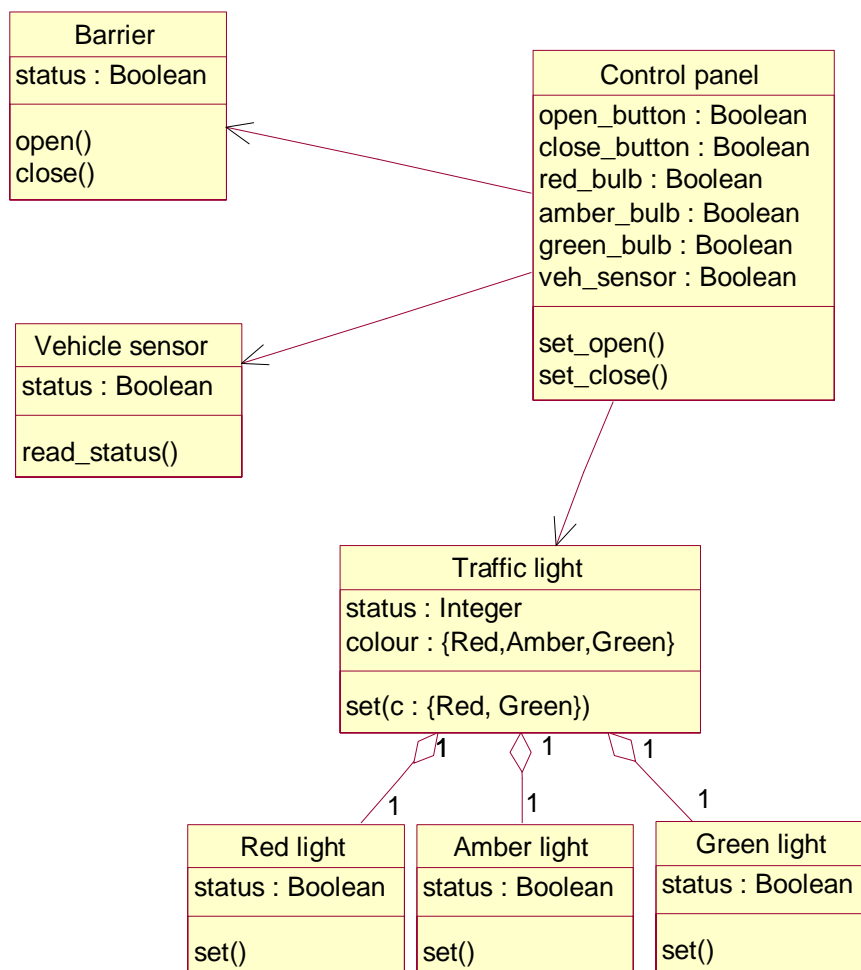


Figure 8: Object-oriented design

To learn more about object-oriented design, see [\[29\]](#).

## UML



For larger T&M software projects, the Unified Modelling Language (UML) (see <http://www.uml.org/>) provides a general purpose graphical language for modelling the system. Although usually associated with object-orientation, UML can be used more generally due to its different types of diagrams and the potential of being extended in controlled ways (using the so-called *extensibility mechanisms*).

There are two aspects to a UML model: the static structure and the dynamic behaviour. Different types of diagrams provide for different views into a model. Typically, a UML model is expressed using a combination of the diagrams below, which are:

- *class and object diagrams*—which include the type of diagram shown in [Figure 8](#) above.
- *use case diagrams*—which can show how external systems and users interact with the T&M software. They describe the functionality of the software as perceived by outside external systems and users.
- *interaction diagrams*—which include the *sequence* and *collaboration diagrams*, which shows the explicit sequence of messages between objects that implement the behaviour of the software. Sequence and collaboration diagrams show the same information, but they emphasise different aspects of the behaviour.
- *statechart diagrams*—which typically describe the behaviour of class instances (i.e. objects), but they can be used to describe the behaviour of other entities. Statecharts are an alternative notation for state transition diagrams such as the one shown in [Figure 7](#).
- *activity diagrams*—which can be used in much the same way as data-flow diagrams such as that shown in [Figure 6](#). Activity diagrams are a variant of statechart diagrams, where the states represent actions or activities.
- *implementation diagrams*—which include the *component* and *deployment diagrams*, which show the source code structure and run-time implementation structure.

Class and implementation diagrams model the static structure, while the other UML diagrams model the behaviour of the software.

As an illustration of how the different diagrams can be used, consider how the behaviour of the barrier system can be modelled. The sequence diagram in [Figure 9](#) shows some of the interactions between the objects defined in [Figure 8](#) during the `set_close` operation of the `Control_Panel`. The flow of time is from the top to the bottom of the diagram. The events correspond to activations of the methods defined in [Figure 8](#), and the method activation times are shown by the boxes on the vertical lines below each object. For simplicity, the diagram omits the initial check on the vehicle sensor, and assumes that the `Green light` is currently showing and the `Barrier` is open. Taking the events in order:

- a message `set(Red)` is sent to `Traffic light` by the `Control panel`

- the Traffic light sets the Green light off and the Amber light on
- the Traffic light sets the Amber light off and the Red light on
- the Control panel closes the Barrier

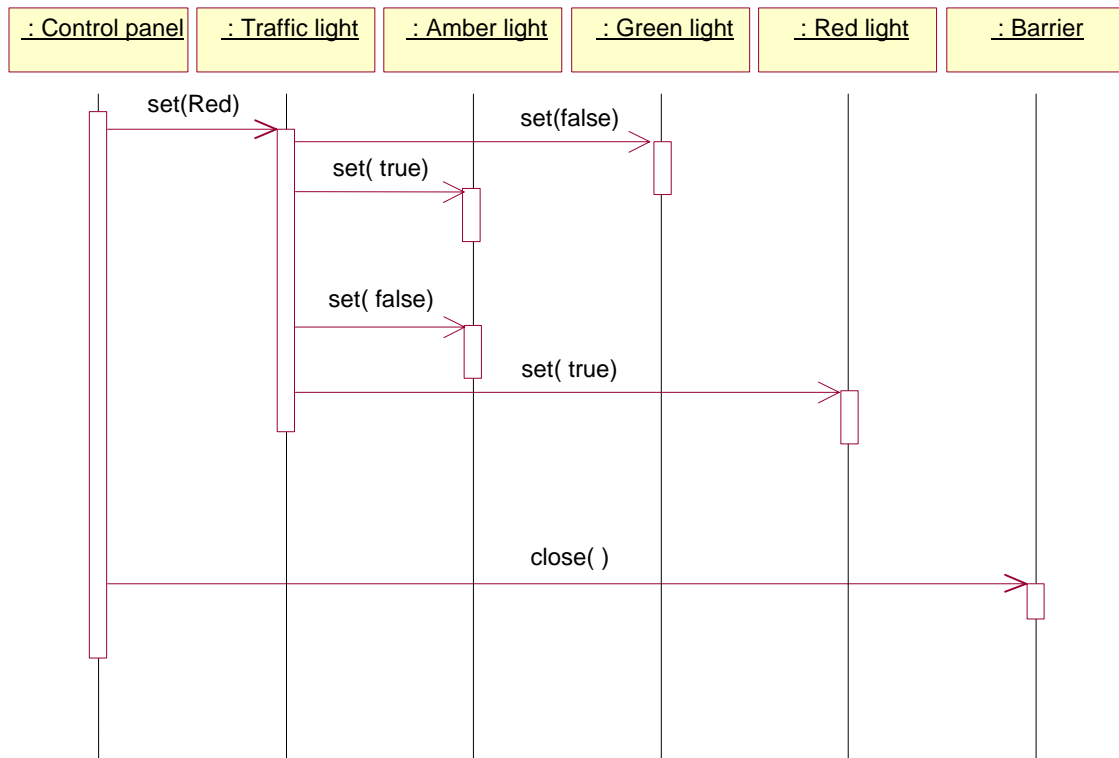


Figure 9: Sequence diagram

A statechart diagram for each component models their behaviour and shows how the components change state as a result of the execution of the operations. Figure 10 models the behaviour of the `Barrier` component. In the diagram below, the `Barrier` has two possible states:

- opened, which corresponds to the case when the attribute `status` is true
- closed, which corresponds to the case when the attribute `status` is false

The state of the `Barrier` changes from opened to closed and vice versa as a consequence of the execution of the operations `close` and `open`.

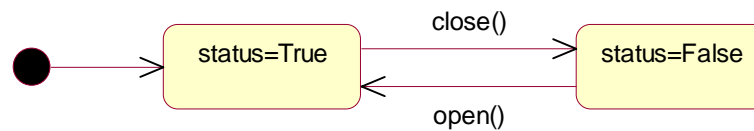


Figure 10: Statechart diagram

Tools are available to support the development of UML models. These tools allow editing of the graphical notation, support the link between different views of the model, and some enable a source code framework to be generated automatically for T&M software languages such as Visual Basic and C.

#### Formal methods



For T&M software of the highest criticality, you should consider writing and validating specifications and designs using a *formal method*. They may also be cost-effective for complex real-time and concurrent T&M software, which may be impossible to reason about informally.

A formal method has two components. The first is a mathematically based notation within which the specifications and designs can be expressed. Some of the notation may be similar to that used in programming languages, but formal specification languages will also have aspects intended for capturing requirements concisely rather than for efficient execution. The other component is a set of rules for reasoning about specifications and designs, so that it can be confirmed unambiguously that the designs have the properties required by the specifications.

Examples of formal methods include B, VDM, Z, and CSP. The UML specification (see) also includes a formal (textual) language that can be used in addition to the graphical notation. The *Object Constraint Language (OCL)* is used to express constraints on the system being modelled, typically class invariants or pre- and post-conditions of operations, which cannot easily be expressed using a diagram. Modelling systems and reasoning about their properties in these languages requires a significant knowledge of discrete mathematics, although there are tools that can help throughout the process. For systems that can be modelled as state machines, *model checking* allows many interesting properties to be established (or disproved) automatically, reducing the amount of specialist knowledge required. SMV (Symbolic Model Verifier) and STeP (Stanford Temporal Prover) are examples of model checking systems, and Statemate provides model checking tools for state charts. See [Appendix A.9](#) for links to more details.

#### 2.4.4 Design reviews

The design should always be reviewed by someone other than the author. If you are operating within an ISO 9000 quality management system, reviews will be a requirement of the system, but they are important anyway to avoid “mind-lock”, where you become blind to potential faults in the design.

Design reviews can be carried out by means of:

- a desk check by an independent reviewer
- a walk-through (one type of walk-through is the Fagan inspection; see [\[22\]](#))



Topics to examine include:

- coverage of requirements
- modularisation
- use of external code modules
- HCI
- error handling
- definition of input ranges
- initialisation and tidying up
- capacity and performance
- correct use of design notations

This list can be expanded to address problems you have encountered with your own T&M software designs.

The review should record agreed changes, and a procedural mechanism should be put in place to ensure they are carried out and checked.

#### *2.4.5 Fault tolerance and fail safety*



A system is said to be fail-safe if it can determine when dangerous faults have arisen, either within the system or in the environment where it is being used, and go to a safe state. In the case of a measurement instrument, this will normally be an unambiguous indication that the measurement may be erroneous. A system is fault tolerant if it can continue to operate safely in the presence of faults. This is more difficult to achieve, and unless very high availability is required (as might be necessary in a process control system) fail safety is the more appropriate design aim. Both fail safety and fault tolerance can complicate the design and increase the resources and processing needed by the software (in critical systems in general, over 80% of the code may be used to deal with faults rather than normal operation). They must be built in from the earliest stages of design rather than patched into the final code.

There are a number of techniques that can be used to achieve fail safety or fault tolerance in software.

- Build in a diverse algorithm for key calculations. For fail-safety, this does not have to be as accurate as the main algorithm, but can be used to set bounds within which the accurate calculation should fall.
- Implement key calculations in two or more programming languages. This will give protection from errors in interpreters and compilers for individual languages. For

example, if your software is implemented in LabVIEW, write a module in C or C++ to repeat important calculations.

- Use an interval arithmetic approach. This involves carrying two or three sets of values through critical calculations: the real data and data perturbed slightly from the real data. Comparing these values at the end will identify software and hardware problems due to numerical instability, singularities, etc.
- Move critical software to special hardware interfaced to the T&M software, such as a Programmable Logic Controller (PLC) or a processor on a plug-in card (as was suggested to overcome timing problems in [Section 2.4.2](#)). This provides segregation of critical functions, diversity, and increased robustness.

The details of how errors are handled (and the consequent impact on the complexity of the code) depend on the support provided by the programming language used for the implementation. This is discussed in more detail in the language-specific parts of [Section 3](#).

See [\[24\]](#) for more details and a list of further references. [\[26\]](#) also contains a good deal of relevant material in Part 3 Section 7.4.3 and the related tables and definitions.

## 2.5 Coding

### 2.5.1 Introduction

A major concern for code development is to satisfy the design intent and ensure that the representation of the program is easy to understand and modify by the software owners (i.e. the design, development, maintenance and management teams). Some estimates of maintenance costs as a proportion of total development effort are in the region of 80%.

It is useful to remember that every piece of advice concerning code design is there for the benefit of its human developers; the underlying computing platform has no requirements, say, for a modular design or easy-to-read code. In fact, purely from a system perspective, tightly coupled, dense code will typically run faster than code optimised for a human reader.

Thus it follows that the underlying code for T&M software not only implements the software and control functionality, but that it also:

- *acts as a repository of design rationale*—Comments and descriptions allow subsequent developers and reviewers to understand the intention of the software and the assumptions made in its development.
- *acts as an aide-memoire for the software developer*—As modules and the software architecture evolve, it should not be necessary for the developer to retain in his/her working memory the precise functioning of each module. An appropriate level of information hiding in separate modules or components means that the top level design is more easily understood; the details can be explored if necessary.
- *delineates the delegation of responsibilities for product design*—For larger software projects the software architecture can be seen as a way of describing how

responsibility for software services is to be divided. Thus ways of implementing a service (e.g. implementing Fast Fourier Transforms) can be delegated to separate development agencies (for example, third party component providers, other team members and so forth), and subsequently reintegrated without huge disruption to the overall product.

Most modern integrated software development environments (IDEs) provide facilities to help the user write readable code, and navigation facilities to browse the emerging software structure. Examples of this include syntax-based text coloration, drag and drop interface design, online help and so on. Much of the software project is now managed by the development tool itself. Due to the syntactic checking of development environments such as Visual Basic and LabVIEW, the scope for coding errors is now largely confined to semantic errors. However, the rich editing environments also allow developers to write code that is quite impenetrable to a new user, and to build user interfaces with almost any kind of behaviour. The importance of code that is easy to read and understand is accepted in standard software development processes.

Developers can improve the comprehensibility of their code by thinking about how the code will be read and understood and adopting coding standards where appropriate.

### *2.5.2 Coding standards*

Most languages have a community of users who evolve simple coding standards to support software developers using those languages. These coding standards are usually a set of conventions and good practices that allow developers to infer more easily the behaviour of code by reading the code itself. In some sectors (e.g. safety critical), these standards may go as far as to proscribe certain coding practices. Specific guidance for particular languages is given in Part 3 of this guide.

It should be noted that coding standards imply a cognitive overhead, especially when first introduced, as they require the developer to remember a set of practices, conventions and restrictions in their software design. The longer-term benefits are code that is easier to understand, review and modify.

Some aspects of coding standards are obviously language-dependent, and we give some specific standards in Part 3. Some general rules you should follow are given below.

- *Avoid ambiguous language features*—If you are using a language for which several compilers are available, avoid the use of features of the language that are interpreted differently by different compilers.
- *Avoid complex language features*—Avoid complex or obscure language features if possible, as they will be particularly error-prone.
- *Keep program units small*—Keep program modules small, preferably so that each module fits onto a single A4 page or a single computer screen. However, do not use modules that are so small that the data flow between them becomes complicated.
- *Avoid global variables*—Do not use global variables and data unless it is essential.

- *Write readable programs*—Make your programs readable by using upper and lower case letters, meaningful identifiers, blank lines, white space and indentation.
- *Comment your code*—Use enough comments to make your code understandable by someone else. The code for the entry control system illustrates a reasonable level of comments (see [Appendix C](#) and [Appendix D](#)). See also [Section 2.5.3](#) for further details on commenting and coding documentation.
- *Write defensive code*—Use the principles of *defensive programming*, i.e. make your code as robust to errors, unexpected inputs, etc., as possible. In particular, make sure that sensor and actuator errors propagate to the user interface so that the user can see that a problem has occurred.
- *Reuse code where you can*—Developing libraries of T&M software components will improve the maintainability of your software and will also improve integrity, as the components will be used and debugged over a range of applications (see also [Section 2.10](#)). Also make use of the extensive libraries and on-line resources maintained by the suppliers of T&M software languages (see [Appendix A.2](#) for some of these).
- *Minimise use of low-level language*—If you have to use a low-level language, or even assembler language, to operate a particular interface, keep the amount as small as possible. Write it as much like a high-level language as possible; e.g., if you are using assembler, construct control structures such as *for* and *while* loop, and use them in preference to jumps.

You may also wish to define standard interfaces to external libraries in your coding standards: see [Section 2.10.2](#).

### [2.5.3 Coding and software documentation](#)

#### *Documentation styles*

Good code documentation is essential for assuring the quality of code and verifying its function, and ensuring its maintainability.

Useful things to document for each procedure or subroutine include:

- *Function*—A short description/summary of the intent of the procedure or function.
- *Assumes*—Things that are assumed to be set or true when the procedure is called.
- *History*—A brief description of how the procedure has been modified as the program has developed.
- *Inputs*—Short description of the input variables and their datatypes.
- *Returns*—The datatypes of the output(s) of the procedure (if any).

- *Issues*—A list of known limitations and outstanding issues for the procedure.

Some documentation should be included with the code (code comments) and other parts produced as a separate document. Advantages and disadvantages of these two approaches are given in [Table 1](#).

	<b>Advantages</b>	<b>Disadvantages</b>
<b>In-line comments</b>	In-line comments can be easily modified and updated as the code develops so that the documentation is always in step with the code	The dispersal of the documentation over the collection of software files means that there may be no one document that contains the documentation for the code. To mitigate against this a top-level description should also be maintained
<b>Separate document</b>	<p>A separate document provides a single point of access for the documentation for the code.</p> <p>Does not require users to have development environment to view the documentation.</p> <p>Can be more discursive and can contain other documentation that is more appropriately stored in separate documentation, such as installation instructions, system requirements, other global considerations such as hardware set-up issues</p>	<p>More effort is needed during development to keep documentation up to date.</p> <p>The documentation can get separated from original code, resulting in out-of-date documentation</p>

**Table 1: Comparison of documentation styles**

In practice a mixture of both is probably the best way forward. Separate documents should provide an abstraction of the code, explaining the algorithms and design of the software, and describing behaviour that is only evident because of the environment (e.g. interrupt service routines, code to stop race conditions, etc.). In-line comments describe the more local, detailed behaviour of the code.

On a practical note, documentation is so important that even if you do not create it as the code is written (for example, if in a burst of creativity you create a whole block of code in one sitting), you should go back and do the documentation later. Your documentation should be finished before the point where it is needed as an input to an activity, so for example test specifications must be completed before testing begins, design documents must be finished before design

reviews, and user documentation (user guide, installation requirements and so forth) must be delivered with the software.

If you are developing a component for other developers to build on (as a subroutine or ActiveX control) it is good practice to create a couple of examples to show how the component is to be used (e.g. in the most typical development languages). This helps acceptance as users can more easily see if your component addresses their requirements. See also [Section 2.10.2](#).

### *Code layout and commenting*

Some basic advice on code design and layout is given below:

- *Code layout and white space*—The use of white space has long been acknowledged as an important component in both standard software development and graphical design. White space helps the reader “parse” the code or representation into meaningful chunks, which are then investigated further according to the goals of the user. In text-based languages such as Basic and C you should use nesting and indentation to show the logical structures in the code, where possible. This allows readers to skip into, or over, data and control structures. In graphical languages you can use layout to show overall program flow and structures.
- *Commenting*—All code should be well documented and commented. Comments should explain to the reader what is happening at key points in the code (including unusual design decisions) but not just transliterate the code. Each main component of the software (e.g. function, procedure, user interface) should also have a summary at the start explaining the overall functionality of that aspect of the program.
- *Proximity relations*—These are often interpreted by readers as implying an underlying logical or temporal relationship. This applies primarily to graphical languages such as LabVIEW. Thus graphical components that are close together will by default be interpreted as being related in some way. This can be further reinforced in user interfaces by the judicious use of boxes or frames around related objects.
- *Standard reading stereotypes*—In countries where the written language is historically European-derived, text and graphical representations have a “default” reading of top-left to bottom-right, which reflects the mode of reading texts that is learnt in early educational experiences. This means that user interfaces and graphical representations will receive a standard reading in the absence of knowledge of the representation or program’s functionality. Some graphical languages (e.g. G for LabVIEW) allow developers to create code that violates this default reading behaviour. For this reason it is generally recommended that graphical language representations be broadly structured top-left to bottom-right, where appropriate.
- *Give variables and functions meaningful names*—If you take care to give your variables and functions meaningful names, you can develop code that almost reads as an English narrative. Thus:

```
If CalibrationIsComplete() Then  
    WriteDataToFile(FileNamePath, TimePeriod)
```

is to be preferred to:

```
If calcompl() Then  
    wrdata(f, t)
```

- *Other expectations and reading behaviour*—Other standard interpretations of graphical languages and interfaces will depend on domain experience and exposure to different representations and their modelling clichés. For example, users familiar with windows-based environments will easily recognise the behaviour of menus, check-boxes and push-buttons in any windows interface. Other visual representation stereotypes include the use of lines between nodes to denote causal connections, data dependencies (e.g. in dataflow diagrams), and temporal relationships (items on left generally happen before those on right).

## 2.6 Verification and validation (V&V)

Verification is the process of checking that the code accurately implements the design (colloquially, that it “does the thing right”), and validation is the process of checking that it meets the user’s requirements (that it “does the right thing”).

V&V can be carried out by testing, and by a number of techniques that do not involve executing the software, including reviews and static analysis. We recommend that you supplement testing with at least one of these other techniques.

### 2.6.1 Testing

It is important to realise the limitations of testing as a method for achieving software reliability. In most T&M software, there are so many combinations of inputs, outputs and internal states that it would be completely impractical to test them all. This means that at the point you start using your software for real measurements, only a small proportion of the possible tests will have been carried out.

Testing is a huge subject, and for further information see [23]. Below we give an overview of “formal” testing, by which we mean testing that is defined by a written test specification and test execution, and which should be carried out for all T&M software where reliance is to be placed on the results. In addition, you may wish to carry out informal testing at the module level to give confidence that the integrated software will function correctly.

#### *Test specification*

The first stage of testing is to develop a *test specification*. This is a document that lists all the tests that are to be carried out, and gives the expected answers. It is highly desirable that the majority of the tests are written by someone other than the developer. Again, this is to avoid “mind-lock”, where the developer writes tests to show that what they intend does happen, but what they intend does not actually meet the requirements.

The test specification should include the following:

- “Realistic” tests that represent the likely values to be encountered when using the T&M software.

- Boundary tests, that involve values just inside or just outside the specified limits for each input. Include special cases such as empty arrays, empty strings, and zero where normally a positive integer will be used (e.g. for “number of scans”).
- Unusual combinations of inputs, including physically unlikely input values.
- Error handling. These include negative values where positive is expected (e.g for “number of scans”), out-of-range inputs, missing files and bad path names, and tests that cause modules or functions within the T&M software to return an error.
- User interface tests. These should include cancelling dialogue boxes, pressing inappropriate buttons, aborting the T&M software when it is running, and random typing at the keyboard. See also [Section 2.13](#) on user interface design issues.
- Stress tests. These test the T&M software under extreme operating conditions. They should include exposing the software to maximum data rates, writing large disk files, operating the software with several other applications running, etc., as appropriate.

Each test should be annotated with which items in the requirements description it checks.

An example of a test specification for the barrier control example is shown in [Figure 11](#).



**Test no. 11**

Relevant requirement: 2.1

Test criteria: The test is passed when all expected results are observed.

Test set-up: Barrier emulator attached to instrument interface.

Pre-conditions: 1) The vehicle sensor interface reads False (no vehicle). 2) Barrier close signal to interface.

Step no.	Action	Expected result
1	Press Open button	Barrier raise signal to interface
2	Send True signal to vehicle sensor interface from emulator	Vehicle present panel light illuminates
3	Press Close button	No change to barrier interface
4	Send False signal to vehicle sensor interface from emulator	1. Vehicle present panel light extinguished 2. Barrier close signal to interface

**Figure 11: Test specification**

Where the development is being carried out for a customer, it is usual to include a set of *acceptance tests* as the basis for formal acceptance of the T&M software. These are often based on measurement scenarios and will be devised to “sell off” the requirements in the contract or other procurement document. The acceptance tests can be written by the customer, or by the developer and agreed with the customer.

For the majority of T&M software developments, it will be sufficient to carry out the majority of formal testing at the “black-box” level, i.e. without looking inside the software. However, you should also test modules or subroutines implementing complex calculations, e.g. FFTs.



If the T&M software is at the higher measurement software levels, it will be necessary to carry out *structural testing* to aim to achieve a specified level of *test coverage*. This is a measure of the proportion of the program that the testing has executed. Test coverage is described in terms of the proportion of the “objects” that make up the software that are to be tested; not surprisingly, the higher the proportion of objects that are tested, the better the reliability of the software is found to be in service, all other things being equal. Test coverage can be expressed in terms of statement coverage (the number of source code statements covered), branch coverage (the number of branches covered following a conditional statement), equivalence partition testing (functional testing at the module level), etc. It can be surprisingly hard to achieve high levels of coverage in software containing code to cover rarely encountered

situations. A realistic goal for simple T&M software is 100% statement coverage and 50% branch coverage. For complex T&M software, it may be difficult to exceed 90% statement coverage.



Also for T&M software at the higher measurement software levels, you may be able to use *reference test sets*. These are very high quality sets of tests and expected results. Data sets may be available concentrating on boundaries and other known areas of difficulty. A difficulty in using them for T&M software testing is that they are aimed at specific measurement problems and algorithms and are not general enough to apply to much of the software's functionality. More information is contained in [38].



You could also consider carrying out some *statistical tests*. These are tests, selected to be representative of the normal operation of the T&M software, that give a statistical estimate of the reliability of the software in use. Roughly speaking, if you carry out  $2.3n$  test measurements without failure, there is a 90% confidence that the mean time to failure of the software is at least  $n$  measurements. One way of generating enough tests to give a high level of assurance is to generate test values randomly within the expected usage profile. More information on statistical testing is contained in [26].

### *Test execution*

The first stage of testing is normally to exercise the T&M software isolated from the hardware, by writing software test modules that mimic the behaviour of the hardware and record or display the data sent to them. (This is the approach we have taken with the worked examples, where an asynchronous procedure is used to model the behaviour of a car approaching the barrier.) These test modules can be substituted for the normal driver modules during testing, or they can be built into the software and data can be directed to and from them by setting a suitable “test” variable. Data for these test modules to send to the software can be stored in a special disk file or, if there are not too many test cases, in the “.ini” file for the software.

User input can be by hand, or from another program that allows test data to be recorded and played back. This can either be a general-purpose test program (see [27] for examples) or a specially-written program.

Test results can be checked by hand, by entering the expected results into the test scripts for an automated test program, or by running the tests on diverse software and comparing the results (with allowance for small variations in numerical results)—this is known as “back-to-back” testing. One of these diverse programs could, for example, be an earlier version of the T&M software written using a different language.

The test results should be documented, for example by producing a new version of the test specification with the actual results next to the expected ones.

Keep the test programs and test data sets under configuration control, as they can be used for re-testing the software if changes are made, as discussed in [Section 2.7](#).



If there are test coverage targets, it will be necessary to run the tests on a more complex test harness that measures the statements, branches etc. covered. Such a test harness will slow the software down and may give problems in hard real-time situations.

### 2.6.2 Code reviews

Code reviews should be carried out in much the same way as design reviews (see [Section 2.4.4](#)). Topics to examine include:

- Does the code correctly implement the design?
- Have the appropriate mechanical checks been carried out?
- Is the code clear and easy to understand?
- Is the code adequately commented (see [Section 2.5.3](#))?
- Is it easy to relate the code to the design?
- Does the program have sufficient capacity and performance?
- Are the agreed standards adequate, and being followed?
- Is the configuration item properly identified and traceable?

### 2.6.3 Static analysis



The term *static analysis* is normally used to refer to analysis of programs, without executing them, by special software tools. Static analysis tools are only available for some T&M software languages and are recommended for T&M software at the highest measurement software levels (although they can be beneficially used at lower levels to detect faults earlier in the lifecycle than testing). Compilers and interpreters carry out some static checking, particularly for consistency of data types. Additional tools may be used for:

- checking conformance to coding standards
- checking data use through programs, by examining the sequence in which variables are read from and written to in order to detect any anomalous usage (e.g. variables that are read from before they are written to)
- checking control flow through programs (e.g. identifying multiple entries into loops, sections of code from which there is no exit, or unreachable code)
- checking information flow, to identify dependencies between module inputs and outputs, in order to check that these are as defined and that there are no unexpected dependencies
- checking performance, by analysis of worst case conditions to ensure timing, accuracy and capacity requirements are met



Static analysis also includes *semantic analysis*, which considers whether the code has the intended meaning. Semantic analysis is an example at the code level of the use of formal

methods (see the section on formal methods under [Section 2.4.3](#)). There are two broad approaches. The first considers whether the code has a meaning that is definitely not intended, such as overflowing array bounds, dividing by zero, and so on. (The kind of behaviour that will cause a run-time exception if it occurs in use.) This requires only the code, and for some languages tools are available that will carry out the analysis automatically, at least to the point of highlighting places in the code that need further consideration.

The second approach considers whether the code has the intended meaning, as expressed by a separate design or specification. This could be provided by the formal specifications described in [Section 2.4.3](#), or may be developed separately, typically at a function level. Again, there are tools to support the comparison of the specification with the meaning of the code.

#### 2.6.4 System tests

You should also plan for and execute the normal type of end-to-end system tests that you would do on a conventional instrument, using reference data and specimens, etc.

## 2.7 Maintenance

During the life of the T&M software, it is almost inevitable that you will want to make some changes. These may be *corrective*, because a fault has been discovered, *perfective*, to improve some aspects of the software's performance while still meeting the original requirements, or *adaptive*, to change it to do a different job.

The maintenance of software always involves a design change to the program or to its data. Maintenance therefore requires reporting of problems, diagnosis of the cause, implementation of a correction to the design, testing, and installation of the correction on the software in the field. The relevant parts of the T&M software's documentation should be revised in line with the change. It is a good idea to keep a special set of tests, known as *regression tests*, for checking revised software. Regression tests should include all or most of the original tests, and be augmented with tests illustrating problems that have occurred. Before the modified software is released for use, it should have passed the regression tests.

Before you make a change, you should evaluate the need for it and its impact. (This sort of *change control* is a mandatory part of formal quality systems.) Generally, this evaluation should be done by means of a review by developers and users, which should consider:

- the importance of the change (does it impact critical results, or is it cosmetic?)
- the direct impact of the change (does it affect many modules or is it local? is it easy to implement or does it involve tricky real-time aspects?)
- the indirect impact of the change (will error logging still work? will failure recovery still work?)
- the re-verification and re-validation that will be required, including regression testing



If your T&M software is in the higher software criticality levels, you should implement some sort of formal in-service fault reporting system (this is sometimes known as a DRACAS—data recording and corrective action system). This could be based on a simple database to collect

error reports, or on a more elaborate bug tracking system such as Bugzilla and the like (see [Appendix A.9](#)). You should provide a mechanism for users to contact you with problems they have encountered with the software, and devise a procedure for analysing the problems and deciding on what corrective action to apply (as described above), and then reissuing corrected software if necessary. You should also share any lessons learnt with others in your organisation who produce similar T&M instruments (see [Section 2.14](#)).

## 2.8 Configuration management

Configuration management is important for T&M software for two reasons:

- It ensures that a measurement can be related to a particular software version. This is as important for T&M software as recording the serial numbers of instruments.
- It enables the reconstruction of previous configurations of the software. This “re-hydration” of previous versions may be necessary to answer a customer query, where the customer is using a previous version, or to roll back to a previous version during the software development if a development path proves problematic.

Users must be able to determine the version of the software they are using. This is best done by placing it on the T&M instrument’s front panel or by adding an item to the Help menu. The software version number should also be printed on all reports and certificates generated by the software.

The basic principal of configuration control for small software projects is to create “safe islands” (libraries), which record a snapshot of the software at some key point in its development. This may be at key review stages, incremental versions, beta releases, released versions and so forth. These libraries are created by saving the software files in a secure repository and recording enough contextual information to allow the correct version to be re-hydrated at some later date. Each should be uniquely identified by a version number or letter, and should have an associated *configuration record* that identifies all the constituent items in the version, and gives a brief summary of the key aspects.

If there is a development team, there needs to be an identified librarian for the library, whose responsibility is to check out files to the developers, ensuring that only one developer is making changes to each file at any one time, and to check in modified files.

The library should be backed-up daily, for example to a floppy disk, Zip drive, tape drive, CD-ROM, or another networked computer. If the development is important from a safety or business point of view, the backup should be protected from fire and theft.

Small software development projects (say a few screens of code) can easily be managed manually. For larger projects you may wish to employ a custom configuration management tool. For text-based languages the following tools are available:

- Microsoft Visual SourceSafe. See <http://msdn.microsoft.com/ssafe/>
- CVS—the most widely used configuration management tool in the free software community. See <http://www.cyclic.com/>

- MKS Source Integrity™ 7.4—source code configuration management tool from MKS. See <http://www.mks.com/>
- Clear Case from Rational. See <http://www.rational.com/>

## 2.9 Metrics



For more critical developments, you may wish to collect some metrics (measures) of the T&M software. Metrics are best used comparatively, for instance to compare project progress to previous, similar T&M software projects, or to identify error-prone modules within the software. Metrics can be used:

- *To measure fault density*—By far the most common measure is the number of residual faults per thousand lines of code (kloc), although this figure depends on the interpretations of “line of code” and “fault”, and the point in the lifecycle at which it is measured.
- *To measure size or complexity of software*—There are various measures of software complexity, many of which are supported by tools, although these have to be used with care. The most effective uses for these kinds of metrics are to identify problem areas (which have anomalously high complexity measures), and to estimate development or maintenance effort.

Metrics can also be used to measure the software development process rather than the software itself, e.g. to assess the degree of control and repeatability provided by the process.

See [18] for more details on all these types of metric.

## 2.10 Software and component reuse

The aim of this section is to provide guidance on the reuse of code and software components in T&M software. The specific issue of the assurance of software components is addressed in [Section 2.12.4](#).

Although a separate best practice guide on software reuse (METROS) is being maintained under the SSfM programme [42], we present an overall perspective as it applies to T&M software development.

There are two common approaches to software reuse for software applications:

- software libraries and in-line code reuse
- stand-alone software components

[Section 2.11](#) contains guidance on mixed language programming, which may be relevant when using libraries written in another language (e.g. the NAG Fortran libraries).

### 2.10.1 Software libraries and in-line code reuse

In this subsection, we consider the use of existing source code in T&M software.

As T&M software developers write code, useful routines can be abstracted and re-used in other modules or programs. When other programs are developed, these functions can be integrated into the project and saved as part of the project files.

Since the developer has access to the underlying source code, the software functions and procedures can be assured as part of the whole application. Any limitations in the software can be overcome by reworking parts of the software and tailoring it to the requirements of the application.

Some drawbacks of this approach are that the reuse is limited to the language (and generally the platform) on which the library is based. Moreover there is a risk that subtle data manipulation algorithms might be modified by users not familiar with the issues involved. You should also check that any assumptions about the way that the library is to be used still hold.

Also third-party software developers have historically not favoured this approach as it exposes some commercial risks of distributing the underlying intellectual property in the form of the underlying source code. (In languages where source libraries are the norm, this can be addressed by code obfuscation.) In-line software reuse within an organisation is more common due to the freer movement of intellectual property.

Any software that is reused should be well-documented (see [Section 2.5.3](#)) to allow efficient evaluation by potential users as to its applicability and interface requirements.

### 2.10.2 Reusable software components

Software components are discrete bundles of functionality with standard defined interfaces that hide much of the internal functionality. Often these components make assumptions about an application or “middleware” layer, which provides the framework in which their interfaces can be exposed; in short the component simply “plugs-in” to the development environment.

Most modern development environments have facilities for incorporating reusable software components into software.

The most obvious example of this historically has been the use of standard device drivers to allow high level code to interface to a hardware component, such as a voltmeter. Modern examples include COM components in a Windows 32-bit architecture and sub-VIs in LabVIEW, for example to implement:

- *Interface components*—These provide graph controls, slider bars, push buttons and so on.
- *Interface-less components*—These provide a simple or standard developer interface to some underlying functionality. Components or T&M software that interface to databases, or provide data manipulation services, are examples of these.

Another example is the NAG libraries, examples of whose uses are shown in [Appendix E](#).



Another implication of the lack of source code is that the end-user is limited to methods and properties supplied by the component developer.

#### *Developing software components*

If you aim to develop a software component for other developers to use, you will need to develop a usable API (Application Programming Interface) for your component, and support the assurance activities for the software component by its end users.

A usable API is based on concise documentation and a coherent underlying object model to support other developers in determining how to interface to your component, how to distribute it, what errors to expect and so forth.

In designing the interface to the component you will need to handle unusual combinations of inputs through a good error model. Some effort is therefore needed in making a component more generic than may have been necessary in its original application.

#### *Procuring software components*

As a evaluator of available software components, you need to build a picture of the suitability of a component to your application's specific requirements, to assure the quality of any external software, and support the users and developers of the software.

The following issues can be used to form the basis of a judgement:

- *Reliability of the component*—This is discussed in [Section 2.12.4](#).
- *Dependencies on other components*—What other libraries or components are required to use the component, and what versions of these libraries are needed?
- *Distribution aspects*—Are there any licensing issues? For example some components are not royalty-free for subsequent application distribution (although this is becoming less common).
- *Availability of source code*—For long term availability, and modification.
- *Cost of component*.
- *Ease of use and evaluation*—What is the quality of documentation? Are there any sample applications to illustrate how it can be used? Are there evaluation versions of the software for building a prototype? Evaluation should be against a defined set of criteria based on the requirements for the T&M software.
- *Side effects*—Are there any unwanted/unexpected side-effects of the software? An example is an interface control that behaves in an unexpected way—see the discussion on twiddle knobs and modal dialogues in [Section 2.13.2](#).



## 2.11 Mixed language programming

In many cases, the implementation language of a standalone software component is incidental: it is merely the language in which the desired functionality happens to have been implemented. In other cases, the use of a second language has specific benefits. It may, for example, provide easier access to the underlying hardware. Device drivers are often written in C or C++ for this reason. It may also offer higher performance for intensive computations through being compiled rather than interpreted.

The problems that arise in mixed language programming depend on how tightly the interface between the languages is defined. There will always be issues of parameter and result types. For the basic types (integers and floating point), the languages are likely to use different names for the same underlying hardware type, and the problem is just one of determining the corresponding names used. The main pitfall here is that these basic types may come in variants with different sizes, and the same variant must be used on each side. Even at this level, there may be no obvious correspondence in some cases: for example, many languages assume that integers are signed, but C and C++ also support unsigned integers.

Data structures (strings, records, arrays, and objects) cause far more problems, because each language makes its own decisions about how the components will be laid out in storage. For example, arrays may be indexed starting at 0, 1, or a program-defined lower bound. Two-dimensional arrays may be represented with either elements in the same row being adjacent in store or elements in the same column being adjacent in store. (Most languages adopt the former convention, but Visual Basic and Fortran use the latter.) The order of components in structures is not necessarily the order in which they appear in the source, and may not even be defined by the language.

Calling conventions can also differ. Parameters are passed on a stack in most language implementations, but they can stack parameters in left-to-right or right-to-left order, and can give the job of unstacking parameters on return to the caller or the callee. Languages can also pass either the current value of the parameter (which may be copied back to the parameter variable on exit) or a pointer to the variable that allows it to be updated directly.

Error handling may also be very different in the two languages. The only language independent technique is to return an error indication through a parameter or the result of a function.

There may be special naming conventions to be considered in calling external code. Where identifiers are case sensitive in the called language, the name must be reproduced exactly in the calling language. (This may cause problems if that language is not case sensitive.) Where the called language is not case sensitive, it may standardise the case used internally, and this may be reflected in the name exported to the caller. Some systems modify the names of interface symbols (by introducing a leading underscore, for example) to reduce the chances of a name clash.

When using a mixed language approach, for whatever reason, you should therefore: be sure that you understand the details of the procedure calling interface that will be used, and any annotations that need to be made to procedure declarations or calls in your program to take account of this. It may be worth developing a simple program to test any mixed language interfaces that are to be used before committing the design of the full system to their use.

If developing the library to be called as well as the calling code, consider the range of potential calling languages to be supported and:

- Restrict the parameter types to those that have a common representation on each side. Where possible, use only basic types rather than structures.
- Use appropriate methods of indicating errors.
- Document the procedure names that the caller will have to use (which are not necessarily those in the source).
- Provide examples of the library use in all the planned calling languages. Where those languages require declarations of external procedures, provide appropriate collections of definitions.

## 2.12 T&M software assurance

Where measurements are being made and the resultant data sets manipulated by computer technology, it is important that you have a healthy scepticism of the results and awareness of the ways in which data and algorithms can be in error or misleading. The relationship between user trust in software and its validity is a complex one, but users can misjudge the actual integrity of a T&M software in both directions. Gullibility errors can occur when, for example, a user interface presents bad data in a seemingly authoritative manner. Incredulity errors are perhaps less obvious, but can occur, for example, when the perception of a credible product is unduly influenced by an unreliable beta version. Interfaces with low usability will tend to reduce users' perception of credibility even if the underlying data and computation are valid.

The Institute of Chemical Engineers publishes a set of guidelines [20] that remind engineers of their responsibilities in decisions based on computing technology, and provide practical advice on common classes of error that can exist in the software and organisational processes, and we recommend that you obtain a copy.

The remainder of this section describes a systematic approach to the assurance of T&M software.

### 2.12.1 Overview

Much T&M software is used in applications where errors can have adverse commercial or even safety consequences. As a result, users will need assurance that the likelihood of errors has been controlled. In some cases, this assurance will be formally examined by a regulatory authority. Very early in the development of the software, you should address the extent of the assurance needed and how it will be produced. This involves:

- carrying out a risk assessment ([Section 2.12.2](#))
- designing a process for assuring the application software in a way appropriate to the risk ([Section 2.12.3](#))
- identifying the means for assuring any off-the-shelf software packages used in the development (including development environment and operating system) to the identified level ([Section 2.12.4](#))



For small, non-critical T&M software, it is sufficient to consider assurance informally. However, for larger T&M software, and certainly for safety-related software or software in the higher measurement software levels, we recommend that you produce a *Dependability Case* for the software. A dependability case presents a convincing and valid argument, based on evidence, that a system is adequately dependable for a given application in a given environment. The idea generalises that of a safety case [36] to cover issues not directly concerned with safety.

For T&M software, the dependability case could be quite short, but should justify the allocation of level and the choice of development process, including the use of software packages, and provide evidence that any numerical dependability targets have been met. You should also make sure you are aware of any sector-specific standards you should be applying.

### 2.12.2 Risk assessment and mitigation

The first step in addressing assurance is to establish the possible consequences of software failure. These may be loss of business revenue due to a process shutdown or the need to recall out-of-specification components, personal injury or loss of life, or environmental damage from escaping materials. The more severe consequences are less tolerable, and will require stronger assurance that they do not occur.

Where an instrument is being developed in isolation, it will not be possible to judge these application-level consequences. Typical consequences at the instrument level are that it will fail to complete a measurement, or that the results will be outside some tolerance, and you should attempt to characterise and document the frequencies of such failures. The instrument user can then decide if this is adequate for their particular application.

The consequences of software failure can often be mitigated by relatively simple hardware or procedural means. For example:

- An emergency stop button can allow the user to force the system to a safe state in the event of failure.
- A hardwired interlock can prevent the software from commanding an operation in a dangerous situation.
- A toughened glass screen, remote operation, or a remote viewing camera can protect the users from an otherwise dangerous failure.
- A “sanity check” performed by the user can detect errors in the calculation of a result. This involves designing the user interface to prompt the user to do a rough calculation to check the result. The user interface will need to provide suitable intermediate results. If the instrument produces a certificate, a space can be provided for the rough calculation. See also [Section 2.13.2](#).

These external mitigations can reduce the level of assurance required for the software and hence are an important part of the design of the application or instrument as a whole. Remember, though, that safety procedures are not always carried out properly, either deliberately or through omission, and so hardware safety features are to be preferred when they are practicable.

The end result of the risk assessment will be a judgement of the criticality of the software, or, in more complex and critical applications, a numeric target for the failure rate of the software as a

whole or its individual functions. For example, in an approach designed by NPL (see [Appendix A.9](#)), software is judged as

- Not critical, if it has a negligible effect on overall result, and there is no danger of loss of business income or reputation.
- Significantly critical, if it may result in incorrect results but these will definitely be spotted, or there is potential for loss of income or reputation.
- Substantially critical, if it may result in seriously incorrect results or errors may not be spotted, or it is likely to lead to loss of income or reputation if faulty.
- Life critical, if it may result in personal injury or loss of life.

The SSfM Best Practice Guide No. 1: *Measurement System Validation* [38] assesses criticality similarly, on a scale from 1 to 4, while IEC 61508 [26] assigns a *Safety Integrity Level* from 1 to 4 based on the tolerable failure rates.

### 2.12.3 Process design

The development process must be designed to provide assurance commensurate with the consequences of software failure and its *a priori* likelihood of occurrence. The latter depends mainly on the complexity of the program. The NPL approach assesses software as very simple, simple, moderately complex or complex based on the complexity of the functionality and how easy it is to understand, the size of the software, the ease of modification, the degree of control of external systems, and the complexity of the mathematics involved.

The likelihood of failure is increased if the software:

- is difficult to test
- is reliant on key staff, is to be produced by inexperienced staff, and/or is to be produced by a large team
- is to be produced to ambitious timescales
- has ambitious or poorly defined requirements
- incorporates new technology or a novel design

On the other hand, the likelihood of failure is decreased if the software:

- has an alternative means of verification
- is to be produced by experienced staff
- utilises a modular approach

The NPL approach is implemented by a software tool that derives a Software Integrity Level (again from 1 to 4, but not related to the Safety Integrity Level of [26]) that takes these factors

into account. The SSfM Best Practice Guide No. 1 derives a Measurement Software Level (from 1 to 4) in a similar but less algorithmic way.

In each approach, the index level is used to choose an appropriate process. This will include both techniques that have a good likelihood of achieving the MSL or SIL (*e.g.* an appropriate programming language); and techniques to evaluate (at least to some degree) the achieved integrity (*e.g.* testing). For example, the SSfM Best Practice Guide No. 1 makes the following recommendations.

Measurement software level	Technique	Reference in this document
1	Review of specification Mathematical specification Defensive programming Code review Structural testing System testing Numerical reference results	<a href="#">Section 2.3</a> <a href="#">Section 2.3</a> <a href="#">Section 2.5.2</a> <a href="#">Section 2.6.2</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> †
2	Review of specification Software inspection of specification Mathematical specification Static analysis Boundary value analysis Defensive programming Code review Numerical stability Statement testing Statistical testing Boundary value testing Accredited testing System testing Stress testing Numerical reference results Back-to-back testing	<a href="#">Section 2.3</a> † <a href="#">Section 2.3</a> <a href="#">Section 2.6.3</a> † <a href="#">Section 2.5.2</a> <a href="#">Section 2.6.2</a> † <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> ‡ <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> † <a href="#">Section 2.6.1</a>

Measurement software level	Technique	Reference in this document
3	Software inspection of specification Mathematical specification Static analysis Boundary value analysis Numerical stability Verification testing Statistical testing Statement testing Branch testing Boundary value testing Stress testing Back-to-back testing	† <a href="#">Section 2.3</a> <a href="#">Section 2.6.3</a> † † <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a>
4	Mathematical specification Formal specification Static analysis Numerical stability Qualification of microprocessor Verification testing Branch testing Boundary value testing Source code with executable	<a href="#">Section 2.3</a> <a href="#">Section 2.4.3</a> <a href="#">Section 2.6.3</a> † † <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> <a href="#">Section 2.6.1</a> ¶
† Outside the scope of this guide. ‡ This is testing against a validation suite developed by a third party. It is very unlikely that such suites will be available for much T&M software. ¶ This is supply of the source code to users together with the executable program.		

Table 2: Techniques for measurement software levels



Safety-related T&M software should be developed in accordance with IEC 61508 [26]. This makes process recommendations based on the Safety Integrity Level, which is based purely on the assessed criticality of the software. The standard identifies a large number of techniques, which are marked as “not recommended”, “don’t care”, “recommended” or “highly recommended” according to the SIL. Highly recommended techniques should normally be used. However, the assessment of *a priori* risk can be used as a justification for omitting highly recommended techniques where they are inappropriate for the T&M software under consideration. Thus for example a SIL 2 program may be of low *a priori* risk because it is very simple and is to be developed by experienced staff. Although IEC 61508 has static analysis (see [Section 2.6.3](#)) as a highly recommended technique at this SIL, the decision might then be taken to omit this and rely on testing (see [Section 2.6.1](#)).

#### 2.12.4 Assurance of software packages and components

T&M software frequently relies on one or more off-the-shelf software packages, which may include an operating system (e.g. Windows 98 or Windows 2000), a graphical programming language (e.g. LabVIEW), a programming environment (e.g. Visual Basic editor), a compiler (e.g. C compiler), etc. It may also use a variety of other software components, as discussed in [Section 2.10](#). You should give some thought to how you are to justify the software integrity of these packages as well as of your application program.

This section also applies to the assurance of complete third-party T&M software components (e.g. VIs or sub-VIs).

Users should be wary of using third party software especially in critical applications. You should define in your quality system what tests and other forms of assurance are required. Be aware that even well known packages contain undocumented features (see <http://www.eeggs.com/> for a list of some, such as the flight simulator in Microsoft Excel 97).

Generally users do not have access to the underlying source code, either because the component has been developed in a different language, or because the component developer wishes to protect their intellectual investment by not giving away the source code. Therefore assurance has to be based on a combination of trust (e.g. a trusted supplier), first-hand experience (e.g. number of hours of bug-free operating experience) and other evidence (good documentation, wide industry support, good technical support, etc.).

A basic level of assurance for a software package is the following:

- reasonable documentation
- development to an appropriate quality management standard, typically the ISO 9000 series
- pre-release testing using a large test suite, with faults logged and corrected; there should be a target for the number and severity of known faults at release (no software above a few thousand lines of source code is fault free)
- adequate configuration control, so the version number can be established and faults and fixes related to a specific version
- a process for recording and publicising faults found in service
- periodic maintenance releases to fix faults rather than add new features

You can build on this basic level of assurance with:

- information about the package in your organisation, if there are a number of users and faults are logged
- information from an established user base in a similar application area, including a “reference site” (possibly identified by the package supplier) or a successful implementation reported in the published literature

- information about the package in more general use, for instance from the Internet—this may be anecdotal, but the absence of bad reports gives some confidence in the package, whereas repeated reports of problems are a cause for concern

You will find that some packages are particularly problematic, such as some ActiveX components that are poorly documented.

The assurance of the numerical computations in your T&M software will be increased if you use one of the established, high-quality libraries of numerical software, such as the NAG [43], LINPACK [44] or NPL libraries [28].

If you cannot obtain adequate assurance from the software packages you are using, and you cannot switch to others, you will have to add design features to mitigate potential problems, as discussed in [Section 2.4.5](#).

A detailed discussion of the assurance of packages and components is contained in [31] and [32].



IEC61508 [26] imposes specific requirements for the validation of packages and components in safety-related software.

## 2.13 Human factors in T&M software development

### 2.13.1 General human factors issues

In this section we address a number of important human factors issues arising in the development of T&M software. General advice on human factors in measurement and calibration can be found in [37]. Human factors impact two particular classes of T&M software user:

- *Design, development and maintenance roles*—These include other developers, maintainers, testers, reviewers, software managers and so forth, typically in the same organisation or department.
- *End users*—Usable software is critical for customer satisfaction and acceptance. Usability is not just restricted to a well-designed graphical interface (if applicable), but should also consider the interaction logic (how will the user expect the system to behave) and integration with other desktop systems (e.g. spreadsheets etc).

Some general human factors points that can be borne in mind in systems development are:

- *Limitations of working memory*—Cognitive research has demonstrated that there is a limit to how much information can be held at any time in a person's working memory. Although the exact amount is a matter for debate, some general evidence suggests about seven items<sup>1</sup> at any time. This can be somewhat increased by chunking (i.e. grouping together related items in one "chunk"). What this means is

---

<sup>1</sup> Consider how long a telephone number can easily be recalled through simple repetition alone.



that you should not expect your users to remember large amounts of information (such as initialisation settings, the behaviour of a complex procedure, and so forth).

- *Common error classes*—A useful distinction is made between two common classes of error, namely: (a) slips and trips (a correct plan incorrectly executed); and (b) mistakes (where incorrect actions are initiated based on an incorrect user goal, underlying user model, or plan of what should be done). The most common slips and trips can quite easily be caught by a well-designed interface that validates user input or performs basic syntax checking on code as it is written. Mistakes are harder to prevent, and require a transparent user model that allows a correct mental model to be built, and relevant labelling and contextual information (e.g. in the form of meaningful messages and labels, hypertext help on the interface, and so forth).

These general human factors issues should be borne in mind when designing and documenting T&M software, and are one of the motivations for the guidance on requirements description, design and coding ([Section 2.3](#)–[Section 2.5](#)) above. Even if there is only one developer (i.e. the “developer-user”), well-designed and documented code is much easier to revisit after a period of absence.

Human factors in the design of the T&M human computer interface are discussed next.

### *2.13.2 Human Computer Interface (HCI) design*

#### *Introduction*

Designing a user interface for the T&M instrument is an important aspect of T&M software design—especially where the T&M instrument is to be used by someone who is not the developer. A good user interface should not intrude and get in the way of efficient use of the instrument; at best it should be “invisible” in as much as it provides a simple and intuitive interface to the instrument.

Most development environments have powerful facilities for building graphical interfaces. Generally the temptation to produce a “flashy” interface should be avoided; simple layout, standard controls, colours and fonts are preferred. Have a thought for the end user of the instrument.

Usability is based on a clear understanding of how the end user will expect the instrument to behave (in terms of the instrument behaviour and functionality) coupled with a simple approach to interface design. Remember that 90% of the time the user will be using other pieces of software, and so your interface should build on (and not subvert) their general expectations of how software interfaces should behave.

There are a number of risks that can occur from poor interface design:

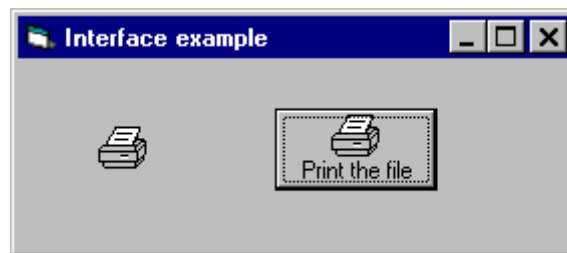
- T&M software often controls sensitive instruments with a physical interface to the real world—an out of range value may damage measurement hardware and samples. Your user interface should expect the user to enter out-of-range and meaningless values.

- T&M software that presents data in a misleading way can lead the user to make incorrect inferences about the measurement itself. For example, a value simply presented as a number may be misinterpreted without a proper indication of its units.
- An instrument that is hard to understand and control will result in user frustration and reduced perceived credibility of the instrument and its supplier.

There are plenty of resources for HCI design. It is not our intention to replicate existing material here. However, some summary points are briefly discussed below. A selection of Internet references concerning interface design can be found in [Appendix A.1](#).

#### *General rules for usability*

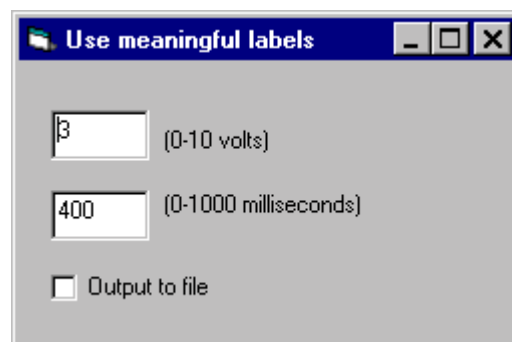
Make use of user stereotypes of computer interfaces where possible. Remember that 90% of the time users will be using other applications rather than your T&M software. They will have certain expectations of how many interface controls will work. Therefore you should use standard controls where possible, and adhere to standard user interface designs and conventions (e.g. in terms of order of menu items—file, edit and so forth—and positions and names of buttons). In the following example, users may not realise that they can click the image of the printer to print. A standard button is much more inviting.



**Figure 12: Use standard controls to invite standard user behaviours**

Group controls logically according to their function and relationship to the underlying logic of the instrument.

Add labels to the interface to indicate the function and meaning of information that users are entering or interpreting. This should include measurement units where applicable, and also a description of what the control is for. Other ways of providing contextual information is via tooltips, help buttons and messages in the status bar. Dialog box messages should also have meaningful text in them.



### Figure 13: Label controls to indicate their function

Use standard fonts (typically the default fonts) where possible. In terms of on-screen readability, sans-serif fonts (Arial, Verdana, Helvetica and so on) are generally considered more readable than serif fonts (Times Roman and so on). Script-based cursive fonts are very hard to read on-screen.

When presenting numerical data for visual comparison, use the following in order of preference: length (e.g. bar chart, linear displacement, graph display), angle (dial knobs and indicators), colour (which should only be used for crude comparisons).

Larger applications should have an integrated help file. Include help on the interface where appropriate.

Use progress bars or other indications of activity (hour glass cursors etc.) to indicate the progress of any activities that can take a long time (longer than a couple of seconds). Otherwise the user may believe the system has crashed and attempt to close the application forcefully.

If data is saved from an application, let the user choose where the file will be saved.

Standard exit routes should be labelled so users know how to end the program. It is normally good practice to create an explicit “stop” button to allow the program to clean up before finishing, rather than just aborting it.

#### *Relationship to hardware interfaces*

An appropriate interface for computer interaction is not always the same as the best one for a hardware device. If you want to create a control panel for an existing device, don't just re-implement the hardware as it is. This will require users to be familiar with the older hardware device to become accustomed to the functioning of the new interface. Users will expect the software interface to use standard controls with familiar behaviour where possible.

Another reason for not simply re-implementing the hardware control panel is that some physical controls that allow sensitive manual fingertip control do not work well for a keyboard-mouse interaction mode. An example of this hardware control panel fixation is the use of very small control knobs that are hard to control with a mouse. Instead, use existing hardware control panels as guidance, but try to build the new interface using standard controls where possible and think of the functionality (rather than just replicating the look of the original instrument).

Lastly, software interfaces enable some interlocks etc. to be implemented much more easily than in hardware. For example, it is simple to “grey out” a software control that is not applicable at the current stage in a measurement, whereas creating hardware interlocks to prevent the operation of controls according to the setting of others is complex and expensive.

#### *Modal effects*

Be careful about presenting information that is live but that freezes in certain modes.

A particular situation arises when modal dialogs are used. These dialogs are designed to lock out the rest of the application and will prevent the user from reaching all other parts of the interface (in particular any abort/stop buttons that have been defined). They will also stop other

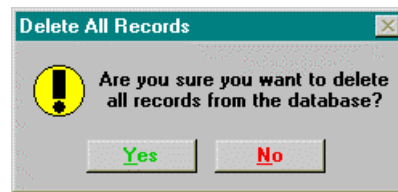
displays updating (e.g. the temperature display does not update when the printer dialog is displayed).

With standard software packages, some modal interaction is unavoidable, but should be reduced as far as possible (e.g. by using drop-down menus rather than dialogs). Some warning of modal effects can be provided on the user interface, for example by greying-out displays that do not update while a modal dialog is displayed.

#### *Use of colour*

Use colour sparingly. Where colours are used, use standard colours, with default (usually grey) for items that do not use colours. Colours can be used to good effect in drawing attention to some behaviour (a value exceeding some set-point) or to indicate a change of state (like an LED indicator). However remember that a significant proportion of male users are red-green colour blind. Colour is a bad way to indicate subtle comparative values (e.g. using shades of green to show values that need to be compared); use linear or angular displacement instead (e.g. a graph or length display).

Colours also have standard cultural connotations and these shared meanings should be overridden with care. For example green will generally be interpreted as indicating a “good” or “acceptable” response; red denotes an “undesirable” or “hazardous” state. As an example of ambiguous use of colour, in [Figure 14](#) (where the Yes button is in green and the No button is in red) consider which is the hazardous response and whether the colours reflect this.



**Figure 14: How to confuse with use of colour**

If your users are from other cultures you should consider whether the colours you employ have different connotations to the ones intended.

The use of garish colours should be avoided.

#### *Error avoidance and handling*

Although the rules above will help the user avoid errors by understanding the interface, you should design-in specific features to avoid and detect errors.

When there are a number of options to choose from, use a select box or enumerated list, rather than relying on the user to type in the value. This reduces the scope for user error.

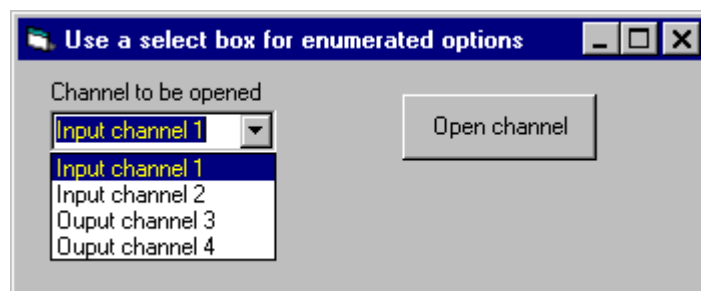


Figure 15: Use select boxes for enumerated options

Use input validation for fields where users enter data by checking whether the value entered is of the right type and in the correct range.

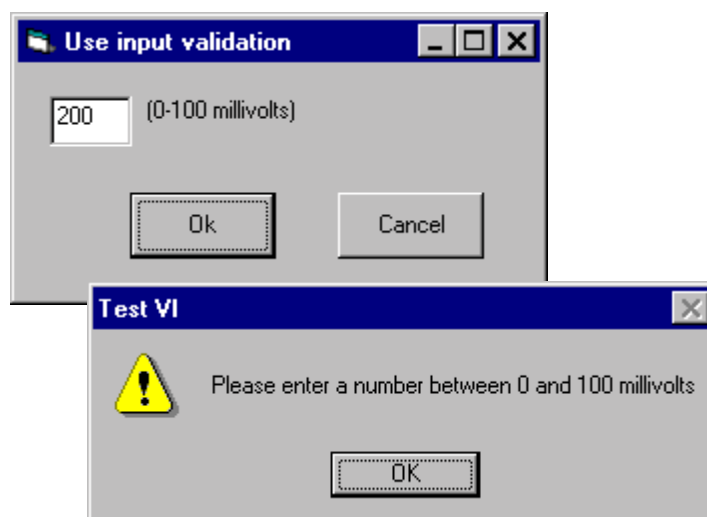
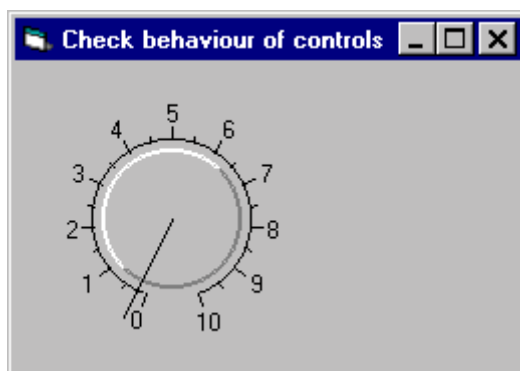


Figure 16: Employ input validation routines to check user data

Design the interface to reject input that is not appropriate for the state of the measurement, for instance by greying-out controls.

#### *Third-party controls*

When using third-party controls you should verify that the interface behaviour is as expected and there are no unwanted interaction modes. For example, the dial knob in the National Instruments “ComponentWorks” ActiveX control library (and also in LabVIEW) allows the user to adjust the value of the knob up to its maximum and then pass directly to its minimum value (i.e. clock-wise from 0–10 and then smoothly to 0 in the following example) in a single mouse movement. This is contrary to the usual experience of a hardware knob with a minimum and maximum value. For instruments controlling sensitive samples this could cause damage to a specimen. A safer approach might be to use a linear slider, or not to have real-time response to user interaction.



**Figure 17: Check interface controls for unexpected behaviours**

#### *Use of different operational modes*

You may design your T&M software to run in different modes. A particular example is the provision of a simulation mode for checking the software and for training. This is potentially dangerous if the user does not realise that they are in simulation mode when making a critical measurement.

Therefore you should make messages regarding operational mode obvious by appropriate use of font size, colour, brightness and flashing display. Messages should also be present continuously, not intermittently.

It may be appropriate to have some sort of access control (e.g. via a password) to prevent inadvertent or unauthorised use of special modes, but it is normally better not to require a password to return to normal mode, in case the software is left inadvertently in the special mode and the normal user does not know the password.

## **2.14 Organisational support and leverage**

In this section we discuss some of the organisational issues that are important in the development of T&M software expertise and competencies within the organisation. Although ultimately software is written by individuals, it is important to address these issues and discuss synergies between organisational-level behaviours and individual practice.

Many of these practices come under a broad banner of what is sometimes termed “knowledge management”. Although this is a somewhat nebulous term, the basic idea is to address how an organisation makes the best use of its intellectual assets. At a practical level, there are a number of practices that can enhance the development of software expertise:

- *Knowledge sharing and the development of expertise*—Developing competencies and skills is a key strategy for organisations to increase their intellectual capital. In large organisations it is typical that “pockets of expertise” develop in isolation in different departments. Users with expertise in certain areas can be encouraged to share their experiences through recognition of the value of this expertise. Certain kinds of knowledge (e.g. so called “tacit knowledge”) are hard to transfer via documentation and other tangible artefacts (for example, knowledge concerning operation of complex equipment and the interpretation of results). Two organisational structures

that aid the dissemination of expertise are “mentor-apprentice” relationships and knowledge-sharing networks. Mentor-apprentice relationships are good for transferring tacit knowledge in which an experienced developer works with newer employees. “Knowledge-sharing networks” are informal groups of individuals who have a specialist interest in a certain problem, research area or technology, both within an organisation and in other organisations or companies. They are typically characterised by a loose coupling and informal communication and knowledge sharing. Building on these emergent networks can be achieved through tolerant organisational support rather than a strict management approach.

- *Organisational culture*—This refers to the set of unwritten rules and practices concerning “how we do things around here”. Individual practices are modulated by the practices and culture of the organisation in which they are working.
- *Access to resources (Internet, external and internal)*—Self-directed learning is the basis for much of the learning that takes place within an organisation. In comparison to formal training courses, which may be relatively costly, developers can learn from books, each other and the Internet. Developers need to be encouraged to contribute to these resources, which include.
  - *Intranet*—An internal intranet is an excellent mechanism for sharing expertise and developer resources within an organisation. An intranet is a corporate information architecture based on Internet standards and technologies (Web browsers, email, search engines). Developers use an intranet for developing internal code libraries, Web pages on issues and concerns, illustrative code snippets and examples. Email is a powerful technology for communication and collaboration. At a managerial level, organisations can put quality documentation (procedures, work instructions, coding standards, etc.) on the intranet to allow easy dissemination, updating and hypertext referencing.
  - *Books and materials*—These include departmental libraries and internal support documentation (quality standards, procedures, etc.).
  - *Links with other organisations*—For example, this can be through common interest clubs such as the SSfM special-interest groups.
  - *Internet*—The widest range of material for software developers is on the Internet (see also [Appendix A](#)). Organisations should collect URLs of useful sites and monitor relevant list servers. Although there are security and IPR concerns arising from the use of material from the Internet (e.g. viruses), organisations are ill-advised to prevent developers from accessing these resources, although some restrictions may be appropriate. Conversely, individuals using these resources need to be aware of the security and assurance issues of using materials from the Internet.
- *Training*—Formal training courses exist for most T&M software development technologies (e.g. LabVIEW, Visual Basic). Organisations should identify courses that address the needs of different levels of developer.

Organisational knowledge-sharing practices should be designed to provide a benefit and incentive for participating individuals.

## 2.15 Operation

Detailed guidance on T&M instrument operation is outside the scope of this guide. However, there are a number of things you should do to maintain the integrity of the software and its results over time, which should be part of your operating procedures. They include:

- *Operator training*—Ensure that the operators are provided with adequate user documentation, and given appropriate training where necessary.
- *Checking the measurement environment has not changed*—Ensure that the environment in which the T&M software operates has not changed away from the original assumptions with respect to the nature of the sample, calibration samples, throughput, ambient temperature, vibration, electrical noise, etc. If you reuse T&M software on another computing platform, you need to re-validate is as described above in [Section 2.6](#).
- *Checking the hardware*—Ensure the hardware is still operating correctly by doing disk fault checks, examining system log errors (e.g. the Windows NT error log can show card driver errors that are not shown elsewhere), etc.
- *Checking performance and calibration*—Carry out a “sanity check” on performance and calibration, and record the results. This is easier if it is “designed-in” to the software, as discussed in [Section 2.4.5](#).
- *Data archiving*—Ensure that measurement data is adequately archived and kept securely away from the T&M instrument.
- *Routine maintenance*—Carry out necessary preventative maintenance and re-calibration on plug-in cards and other T&M instrument hardware items.
- *Perfective and adaptive maintenance*—Make sure there is a way in which users can feed back their experience of the T&M software and make suggestions for improvements. See also [Section 2.7](#).



## Part 3 Technology-specific guidance

### 3.1 Introduction

In this section we present a number of best practices associated with the dominant technologies for developing T&M software, namely:

- LabVIEW and the graphical programming language G
- Visual Basic/Visual Basic for Applications (VB/VBA) together with ActiveX objects
- C/C++
- Java
- Delphi

For each technology we provide specific guidance following the lifecycle described in Part 2.

### 3.2 How to select appropriate tools

#### 3.2.1 Initial selection

In choosing between technologies there are a number of aspects to consider:

- *Existing investment in hardware and software*—Are you migrating from an existing system? For example, upgrading from QuickBasic to Visual Basic will be less onerous than re-implementation in another language.
- *Availability of drivers*—Are hardware drivers available for your proposed development platform? You may decide to write your own driver, but this is not always a trivial task.
- *Throughput/determinacy requirements*—Are there any real-time or high speed data capacity requirements that rule out certain platforms. For example, Windows 95 is non-deterministic in the sense that it is not always possible to guarantee measurements at a specific time interval (for example, the system of interrupts can interfere). Other platforms are available (for example VME, LabVIEW RT or embedded systems) that are deterministic, but this may limit your choice of development software.
- *Skills available*—Does your organisation have an existing skills base in one or more languages? Even though most modern development environments are quite user-friendly, there is still a learning curve in adopting new tools and technologies. Most organisations tend to underestimate this.
- *Customer requirements (e.g. platform, integration etc)*—Are there any end-user requirements in terms of platforms or integration with existing systems?

- *Long-term software and IT strategy*—You should consider how the proposed tool fits in with your organisation’s long-term software strategy. For example, there may be risks in using obscure languages or hardware components that may not have good long term technical support, or may not have an upgrade path. Further, there is a risk of “lock-in” if you invest heavily in one particular vendor. As the amount of effort increases you should use tools and technologies based on industry-based and non-proprietary standards where possible to reduce these risks.

### *3.2.2 Upgrading existing tools*

The market in software development tools is such that vendors release new versions with new features at frequent intervals. This raises the issue of which version to use for new developments, and whether to upgrade old T&M software to use new versions.

When a new version of a tool appears, the first stage is to do an impact analysis to determine the nature of the changes (e.g. the object model might have changed, or just the editing environment). Usually vendors have a “what’s new in this version” section in the documentation.

As a general principle, you should upgrade existing T&M software if you do not want the code to become legacy software. Otherwise, when you upgrade the computing platform (e.g. to a new operating system), there are likely to be changes that make software maintenance more difficult, even if the T&M software continues to run.

## 3.3 LabVIEW

### 3.3.1 Introduction

LabVIEW is a software development environment produced by National Instruments (<http://www.ni.com/>). It uses a wholly graphical<sup>2</sup> approach to developing software. T&M programs developed in LabVIEW are referred to as “virtual instruments”, and for this reason the term is used throughout this section. LabVIEW has always had a focus on supporting software instrumentation and comes with large libraries of examples, wizards and documentation.

In the consultations that preceded development of this guide, we found LabVIEW to be the most widely used technology for VIs. This is partly because its graphical, data-flow programming language provides a more hardware-like notation for non-programmers to use, and partly because of the extent of support for measurement hardware.

National Instruments also provide other VI products, such as LabWindows CVI (an integrated C environment for VI development), LabVIEW RT (which generates embedded executables that run in real time on dedicated hardware, leaving the PC to provide non-time-critical functions such as graphical interfacing and networking) and BridgeVIEW (a version of LabVIEW for industrial supervision and control).

Technical support and documentation from National Instruments and third-parties [1] is of a high standard. The on-line help for LabVIEW is supplemented by a range of technical support on the National Instruments Web site, which can be navigated in a number of ways, including searching on function names.

The illustrations were prepared using LabVIEW 5.1.

### 3.3.2 Requirements description

All the general guidance in [Section 2.3](#) on developing the requirements description applies well to LabVIEW.

### 3.3.3 Design

See [Section 2.4](#) for general guidance.

Top-down design is appropriate for large LabVIEW VIs. We recommend that top-down design starts with the user interface, by establishing the front panels that will be needed, and the controls and indicators. This will also identify the need for real-time analysis, data manipulation, etc. The design can then proceed by identifying functional blocks or sub-VIs for the major functions as illustrated in [Figure 5](#). Although sub-VIs provide a powerful way of hiding detail, be careful about creating “spaghetti code” by making the hierarchy too deep and with too many wires connected to each sub-VI. This is discussed some more in [Section 3.3.4](#) below. For more guidance on top-down design, see *Program Design* in the LabVIEW Online Reference help.

---

<sup>2</sup> An overview of other visual programming languages can be found at: Visual Language Research Bibliography: <http://www.cs.orst.edu/~burnett/vpl.html>

Bottom-up design will be more appropriate for driver sub-VIs, low-level bit handling, numerical algorithms etc. LabVIEW provides a large number of drivers and it will save programming effort, and probably make your VI more reliable, if you make use of existing drivers in your design, or one that is close and can easily be modified. If you are using a National Instruments board, the data acquisition (DAQ) examples included with LabVIEW are a good starting point.

### *Timing*

See [Section 2.4.2](#) for general guidance.

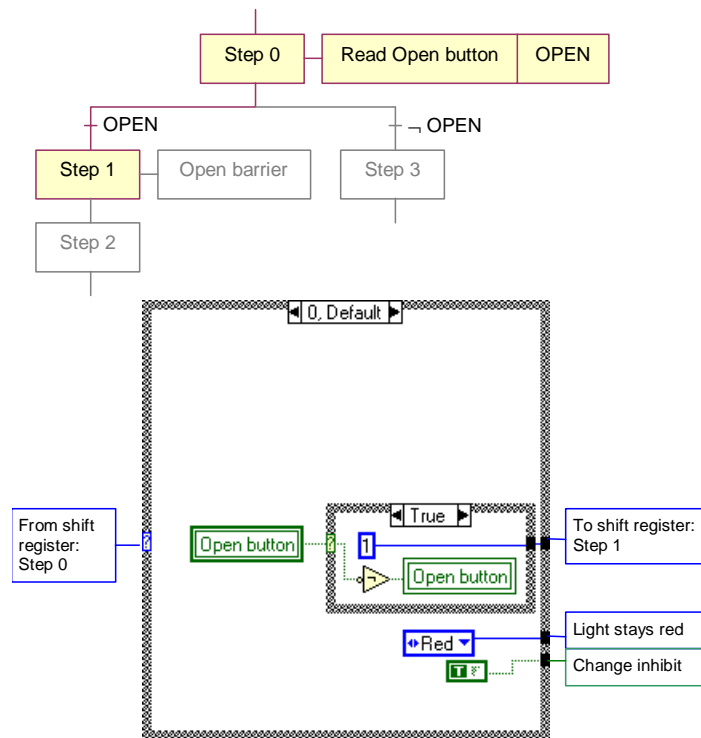
Users report that LabVIEW typically runs six times slower than Visual Basic. If your application has hard real-time requirements, you may need to consider LabVIEW RT or use of another language.

Because it runs on dedicated hardware, LabVIEW RT gives protection against operating system problems and the core functionality will continue even if the operating system crashes; however, the user interface will be lost.

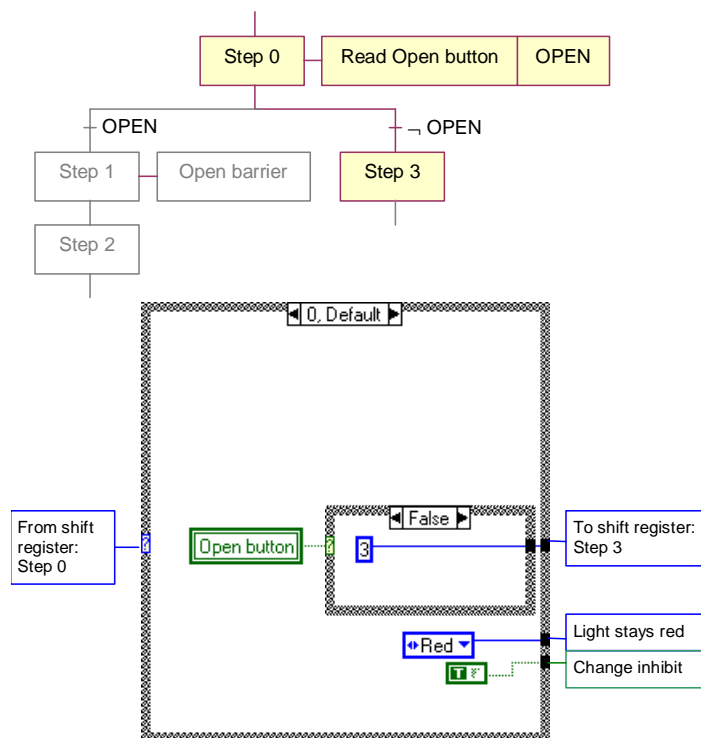
### *Design modelling*

The design notations illustrated in [Section 2.4.3](#) can all be applied to LabVIEW.

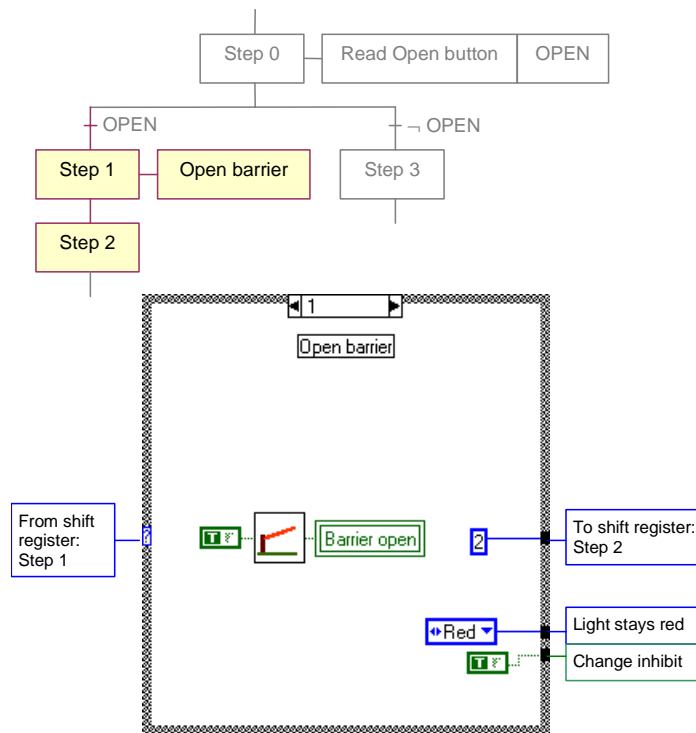
- *Dataflow diagrams* ([Figure 6](#))—These map onto functional blocks, with the data flows corresponding to wires.
- *State transition diagrams* ([Figure 7](#))—These map onto the state machine structure provided in LabVIEW by a Case structure within a While Loop. The state number is the case variable, which is fed back from the output of one case to the input of the next using a shift register in the While Loop. [Figure 18\(a,b,c\)](#) shows the Case blocks for the transitions from Step 0 and Step 1 of our example. (Note that, because it is necessary to provide the same terminals in all blocks for a Case, we have to output an aspect value even though it is not required by the state transition diagram; the “change inhibit” Boolean is to prevent the traffic light responding to this. Note also that we reset the Open button if it is True.)
- *Object models* ([Figure 8](#))—These map onto blocks or sub-VIs. The attributes of each object represent indicators, controls, shift registers, etc. Note that the G language is object-oriented, and a LabVIEW program itself is composed of objects. Thus the front panel is an object, with attributes (e.g. Front Panel.ShowScrollBars) that can be accessed via the Property Node structure on the Application Control Functions pop-up menu.



(a) Step 0 to Step 1



(b) Step 0 to Step 3



(c) Step 1 to Step 2

Figure 18: State transition diagram implemented in LabVIEW

Be careful about order of execution of LabVIEW blocks. Blocks that are not sequenced by means of a sequence block or data flow may execute in any order, and not in the left-to-right, top-to-bottom order implied by the layout. (We take advantage of this to model the approaching vehicle in the worked example.)

#### Data dictionary

We recommend that you maintain a data dictionary for LabVIEW VIs, as described in [Section 2.4.3](#). If you wish, you can add pictures of the front panels to illustrate the controls and indicators. Part of the data dictionary for the example is given in [Table 3](#).




Name	Description	Type
Aspect	Internal variable with four values representing traffic light aspects	Enumerated type (underlying type Integer): "red", "red_amber", "green", "amber"
Change inhibit	Internal variable to prevent traffic light state machine responding to aspect data when not required	Boolean
Close button	Control (switch when released): True = released. Clicked by user to close barrier 	Boolean, local variable
Enable controller	On/off control for the VI, linked to indicator showing value 	Boolean, local variable
Green 1	Indicator: State of the green bulb: True = on 	Boolean, local variable

Table 3: Data dictionary for LabVIEW program

### UML

UML can be used for LabVIEW design. Statechart and activity diagrams and class diagrams map well onto LabVIEW programs as described above. The other diagrams in the UML are more suitable for conventional programming languages like Visual Basic and C++.

### Formal methods

Formal methods such as VDM and Z can be used with LabVIEW. Statecharts can also be used for LabVIEW VIs that are state-based, as in our example. Methods such as B that are intended to translate into a procedural language are less suitable. Pre- and post-conditions in UML models can also be used.

### Design reviews

Design reviews are as important with LabVIEW as a procedural language. Some review checklists for LabVIEW are contained in *LabVIEW with Style*, which is on the CD-ROM with [1].

See [Section 2.4.4](#) for general guidance.

### *Fault tolerance and fail safety*

All the guidance in [Section 2.4.5](#) applies to LabVIEW.

You can implement diverse calculations in LabVIEW and another programming language in several ways. You can call code in another language directly from a block diagram using a *Code Interface Node* (CIN). You can call 32-bit libraries (dynamic linked libraries in Windows, code fragments on the Macintosh or shared libraries in Unix) using the *Call Library* function. You can also run a diverse VI as a separate application, and communicate with it via TCP/IP or Windows DDE protocols.

LabVIEW RT gives you a way of running time-critical or other critical code independently of the Windows system.

### *3.3.4 Coding*

The guidance in [Section 2.5](#) should be applied to LabVIEW as follows. Further guidance on good programming style is contained in *LabVIEW with Style*, which is on the CD-ROM with [\[1\]](#).

#### *Coding standards*

A LabVIEW coding standard should include the following guidance:

- *Keep VIs small*—You should try to size each VI or sub-VI so that you can view it all at once on the screen. Sometimes this is very hard on a small screen and if possible use a large screen—17 inch or preferably 21 inch. Although modularisation using sub-VIs is a good way of control the complexity of diagrams, you should not arbitrarily cut your VIs into sub-VIs as this can result in complex wiring. Try to restrict each sub-VI to three or four inputs and outputs. If you genuinely need to pass many related wires to a sub-VI (e.g. a data set from an instrument), bundle them together in a *cluster*.
- *Avoid global variables*—LabVIEW provides two sorts of “global” variables: local variables which are global to a VI, and global variables which are global to all VIs in memory. Sometimes their use is essential. However, using them obscures the data flow. It also sets up the possibility of race conditions, the situation where these variables are read or written to in a different order from the one you expected, because of an asynchronous sub-VI or block. You can find all instances of local and global variables using the Find command on the Project menu, or by right-clicking on one instance and selecting Find.
- *Write defensive code*—You should always include an error handler for any I/O call, including file handling. Particularly if you are writing re-usable VIs, you should also check for as many other things that might go wrong as you can think of: these include errors in input data, such as out-of-range values, impossible numbers, etc.; and errors when calling software libraries and operating system functions, which may exist in different versions on different computers. Error handling should be done at the point in the hierarchy at which the error can be fixed (often at or near the top level), and the approach is to propagate error messages through the program using a special wire, which you test to see whether to carry out normal processing or just pass the error on.



LabVIEW has a standard format for error information: the *error cluster*. This is a cluster with three components: a Boolean value, set to True if an error exists; an error code; and a string defining the source of the error. There are also three built-in error handlers included in the Time & Dialog functions. See *Error Handling* and *Check For Errors* (under *Good Diagram Style*) in the LabVIEW Online Reference help for more details, and *Error Codes* for a list of built-in error codes. It is also possible to define your own error codes, as explained in the help.

Error handling for the entry barrier example is illustrated in Figure 19. If the vehicle sensor reports an error, the simple error handler is called and the state machine goes to step 7, where it sets a safe state (barrier open and light on red) and then loops until the VI is restarted.

- *Use enumerated types and clusters*—One feature that should be used where applicable is *enumerated types*; these are labels (strings) associated with unsigned integers, which enable meaningful names to be used instead of integer values; for example *red*, *red\_amber*, *green* and *amber* stand for 0...3 in our example. You should also make use of *clusters* to group related wires. See *Enumerated Constants* and *Clusters* in the LabVIEW Online Reference help for more details.
- *Reuse VIs*—Where possible, make use of the extensive library of drivers and examples maintained by National Instruments, provided with the LabVIEW distribution, from their Web site, and on the *Instrupedia* CD-ROM.

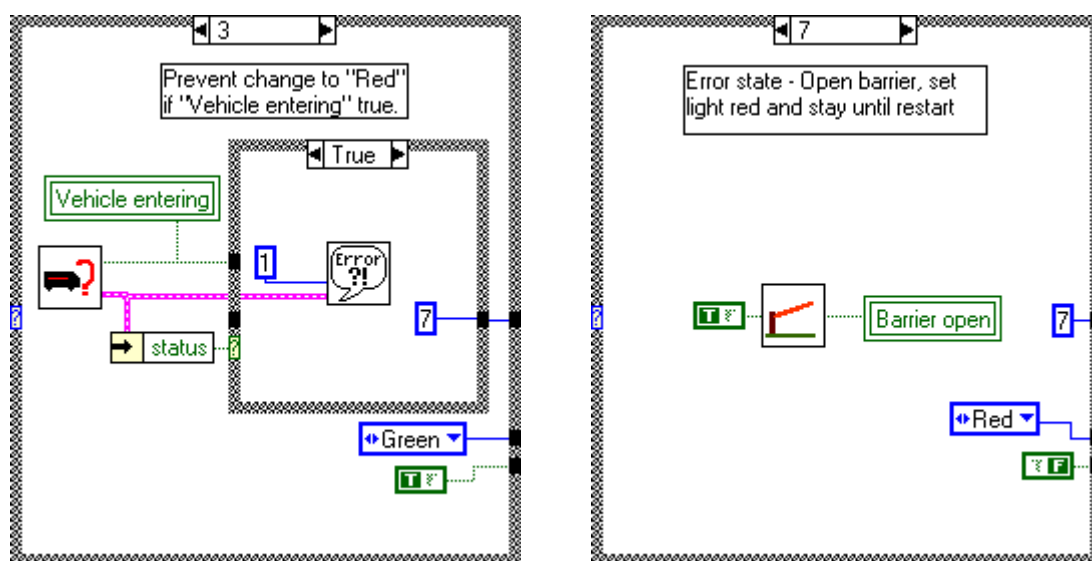


Figure 19: Error handling

#### *Coding and software documentation*

LabVIEW programs look very different from text-based languages such as Visual Basic. Guidelines for laying out and documenting LabVIEW programs are follows:

- *Write readable programs*—Producing readable LabVIEW programs involves neat layout, by obeying the left-to-right, top-to-bottom convention, avoiding crossing

wires and wires running under blocks and icons, achieving neat wiring by aligning sources and sinks, and keeping wires parallel. If you create sub-VIs, try to give them a meaningful icon, using text if there is no suitable image. See *Good Diagram Style* under *Program Design* in the LabVIEW Online Reference help for more details.

- *Comment your code*—LabVIEW provides several way of commenting your code. The diagram in [Appendix D](#) shows some of these, and the worked example (obtainable from <http://www.adelard.com/>) also contains examples of Description dialog boxes and change histories.
  - You can place text comments anywhere on the diagram using the text tool. This is useful to add comments to individual case and sequence structures, and also to label wires, which you can do by setting the foreground colour transparent (T) and placing the label over the wire. Because of the limited “real estate” on a LabVIEW diagram, you will have to restrict free-text comments; you can fit more comments in by using a scrolling string constant on the diagram.
  - You can right-click on any block or wire and fill in the Description dialog box. This is another way of labelling key wires. It is also useful for making general comments on structures such as While and Sequence. It is helpful to include an index to Sequence and Case structures with several blocks.
  - You can produce a help page. This is particularly useful if you are writing reusable VIs. See *Creating Your Own Help Files* in the LabVIEW Online Reference help for more details.
  - You can use the File, Print Documentation feature. This offers a choice of formats. One option includes the complete diagram, with individual Case and Sequence blocks shown afterwards. You can print the report directly, or save it to a file; this is more useful, as it allows you to add your own commentary. It is also possible to copy the diagrams in the report and use them in your own documents.

### 3.3.5 Verification and validation

#### Testing

The general guidance in [Section 2.6.1](#) is all applicable to LabVIEW.

If your application is sufficiently critical to require measurement of test coverage, LabVIEW presents a problem. Since the conventional code produced by the LabVIEW compiler is not accessible, standard test coverage measures cannot be applied. The alternative is to obtain a measure of coverage of the diagram. For procedural languages, tools are available that “instrument” the code by adding statements to record coverage; such tools do not exist for LabVIEW, and, it will be necessary to instrument the diagrams yourself. [Figure 20](#) shows one possibility, where execution of each option in a Case structure increments an element in a global array. This adds complexity to the diagram (although it could be packaged as a sub-VI with the array number as input), and also will slow the program down.

For critical VIs, the goal will be to achieve 100% coverage of structures, and of frames and loops in Sequence and Case structures.

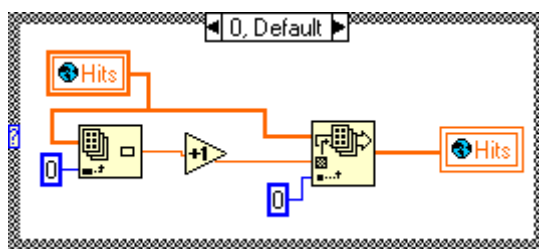


Figure 20: Coverage measurement for LabVIEW

National Instruments supply a VI known as the *Test Executive Toolkit* that enables automated test case execution and reporting. Alternatively, tools such as Microsoft Visual Test can be used to supply values to the user interface and check the settings of the indicators.

#### *Code reviews*

Code reviews should be undertaken with LabVIEW just as with a conventional language. Although the notation resembles wiring diagrams, users report that it takes a while to become conversant with dataflow programming, and therefore all members of the review team will need to be familiar with LabVIEW programming. See [Section 2.6.2](#) for general guidance and *LabVIEW with Style* [1] for style checklists that could be considered during reviews.

#### *Static analysis*

Dataflow analysis is done by the LabVIEW compiler, which identifies broken wires and type mismatches. Control flow is enforced by the compiler.

LabVIEW does not provide facilities for information flow analysis or timing analysis. Since the underlying G language does not have a mathematically formal definition, semantic analysis is not possible.

#### *3.3.6 Maintenance*

The general guidance in [Section 2.7](#) is all applicable to LabVIEW.

#### *3.3.7 Configuration management*

See [Section 2.8](#) for a discussion of the need for configuration management.

The software version number should be displayed on the VI front panel and in the VI's help page if provided. It should also be included on any printed reports.

We recommend that you keep a history of your VI using the History window (on the Window menu). This has a revision number that automatically increments each time the VI is saved, and an area for adding comments. It can be configured in various ways from the History item in the Preferences item on the Edit menu, for example to pop-up the window automatically on closing the VI. See *History, Preferences* in the LabVIEW help for more details. The history can be included in reports printed from the File, Print Documentation item.

The LabVIEW library structure, which places all VIs in a single file, was introduced to overcome the eight-character limit on filenames in Windows 3.1 PCs. This limitation does not

apply to later versions of Windows, and since the libraries can become very large and unwieldy, and also occasionally become corrupted, we recommend that you do not use them unless you need to maintain Windows 3.1 compatibility.

For small VIs, you can implement configuration management by using a series of directories for each version. For larger developments, the Pro G Toolkit provides a Source Code Control system.

### 3.3.8 Metrics

Standard metrics are intended for procedural languages and do not apply well to LabVIEW. However, LabVIEW does provide a VI Metrics item under the Project menu. This provides a number of metrics including the number of nodes, structures and diagrams, the size of the diagram, the number of wire sources, controls and indicators, etc. Some of the metrics for the example are shown in [Figure 21](#).

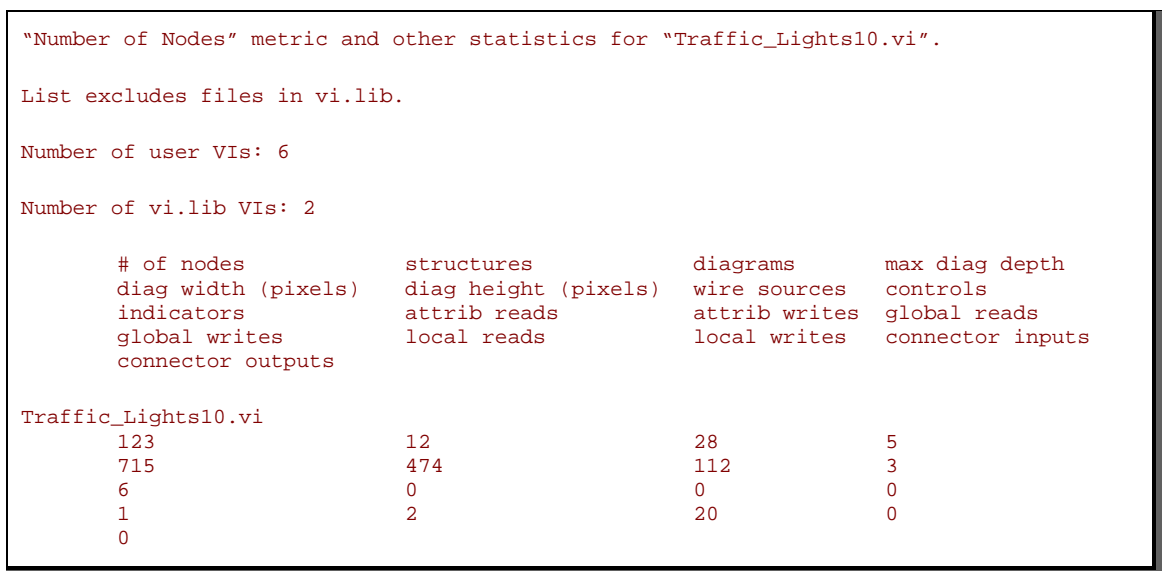


Figure 21: LabVIEW metrics

It is hard to compare these directly to conventional metrics, and it will be necessary to build up a picture of your VI developments over a period of time. These metrics could be combined into a single value by weighting and summing them, in an analogous way to function points (see [\[18\]](#)).

Another metric that may apply to LabVIEW is fan-in/fan-out, which for dataflow diagrams is a count of the number of wires entering and leaving each structure, plus use of local and global variables.

### 3.3.9 Mixed language programming with LabVIEW

General issues of mixed language programming are discussed in [Section 2.11](#).

An example of calling a Fortran library from LabVIEW is given in [Appendix E.5](#).

## 3.4 Visual Basic

### 3.4.1 Introduction

Basic (Beginners All-purpose Symbolic Instruction Code<sup>3</sup>) was developed at Dartmouth College in the 1960s, and was designed to be a very simple language to learn and translate. Originally it was intended as a teaching aid, although its potential became clear and many early microcomputers came with Basic interpreters and then later compilers. Microsoft adopted the technology early on, Basic being the first program it sold.

Early T&M software developers adopted Basic as their development language, given its ease of use and support across a wide range of platforms as the microcomputer emerged as a generic computing platform.

Today, Basic can be considered as a family of languages (IBM-Basic, QuickBasic, Visual Basic, VBA and so on<sup>4</sup>), although the most important variant for the purpose of developing programs on a PC is Visual Basic<sup>5</sup> (VB) from Microsoft, given the breadth of industry support and the general prevalence of the Windows operating system. T&M software written in older variants are still around, for example there are still programs written in QuickBasic used in older T&M software.

Visual Basic is a full development environment for the Windows platform, and recent additions to its functionality includes basic object-orientation and support for third party controls via the 32bit ActiveX architecture. The general model for interface and application design is to use Visual Basic to manipulate defined objects (for example an output display on the user interface, or an object that controls a hardware device), and to respond to events generated by the user interface or system objects. At the time of writing, Visual Basic comes in three versions, all based on the same underlying language model:

- *Visual Basic*—A full Windows application development environment.
- *VBA (Visual Basic for Applications)*—A version of Visual Basic that is embedded in Windows applications and allows the user to script and control predefined application objects. VBA allows application developers to provide a generic programming interface to their application. Current versions of Microsoft Office applications use VBA. Applications are generally distributed as an “Add-In” for the particular application or as a file that contains the embedded macros and customisations. Some SCDA control applications<sup>6</sup> have a VBA interface that allow users to build

---

<sup>3</sup> A history of the early years of BASIC can be found at: <http://www.fys.ruu.nl/~bergmann/history.html>

<sup>4</sup> A collection of General Basic Resources can be found at the Chipmunk Basic home page: <http://www.nicholson.com/rhn/basic/>

<sup>5</sup> Microsoft Visual Basic Home page: <http://msdn.microsoft.com/vbasic>

<sup>6</sup> See for example iFIX from Intellution: <http://www.intellution.com>

customised applications by scripting the SCDA objects and user interface components.

- *VBScript*—A cut-down version of Visual Basic, primarily used in developing dynamic Web applications (server-side and client side scripting). Here the principle of scripting predefined objects is the same; the only limitation is the inability to define new objects and collections directly. Applications are typically distributed as a collection of script files.

Migration from other (generally older) versions of Basic to VB is generally quite straightforward, given that the underlying control structures are the same. Tricky aspects can be that user interfaces in older version used custom graphics libraries, which some developers found difficult to use and debug. User interfaces are easier to implement using a modern GUI tool such as VB, which provides a good range of widgets and controls and allows the importing of third party interface controls.

Online help (help files) and Internet support for VB is generally very good.

In the rest of this section we assume the reader is using VB5 or later, although some of the discussion applies to VBA applications and older versions of basic.

A Visual Basic project consists of a top level project file (.vbp) and associated modules, forms and classes. The project file is a text file and contains references to the other files, so that they are loaded into the development environment when the project is opened. The project's forms provide the user interface to the application.

The model for building a VB application is generally event driven, that is to say the logic of the application is triggered by user-driven and application-driven events.

### *3.4.2 Requirements description*

The general guidance in [Section 2.3](#) applies.

### *3.4.3 Design*

For simple applications all of the code can be stored on the form. For larger applications where you reuse functions defined, you should create modules to contain the functions and methods. Give the modules meaningful names to help locate the functions.

Define variables with their object types (string, object, collection, database and so on). Otherwise VB will treat them as variants (a general purpose variable type), which uses more memory. The other advantage of declaring variable types is that you get drop-down lists of available properties and methods according to where you are in the object hierarchy. In VB, variables are declared using the “Dim ...” statement. Use “Option Explicit” at the head of each module and form. This will force you to declare variables before they are used.

Rename controls as they are added to the user interface, and give them meaningful names, such as:

- “Text1” → “txtUserName”

- “Button1” → “btnInitialiseCalibration”
- and so forth

Give variables useful names that encapsulate the use of the variable and its datatype; use coding standards if you want, but the most important aspect is to be consistent.

#### *Data dictionary*

The equivalent data dictionary for the traffic light example is given in [Table 4](#). The main module is given in [Appendix C](#).

Name	Description	Type
blnRequestOpen	a request for the barrier to be opened—fired by btnOpen	Boolean
blnRequestClose	a request for the barrier to be closed—fired by btnClose	Boolean
blnVehicleArrived	a simulator of a random car arriving at the barrier	Boolean
blnVehicleInZone	a detector whether a car is in the zone	Boolean
iCurLightState	a variable for the current light display	Integer (specifically 0,1,2,3)

**Table 4: Data dictionary for Visual Basic program**

#### *Timing*

Visual Basic was not originally designed for building applications with critical timing requirements, although it will cope well with medium to low data capacities. For high-speed applications you should use a dedicated DAQ card.

To build a polling application in VB you can use the timer control as a top level event driver.

#### *Reuse of components*

ActiveX (also known as COM) is the most widely used technology for creating reusable components on PCs. The basic concept revolves around the idea of an ActiveX server that has a unique identification reference. When the ActiveX component is loaded on the same machine as the running application, the application creates an instance of the object that is delivered by the ActiveX server. When the object has been created, its properties can be set and retrieved, and its methods can be invoked.

Visual Basic 5 or later allows the user to develop his/her own ActiveX controls (which have a user interface) and ActiveX DLLs (which have no user interface), which can be compiled and distributed for inclusion in other projects.





- “live” rewriting of code if there is an error when the code is run in the development environment

For T&M software development there is often a real-time aspect to the software. For example a monitor loop might poll a hardware device every 500 milliseconds, get the result and write to a file. These real-time aspects can be hard to debug, as stepping through the code will disrupt the timing component of the application.

A good general resource on VB design and coding is [\[10\]](#).

#### 3.4.4 Coding

##### *Coding standards*

See [Section 2.5](#) for general guidance on code development. In addition, there are a number of coding standards for Visual Basic (see [Appendix A.4.2](#) for further details). A useful book on avoiding common errors and error handling in VB code is [\[5\]](#).

It is good practice to ensure that ‘Option Explicit’ is present in each module to force all variables to be declared.

There are three main approaches to error handling in VB:

1. *No explicit error handling*—This is not such a bad idea, since the errors will bubble up to the calling procedure, and so can be handled there.
2. *User defined error handler*—These are typically implemented with “On error Goto label”. If you create an error handler, you should cover all eventualities, otherwise a procedure may just fail silently (see next item). This error handler can pass them up to a higher level if necessary.
3. *Ignore Errors*—Errors in a procedure can be ignored with “On error resume next”. This should be used with caution as all errors are ignored.

##### *Coding and software documentation*

General guidance on software documentation issues is given in [Section 2.5.3](#).

For simple VB programs it is adequate to document the code by summarising the program and adding comments to describe its function in the source code itself.

Each module should have a comment section at the head of the procedure that explains the constraints, limitations, assumptions and history of the module.

#### 3.4.5 Verification and validation

General guidance on V&V is given in [Section 2.6](#).

#### 3.4.6 Maintenance

General guidance on maintenance is given in [Section 2.7](#).

### 3.4.7 Configuration management

General guidance on configuration management is given in [Section 2.8](#).

In VB, a project is managed by means of a project file (\*.vbp), which is a text file. This project file contains a list of the forms and modules that are included in the project, and any references to ActiveX controls that are referenced.

#### *Manual configuration control*

There are two ways of doing manual configuration control in VB. The basic idea is to create a separate folder that has safe copies of the current version of the project files. VB project files (\*.vbp) contain *full path references* to the forms and modules contained in the project. This means that if you save or copy across a project file to a new location, it will still retain references to the previous version of the files. In order that previous modules and forms are not modified, you need to make sure the new project contains references to a new set of files. In the following we assume that all the project files are stored in the same directory (currently called “v01”). Two alternative approaches are given below:

- *Either*—Create a new folder named after the version of the project (say “v02”). Open the existing project from v01 and save the project in the new location (v02) *AND* save each object (module, form, class module, etc.) individually to this new location. Set all files in v01 to be read-only.
- *Or*—Copy the existing folder (v01) and its contents to a new folder (say “v02”). Open the new (v02) project file (.vbp) in a text editor, removing hard-wired paths for the entries that refer to forms and modules that are part of the previous version (v01) of the project. When you re-open in VB, the new (local) files should be opened. Set all files in v01 to be read-only.

#### *Tool supported configuration control*

The most appropriate configuration control application for VB is Microsoft Visual SourceSafe, since it integrates into the Visual Basic Development environment. Other configuration management tools are listed in [Section 2.8](#).

#### *Other files*

When you distribute a Visual Basic application you need to use the Visual Basic “Application Setup Wizard”<sup>7</sup> to bring together all the files needed to distribute the application. This generates a “.swt” file that lists all of the project files, DLLs and ActiveX controls needed for distributing the application. It is a good idea to save local copies of these system files with your application so that there is a complete record of the resources needed.

### 3.4.8 Metrics

General guidance on metrics is given in [Section 2.9](#).

---

<sup>7</sup> This is known as the “Package and Deployment Wizard” in VB6.

### 3.4.9 Further resources

There is a very wide range of resources for developing Visual Basic Programs.

For T&M software development there are a number of books that outline how to write a program that can interface to the serial or parallel ports on a PC (see [Appendix A.4](#)).

### 3.4.10 Mixed language programming with Visual Basic

General issues of mixed language programming are discussed in [Section 2.11](#).

Visual Basic shares with Fortran the property that arrays are laid out with elements in the same column adjacent. Multi-dimensional arrays passed as arguments to most languages will have to be transposed. Fortran arrays start at index 1, so if the library being called is in Fortran, the command “Option Base 1” should be placed at the start of the Visual Basic module to make VB arrays also start at index 1.

External components and libraries can be used in Visual Basic for Applications as well as Visual Basic.

The table below lists the mapping between Visual Basic types and C types, which we can take as a common basis for comparison.

C types	Visual Basic types
double d;	double d
float r;	single r
int i;	long i
boolean l;	long l
char s;	string*1 s

The use of structures, a feature commonly used in C programs, may also be accommodated. For example a type Spline might be defined as:

```
typedef struct {
    Integer n;
    double *lamda;
    double *c;
    Integer init1;
    Integer init2;
} Spline;
```

The corresponding Visual Basic user-defined type would be defined as:

```
Type Spline
    n As Long
    lamda As Long
```

```
c As Long
init1 As Long
init2 As Long
End Type
```

The major deviation is that the C pointers to type double have been approximated to integers of type Long.

Calling C subroutines from Visual Basic is discussed in [Appendix E.3](#).

Examples of calling Fortran libraries from Visual Basic are given in [Appendix E.4](#).

## 3.5 C/C++

### 3.5.1 Introduction

A small number of T&M software developers use C and its object-oriented superset C++ to develop T&M software, or libraries for other T&M software.

C/C++ is much less amenable to the casual developer who wants to pick up a new language to solve some problem; a simple windows application typically requires considerably more effort and knowledge (e.g. of the operating system and general computing principles) than the equivalent program written in VB or LabVIEW. However use of C++ coupled with application generating “wizards” and libraries, e.g. those provided in Microsoft Visual C++, can produce applications reasonably quickly.

The main advantage of using C/C++ over other languages is that the developer can have complete control over every aspect of the application. This is because C and C++ are powerful languages that allow the developer greater flexibility and control of both high and low-level programming structures. An example of this is that C/C++ programmers can implement their own low-level data processing and memory management algorithms through direct manipulation of user-defined byte-structures and other low-level library functions.

As a result C/C++ programs for T&M software tend to be focused towards one particular operating system (especially for applications with a user interface). Windows applications in C interact with the user interface part of the operating system using calls directly to the windows API (Application Programming Interface), which is the lowest level available to applications programmers.

For Windows applications the most common development environment is Microsoft Visual C (<http://msdn.microsoft.com/visualc>), although Borland (<http://www.borland.com/>) also produce a Windows development environment for C/C++.

C/C++ programs can be statically<sup>8</sup> compiled into very small and efficient libraries and executables. For this reason, and the increased flexibility of the language, C (but not C++) is well-suited to as a language for developing low-level device drivers and for manipulating high volume bytestreams—e.g. for very high-capacity DAQ (Data Acquisition).

In this best practice guide we provide only an overview of some of the issues concerning the development of C/C++ programs, rather than expand the lifecycle as we have for LabVIEW and Visual Basic. We have adopted this position for two reasons:

1. There are relatively low numbers of developer-users compared to LabVIEW and VB.
2. Developers who use C/C++ are generally more aware of the importance of adopting good software engineering practices and tend to come from computing-based backgrounds.

---

<sup>8</sup> I.e. without any references to external libraries or components.

For developing complete VIs using C, a good starting point is LabVIEW CVI (<http://www.ni.com/cvi/>), which is a collection of user interface tools, instrument drivers, analysis routines, and I/O libraries for users of standard C/C++ development tools from Microsoft, Borland, Symantec, and WATCOM.

### 3.5.2 Benefits and pitfalls

#### Good points

Some good points of using C are as follows:

- The language's low-level nature means there is an obvious mapping for each language feature to equivalent machine code, which results in more predictable performance.
- Variables can be defined (and initialised) in the smallest possible scope, giving good locality and information hiding.
- Literals can be named and safely modified.
- Static typing.
- Simple assertions can be executed at run-time and also by the macro preprocessor at compile-time to check for errors.
- Declarations have attributes for controlled access to hardware addresses and for indicating the initialised but unchangeable nature of a variable.
- Independent compilation of "modules", the linkage of which can be made safe by using appropriate tools.

#### Bad Points

Most of the potential pitfalls (of which there are many) arise from the fact that C/C++ is such a powerful language. Compilers tend to accept at face value any instruction written by the developer, using coercion<sup>9</sup> where necessary. This can result in developers adopting lazy programming styles to implement shortcuts in their programs; the language is seen to be defined by what is accepted by the compiler.

References [13] and [14] fully explore the issues of "safe" coding in C in the light of the many potential hazards. Some notable problems include:

- Use of "=" and "=". In C, an assignment statement (=) has a truth value and therefore can be used in a test instead of == by mistake. Thus:

```
if (a = 0){  
    //do something
```

---

<sup>9</sup> Forcing one datatype (e.g a string) to be evaluated as a different one (e.g. an integer).

```
}
```

will be interpreted by the compiler as “set *a* to 0” (which returns true), and results in the statement being evaluated always.

- Use of side-effects in a procedure. This is generally discouraged as undesirable coding style since it makes it harder to debug a program and track how it actually should behave.
- String manipulation (which is important for parsing strings) can be error-prone.
- Use of multiple inheritance in C++ is generally considered hard to understand and manage.
- Case statements require a “break” to mark the end of each distinct case, otherwise the next case is executed as well as the one intended.

#### *Resolution*

“Safer C” [13] is one attempt to restrict C to a safe subset of programming constructs and practices. In the automotive industry the MISRA (Motor industry Software Reliability Organisation) guidelines (see <http://www.misra.org.uk/>) have been developed to support the development of an agreed safe subset of C for embedded automotive systems. Both of these have static analysis tool support.

#### *3.5.3 Further help*

A small selection of further resources for C/C++ are listed in [Appendix A.3](#) and [Appendix B.1.3](#).

#### *3.5.4 Mixed language programming with C/C++*

An example of the way Fortran subroutines can be called from C/C++ programs is given in [Appendix E.1](#).

## 3.6 Java

### 3.6.1 Introduction

There are two components to Java: the Java programming language, and the Java platform on which it runs.

Like C++, the Java programming language is object oriented: that is, a program consists of a number of class definitions that encapsulate descriptions of data types and the methods that operate on them. Its basic syntax and choice of keywords follow the C traditions, but it is not just another object-oriented extension of C. The language design allows for strong type checking almost everywhere and storage management is completely automatic (there are no explicit pointers, or calls to allocate and free storage), so a number of the most error-prone aspects of C are removed. It is also inherently multithreading, so a program can be expressed as a number of concurrent tasks.

The Java platform is responsible for the execution of Java programs. Java programs are normally compiled to a standardised “bytecode” form representing instructions to an abstract Java Virtual Machine (JVM). The Java platform implementation for a particular processor and operating system configuration interprets this bytecode as a series of hardware instructions and operating system calls. The same program can therefore be run without code changes on any system where a JVM has been implemented. Implementations exist for most workstation/operating system combinations and have been embedded into consumer devices such as digital set-top boxes and mobile telephones. The bytecode representation results in compact programs that can readily be served over networks, and the integration of the JVM with standard Web browsers allows programs (*applets*) to be embedded in HTML pages and executed on the client machine.<sup>10</sup> Again, this is independent of the hardware and operating system that that machine is using. The JVM also imposes strict limits on the local machine resources that the applet can access, so the client is less vulnerable to the downloading of potentially malicious code. (This may, however, restrict the functionality that an applet can provide.) In the VI area, this potential has been exploited to allow instruments to be controlled remotely from anywhere on a network.

Java class libraries provide standard data structures, input-output, and the elements of graphical user interfaces. As in Visual Basic and C++, Java programs with GUIs are typically event driven, with user actions on the interface controls invoking methods within the code.

Java is supported by a number of integrated development environments (IDEs). The Java Bean interface provides the Java analogue of an ActiveX control: a user interface component that can be developed by the user or obtained from a third party, and added to an application by dragging it into the application interface and establishing handlers for its various events. The appearance of a Bean can also be altered through interaction with the IDE, without explicit programming.

A number of test equipment manufacturers have used Java either to define Java Beans or remote interfaces to specific items of measurement equipment. A Java Beans package will typically

---

<sup>10</sup> Unfortunately, disputes between Microsoft and Sun have meant that Internet Explorer is supplied with version 1.1 of the JVM, so if applets are to be widely distributed they must be compatible with this version.



contain classes for interfacing to instruments (which is easy if they support the TCP/IP protocol and somewhat more difficult for instruments using GPIB or other busses), and other classes for providing T&M instrument displays (meters, pen recorders and so on), scaling, disk logging and the like. These can then be combined with Java Beans such as the Swing GUI classes which provide user input devices to build complete instruments. Simple examples can be constructed using only a little Java code to glue the components together.

Java platform implementations, Java compilers, and many of the Beans and IDEs are made available freely and often on an open source basis. This, in conjunction with the possibility of presenting the virtual instrument interface on a Web browser, can significantly reduce software licensing costs.

Java and its development environments have been through a number of versions, and changes in the structure of libraries and components may require changes to be made when using old code in newer settings.

Java is supported by a strong advocacy group and a great deal of on-line material, including extensive tutorials on the language and development environments.



It should be noted that the licensing conditions for most Java implementations warn that it is not designed or intended for use in safety-related systems, and is not warranted for fitness in such applications. Given that very little software is warranted to be suitable for anything, the Java user is probably no worse off than the user of other languages, but this emphasises the need for developers of safety-related systems to gather their own evidence of fitness of their product for its purpose.

### *3.6.2 Requirements description*

The general guidance in [Section 2.3](#) applies.

### *3.6.3 Design*

A Java application with a GUI will contain a class that is a subclass of one of the GUI container classes provided by the Java library. Each control on the interface is represented by an interface variable of the appropriate class for the control and is suitably initialised when an instance of the main form is created. (The initialisation includes defining handlers for each control event of interest, as well as details of the presentation of the control.) The `main` method of the main form class creates an instance of the class and displays it. Because all the form structure is expressed as Java code, this can be programmed directly, but the GUI builder component of an IDE will normally generate most of the code automatically.

If using a GUI builder, you should ensure that the controls are given meaningful names as they are added to the user interface. The strong typing of Java makes the use of type prefixes recommended for Visual Basic less necessary.

### *Timing*

The interpretation of bytecode inherent in the Java platform entails some loss of speed, so Java is not the best choice for applications with heavy data processing requirements or tight response time limits. However, threading allows quick response to external events while updating displays less frequently. Some Java platform implementations for embedded applications offer

real-time capability, and there is an effort underway to produce a standard definition for real-time Java.

The Java library includes a timer control that can be used as a top-level event driver.

### *Reuse of components*

Reuse is a central goal of object oriented languages, and Java provides strong support for combining class definitions into reusable packages (which will typically include a number of classes). Java Beans provide an open standard, platform-independent alternative to ActiveX components.

### *User interfaces*

A general discussion of interface design issues can be found in [Section 2.13](#).

The Swing collection of Java foundation classes is included in the Java platform and provides a good selection of standard controls, including push buttons, text boxes and file system browsers. There are a number of on-line tutorials describing the contents of the classes and giving examples of their use. The collection of controls can be extended to include more specialised controls from third parties, usually packaged as Java Beans as described above. It is also relatively easy to construct new controls, although it is better where possible to use standard control mechanisms rather than adopt custom approaches with functions that may not be immediately clear to the user. The look and feel of Java interfaces can be changed relatively easily to fit in with the windowing style of the host platform.

Many third party controls are supplied as source code as well as examples of use, so their behaviour can be fully understood and even modified if necessary.

### *Debugging*

Java IDEs normally provide code debugging capabilities. For example the Sun Forte Community Edition IDE supports breakpoints (including conditional breakpoints based on variable values), watches on variable values, observation of the call stack and so on. The debugger can monitor activity in multiple program threads simultaneously.

As usual, any program requiring a real time response to input devices is not amenable to breakpoint-based debugging. If the program has been designed as a set of threads, it may be possible to debug those that are not time-critical without affecting the others.

## *3.6.4 Coding*

### *Coding standards*

See [Section 2.5](#) for general guidance on code development. There are a number of suggested standards for Java coding, and some on-line examples are listed in [Appendix A.6.2](#). There are also tools that can automatically check for compliance with these standards (see [Appendix A.6.3](#)).

Java adopts a throw-and-catch approach to handling errors. The “catching” side of this is written as `try {<block>} catch {<exception handlers>}`. Any exception thrown in the block within the `try` part is matched against the exceptions for which handlers are defined in the `catch` part. Those that match are processed by the corresponding handler, while the rest continue to propagate up through the calling structure of the program. Exceptions can be thrown

by fundamental operations such as division by zero, library code, or the user's own code using the `throw` statement. This mechanism makes very explicit which errors will be handled and how they will be treated at any point in the code, without burdening it with explicit error flags.

#### *Coding and software documentation*

General guidance on software documentation issues is given in [Section 2.5.3](#).

The obvious places to document Java code are at the starts of packages, classes and methods. The `javadoc` tool defines some conventions for structuring these comments and a method of extracting them to produce documentation as HTML pages (or in a number of other formats).

#### *3.6.5 Verification and validation*

General guidance on V&V is given in [Section 2.6](#).

There is a broad range of tools to support verification and validation. We have already mentioned tool support for coding standards. Other tools offer deeper static analysis (which can identify dead code, uninitialised variables and similar problems), test suite construction and execution, code coverage assessment and so on. These tools may be offered commercially or as freeware or shareware. A short list of examples and a pointer to a more complete listing are given in [Appendix A.6.3](#).

#### *3.6.6 Maintenance*

General guidance on maintenance is given in [Section 2.7](#).

In an ideal world, code would be associated with sufficient documentation to allow the impact of proposed maintenance changes to be assessed. In practice, this may not be the case, and there are a number of Java tools available for constructing call trees, class trees and other structural views of existing Java code, browsing from uses to definitions, and similar code understanding tasks. One particular issue that arises in object oriented languages is that of *refactoring*: that is, changing the class structure of the code to express better the relationships between different ideas and maximise the re-use of method definitions.

#### *3.6.7 Configuration management*

General guidance on configuration management is given in [Section 2.8](#).

The Java package and interface mechanisms, and the Java Beans conventions, provide convenient units for configuration control. Java IDEs such as the Sun Forte Community Edition IDE and Borland's JBuilder define the notion of a project to encompass a complete development.

#### *Manual configuration control*

As the discussion of Visual Basic shows, IDE project files normally include lists of component source files, and copying the current development directory may not be enough to checkpoint the development. You should refer to the IDE documentation to determine the steps that are necessary.

The Sun Forte Community Edition IDE allows an existing project to be renamed and saved using the Save As item on the file menu. This is enough to start a new development branch.

### *Tool supported configuration control*

Both the Sun Forte Community Edition IDE and JBuilder provide an interface to the CVS version control system. Other version control systems can be supported in Forte by defining an alternative connection to the interface.

### *Distributing applications*

Java applications assume nothing more than the existence of the corresponding version of the Java platform on the receiving machine. All controls and libraries that are specific to the application are represented in the bytecode of the application. This avoids the “DLL hell” associated with Visual Basic and Visual C++ application distribution (and the need for the packaging tools designed to avoid it).

### *3.6.8 Metrics*

General guidance on metrics is given in [Section 2.9](#).

A number of Java tools will report complexity metrics for code, typically in conjunction with other forms of static analysis.

### *3.6.9 Further resources*

There is a very wide range of resources for Java developers, reflecting the open source roots of the language. A few major sites are listed in [Appendix A.6.1](#).

### *3.6.10 Extending Java with code from other languages*

The Java Native Interface (JNI) supports interfacing with so-called “native methods” in a platform-independent way, which avoids many of the problems raised in [Section 2.11](#). JNI supports both Java calling native code and a native application calling Java, although in the context of testing and measurement the former is more likely. A tutorial on JNI is presented in <http://java.sun.com/docs/books/tutorial/native1.1/index.html>.

In principle, the native methods could be written in any language. However, because the methods must be able to interpret the type names used in the interface and make use of pointers to the Java environment and current objects that are passed by the method call, it is easiest to implement them in C or C++. The native code can then use the header files provided with the Java SDK or produced when compiling the Java code. A C wrapper function that obtains the necessary data and passes it on can be used to call other languages. This will certainly be necessary if using code from a binary library.

When native methods are used, the Java platform can no longer guarantee that downloaded applets will not interfere with the host machine, and so applets containing native code may be rejected by local security policies.

## 3.7 Delphi

### 3.7.1 Introduction

Delphi is a proprietary IDE from Borland that supports the programming language Object Pascal, an object-oriented extension of Pascal. Object Pascal was originally developed by Apple in conjunction with Niklaus Wirth, and has a variety of implementations although Delphi is the most commercially significant. Delphi was originally developed as a Windows system, but has recently been released in a version for Linux known as Kylix. Unlike Java, Delphi applications are sensitive to differences between the platforms such as file name conventions, but because Delphi programs are compiled to native code rather than interpreted there is no performance penalty in using it.

Object Pascal, like Java and C++ (but unlike Visual Basic) is a true object oriented language, with encapsulation, inheritance and polymorphism. Like Pascal, it is strongly typed, provides pointers, and requires explicit memory allocation and release. Like C++, it is possible to write programs without using objects: these are essentially Pascal programs, in the same way that C programs are legal C++. It is therefore a hybrid language rather than being purely object oriented.

The Delphi IDE allows the user to construct interactive applications by dragging and dropping controls onto forms, adjusting their visual properties, and linking the events they generate to handler code. Controls may be defined either using Borland's proprietary Visual Component Library (VCL) conventions or as Microsoft ActiveX controls. Many vendors provide laboratory instrument controls as ActiveX components and some also provide them in VCL format. The usual range of generic controls is included as part of Delphi.

A personal use version of Delphi is available for free download, while enriched versions designed for professional and enterprise use are sold as commercial products.

Borland supplies a substantial body of documentation on-line, and there are also a number of third party Delphi sites.

### 3.7.2 Requirements description

The general guidance in [Section 2.3](#) applies.

### 3.7.3 Design

The structure of an Object Pascal GUI program built with Delphi or Kylix is very much like that of a Visual Basic program. There is a main form, where the visual design of the GUI is recorded, with a corresponding unit (the Object Pascal equivalent of a module) where the event handlers are defined. For a simple application this is all that is needed. More complex applications may define a number of subforms, and may also have units with no associated forms that contain the definitions for the underlying model that the GUI manipulates. Controls are placed by dragging them onto the interface from a palette of available options. The IDE will construct a code skeleton for each event handler that is required, but the user must fill in the details of the action to be taken. Controls on the form are explicitly represented by objects of the appropriate type declared in the unit.

You should ensure that the controls are given meaningful names to replace the defaults created by the IDE as they are added to the user interface.

### *Timing*

Because Delphi produces native code from the Object Pascal source, execution speeds should be comparable with C and C++. There is third party support for developing device drivers for custom devices, even in difficult environments such as Windows NT. However, operating system limitations will probably require the use of a dedicated DAQ card at high data rates. Multiple threads are possible in Delphi, in much the same way as they are in C++, but are not as tightly integrated into the languages as they are in Java.

Delphi includes a timer control that can be used as a top-level event driver for polling input devices.

### *Reuse of components*

Object Pascal is unusual amongst object oriented languages in that class definitions are not stand-alone components of the language. Instead one or more classes will be encapsulated in a unit, in the same way that Java groups classes in packages. Both languages require explicit mention of any units used by another definition. In practice, reuse is most readily managed by packaging code into a control or library.

The community of Delphi developers is smaller than that for C++ or Java, but large enough to offer a choice of components for common requirements (database support, import from and export to standard applications, and so on) as off-the-shelf items.

### *User interfaces*

A general discussion of interface design issues can be found in [Section 2.13](#).

### *Debugging*

The Delphi IDE provides the usual code debugging capabilities, including conditional breakpoints based on variable values, watches on variable values, variable value display and so on.

As usual, any program requiring a real time response to input devices is not amenable to breakpoint-based debugging, but Delphi provides logging facilities so that actions can be tracked and assessed retrospectively. Debugging can be applied simultaneously to several processes.

## *3.7.4 Coding*

### *Coding standards*

See [Section 2.5](#) for general guidance on code development.

There are a number of suggested standards for Object Pascal coding, and some on-line examples are listed in [Appendix A.7.2](#). In general, they tend to deal with issues of naming and layout, rather than identifying areas of the language that are to be avoided as particularly error-prone. They often suggest conventions for Hungarian notation, the practice of prefixing a variable name with an indication of its type that is often adopted in Visual Basic programs. This is generally agreed to be less necessary in strongly typed programming languages, and tends to generate strong feelings for and against.

The Delphi IDE generates a lot of code automatically, and this will follow Borland's layout conventions. There are other editors and source code formatters that will also generate code following conventions. However, given the absence of language-oriented rules, there is also an absence of automatic rule checkers of the kind found for C/C++ and Java.

Like Java, Object Pascal uses a throw-and-catch approach to handling errors. Exception handling is written using a `try {<block>} except {<exception handlers>}` statement. If an exception is thrown in the block within the `try` part, its code is matched against the handlers in the `catch` part. If there is a match, the exception is handled by the corresponding code, while other exceptions continue to propagate up through the calling structure of the program. Exceptions can be thrown by fundamental operations such as division by zero, library code, or the user's own code. This method is much less prone to exception handling errors than setting error handlers procedurally, as in Visual Basic.

#### *Coding and software documentation*

General guidance on software documentation issues is given in [Section 2.5.3](#).

The obvious places to document Object Pascal code are at the starts of units and procedures. The coding standards referenced above usually include recommendations for commenting.

#### *3.7.5 Verification and validation*

General guidance on V&V is given in [Section 2.6](#).

There are a number of tools that support verification and validation. We have already mentioned the limited tool support for coding standards, and in particular there seems to be little help in static analysis of code and data flow. However there are tools for test execution and coverage measurement, and for more specialised tasks such as source code profiling and resource leak tracking. There appears to be less available for test generation than in the case of C/C++ or Java. Both commercial and community tools are available. A short list of examples is given in [Appendix A.7.3](#), and links to other tools are included at the general sites listed in [Appendix A.7.1](#).

#### *3.7.6 Maintenance*

General guidance on maintenance is given in [Section 2.7](#).

As with Java and C++, tools are available for extracting structural information from existing code.

#### *3.7.7 Configuration management*

General guidance on configuration management is given in [Section 2.8](#).

#### *Manual configuration control*

Unlike many other IDEs, a Delphi project is defined by a file that contains the main program and references those files that it imports explicitly in the code. It is thus easy to determine whether any absolute paths need changing and change them appropriately when generating a new version.



### *Tool supported configuration control*

Delphi can be extended with third party software to provide version control support within the IDE (see [Appendix A.7.3](#)).

### *Distributing applications*

Like Java, Delphi integrates all imported controls into the application, so applications are straightforward to distribute (provided ActiveX controls are not used). Tools are available for analysing dependencies between units, and for generating installers for applications (see [Appendix A.7.3](#)).

### *3.7.8 Metrics*

General guidance on metrics is given in [Section 2.9](#).

There are Delphi tools that will report complexity metrics for code (see [Appendix A.7.3](#)).

### *3.7.9 Further resources*

Delphi has an active user community that is well represented on the Web. A few major sites are listed in [Appendix A.7.1](#). There are also books on Delphi and Object Pascal.

### *3.7.10 Extending Delphi with code from other languages*

Borland Object Pascal supports the use of foreign language procedures in a partially platform-dependent way. Both the Linux and Windows versions support VCL components. Delphi can make use of ActiveX controls and DLL libraries, while Kylix can incorporate ELF format object files produced by other compilers including GCC. Strong type checking is maintained up to a point because the procedures, their parameters and their results must be declared as external, but there is no check that the description given matches the definition. Procedure names must be copied carefully, with attention to the cases of the characters, and parameter types must also be matched carefully. Mistakes will cause run-time errors that might not be revealed in some cases.

Object Pascal supports a number of calling conventions through keywords in the definitions of external procedures. For calling functions in DLLs that conform to the Windows API conventions, the appropriate choice is `stdcall`, as in:

```
function S18AEF(var X: Double;  
var IFAIL: Integer): Double;  
stdcall;  
external 'nagsx.dll' ;
```

which shows a call to a NAG Fortran DLL procedure. Object Pascal passes its parameters by value unless, as here, they are qualified by the keyword `var`, in which case they are passed by reference as expected by Fortran.

This declaration shows how a single external procedure would be defined, but Object Pascal also allows the definition of units containing many external procedures that can then be imported wherever they are needed, which is the most effective way to define the connection to a DLL. With this form of declaration, the compiler handles the loading of the DLL with the



application (but will not report a failure to load). It is also possible to load DLLs explicitly and check the returned result.

An example of calling a Fortran library subroutine from Delphi is given in [Appendix E.2](#).

### 3.8 MATLAB

The MATLAB environment integrates mathematical computing, visualisation, and a powerful technical language. A built-in interface enables the access and import of instrument data. In the consultations preceding the development of this guide, MATLAB was identified as being less commonly used for T&M instrument development as the languages discussed above, so this section addresses just the integration of external programs.

MATLAB provides a means to call Fortran (and C, C++ and Java) routines through “gateways”. At their simplest, these gateways call the MATLAB API to copy their input arguments from MATLAB storage to Fortran variables and arrays, call the Fortran routine and finally copy the results back into MATLAB storage.

Gateways are notoriously unforgiving when mistakes are made and can be very difficult to debug. It is recommended that all arguments are checked in the gateway to ensure that the arguments have a valid data type and structure. Input array arguments should be also checked to ensure they are the right size, where possible. The consistency of all data should be checked where possible. The aim of verification is, obviously, to ensure that the input data in MATLAB matches the Fortran storage allocated and the dimensions passed to the MATLAB API copy functions, and vice versa for the output arguments.

The example Gateway provided later in this document clearly illustrates that its production is a non-trivial programming task if undertaken from scratch. Fortunately some commercial tools, for example the NAGWare Gateway Generator (see [17]), ease the task considerably. These tools produce the necessary gateways by analysing the source code, Fortran in the case of the NAGWare product, and producing gateways from the information obtained.

It should be noted that this interface to MATLAB may not be viewed strictly as mixed language programming as, in later examples, all of the coding is actually performed in Fortran. However the MATLAB environment, which enables the execution of the subroutine, can be viewed as the non-Fortran component in the application as, just as with conventional mixed language programming, certain interfacing rules must be followed precisely. Indeed the similarity is reinforced as the compiler, to be used to compile the Gateway and the subroutine, must be able to generate a calling convention compatible with MATLAB. As a consequence not all compilers will be able to compile codes that can be accessed from within MATLAB.

An example of calling a Fortran library subroutine from MATLAB is given in [Appendix E.6](#).

## Appendix A Internet resources

The Internet is probably the broadest and most comprehensive collection of information resources regarding computers and software. Most of this information is available free, and increasingly software and content vendors are using the Internet to provide a distribution channel for their products.

One of the main problems in using the Internet is finding those resources of whose existence you are unaware. Probably the easiest way is to use some of the Internet search engines such as Google (<http://www.google.com/>), AltaVista (<http://www.altavista.com/>) or HotBot (<http://www.hotbot.com/>).

Other key resources include Web directories such as Yahoo! (<http://www.yahoo.com/>) which attempt to provide some structure to key resources in the form of categories.

Apart from the many useful links provided here, we recommend that you familiarise yourself with Internet search engines and large Web directories. Many of the sites listed below also collect related links, which will provide further points for your own exploration.

### A.1 User interface design

The following links are good starting points for finding out more about good interface design practices.

- <http://www.useit.com/> Jakob Nielsen's Web site on usability
- <http://www.iarchitect.com/fame.htm> Isys interface Hall of Fame and Shame
- <http://www.asktog.com/> Bruce Tognazzini's Web site on usability and interface design
- <http://www.baddesigns.com/> BadDesigns.com

### A.2 Component libraries on the Internet

ActiveX is a key technology for building Windows applications. There are a number of Web sites that collate ActiveX controls. Some controls are user interface widgets, others are for data processing. Often component vendors provide evaluation copies of software for download.

- <http://www.vbxtras.com/productsearch.asp> VBXtras is a tools catalogue for Visual Basic
- <http://www.componentsource.com/> ComponentSource; "The definitive source of software components", a UK based reseller of third party components
- <http://www.activex.com/> ActiveX.com; large repository of ActiveX components (mostly freeware and shareware)

- <http://www.greymatter.co.uk/> Greymatter; independent supplier of software and consultancy services to the programming and Web development community
- <http://www.pparadise.com/> Programmers' Paradise; another online component repository
- <http://www.ni.com/cworks/> National Instruments' "ComponentWorks" suite of ActiveX instrument controls
- <http://www.globalmajic.com/> Global Majic provide their own collection of instrumentation controls

### A.3 C/C++ resources

- Microsoft Visual C++ home page: <http://msdn.microsoft.com/visualc/>
- Microsoft Foundation Classes Development guidelines (contains useful information on C++ generally):  
[http://msdn.microsoft.com/library/backgrnd/html/msdn\\_mfcgde.htm](http://msdn.microsoft.com/library/backgrnd/html/msdn_mfcgde.htm)
- Association of C and C++ users: <http://www.accu.org/> (but you have to pay to join)
- General C resources: <http://dmoz.org/Computers/Programming/Languages/C/>
- C FAQs: <http://dmoz.org/Computers/Programming/Languages/C/FAQs/>
- General C++ resources: <http://dmoz.org/Computers/Programming/Languages/C++/>
- Motor Industry Software Reliability Association: <http://www.misra.org.uk/>
- LabVIEW CVI: <http://www.ni.com/cvi/>

### A.4 Visual Basic resources

#### A.4.1 General sites for VB developers

- <http://msdn.microsoft.com/vbasic/> Microsoft Developers Network Visual Basic site
- <http://www.vb-helper.com/> VB helper, a very good Web site for Visual Basic (especially /links)
- <http://www.freevbcode.com/> Free VB Code, contains a wealth of controls, sample applications and snippets to download

- <http://www.vb-web-directory.com/> VB WebDirectory—a Web directory of VB-related links
- <http://www.geocities.com/SiliconValley/Pines/4038/dev/vb.htm> A collection of VB-related links
- <http://www.dcs.napier.ac.uk/hci/VB50/home.html> Introduction to Visual Basic 5.0 Programming, an online tutorial on VB5

#### A.4.2 Visual Basic coding standards

- <http://support.microsoft.com/support/kb/articles/q110/2/64.asp> Q110264 Microsoft Consulting coding standard
- <http://www.xoc.net/standards/> Reddick VBA naming standards; widely used and referred to
- [http://www.gui.com.au/html/coding\\_standards.htm](http://www.gui.com.au/html/coding_standards.htm)

### A.5 LabVIEW resources

Some key LabVIEW Internet resources are listed below:

- National Instruments' LabVIEW site <http://www.ni.com/labview>
- LabVIEW listserv (email discussion list) and general LabVIEW users' site: <http://www.info-labview.org/>
- LabVIEW webring: <http://www.webring.org/cgi-bin/webring?ring=labview;list>

### A.6 Java resources

#### A.6.1 General sites for Java developers

- <http://java.sun.com/>. A large collection of official Java resources, including downloads of Java implementations.
- <http://www-106.ibm.com/developerworks/java/>. An IBM-supported site containing tutorials, how-tos and pointers to Java tools.
- <http://www.jroundup.com/>. Another collection of tutorials, articles and downloads supported by a number of companies with an active role in Java development.

#### A.6.2 Java coding standards

- <http://java.sun.com/docs/codeconv/>. The official Sun Java coding conventions.

- <http://www.ambysoft.com/javaCodingStandards.pdf>. Largely about naming and documentation, but with good advice on a broad range of issues. Also available in book form [16]
- <http://www.geosoft.no/javastyle.html>. A compact list of recommendations with associated rationale.
- <http://www.mindprod.com/gotchas.html>. A list of common pitfalls in Java.

### A.6.3 Java tools

The following is not intended as an exhaustive list, or as an endorsement of any of the tools mentioned, but purely as an illustration of the range of tool support available. A more complete list is available at <http://industry.java.sun.com/solutions>.

- <http://www.borland.com/jbuilder/>. Describes the Borland JBuilder IDE.
- <http://www.jenssoft.com/>. A coding standards compliance checker with a fixed set of coding conventions.
- <http://www.mmsindia.com/jstyle.html>. A coding standards compliance checker and metric calculator.
- <http://www.parasoft.com/jsp/products/home.jsp?product=Jtest&/>. A test generator and static analyser.
- <http://www.testersedge.com/gjtk/>. A test coverage measurement tool
- <http://java.sun.com/j2se/javadoc/>. Describes a tool for generating HTML documentation from comments and added tags in the Java code.
- <http://research.compaq.com/SRC/esc/>. A research tool that detects a significant class of Java programming errors by static analysis rather than testing, using a limited form of semantic analysis.
- <http://www.scitools.com/uj.html>. A tool for browsing and extracting structure from existing Java programs.
- <http://www.refactorit.com/>. A tool for refactoring, and a discussion of the nature and benefits of the technique.

## A.7 Delphi resources

### A.7.1 General sites for Delphi developers

- <http://www.inner-smile.com/delphi5.htm>. Top page of a large independent Delphi resource, with many links to other sites and some local FAQs and other information.

- <http://www.simplythebest.net/info/delphi.html>. One page on an independent site with many links to tools, tutorials, and example source code.

#### A.7.2 Delphi coding standards

- <http://community.borland.com/soapbox/techvoyage/article/1,1795,10280,00.html>. The conventions for layout and naming (together with a few coding style recommendations) that Borland uses for the Delphi code it distributes.
- <http://www.econos.de/delphi/cs.html>. Mainly concerned with naming and layout, but with some pointers to correct and readable code.
- <http://www.ocdelphi.org/standard.htm>. Widely referenced coding standard, again mostly concerned with naming and layout.

#### A.7.3 Delphi tools

- <http://www.preview.org/e/scintro.htm>. The SourceCoder tool formats source code, calculates metrics, and instruments code for test coverage and profiling measurements.
- <http://www.automatedqa.com/products/memproof.asp>. A memory and resource leak debugger. A vital tool where storage is managed explicitly in the program
- <http://www.prodelphi.de/>. A Delphi profiler: also available in a Linux version for Kylix.
- <http://ourworld.compuserve.com/Homepages/wviahmann/eng.htm>. A tool to generate structure diagrams from source code.
- <http://dunit.sourceforge.net/>. An open source testing framework for Delphi programs.
- [http://www.inner-smile.com/dl\\_inf.htm](http://www.inner-smile.com/dl_inf.htm). A tool for generating installation scripts for applications.
- <http://jvcl.sourceforge.net/>. A repository of open source VCL components for Delphi.
- <http://www.freevcs.org/j/index.htm>. Version control that can be integrated with Delphi.

### A.8 Other T&M software technologies

There are other specialist technologies for building T&M software. An illustrative selection is given below:

- Quatech designs and manufactures communication, data acquisition and signal conditioning PCMCIA, ISA and PCI adapters for PCs. See <http://www.quatech.com/>

- Pico Technology Ltd are manufacturers of PC based test & measurement equipment, PC based oscilloscopes, spectrum analysers, data acquisition, temperature and environmental monitoring devices. See <http://www.picotech.com/>
- Windmill Software Limited specialise in data acquisition and control software for Windows. See <http://www.windmill.co.uk/windmill.html>
- Superlogics develop WinViewLE a free software tool for building simple DAQ applications. See <http://www.superlogics.com/>

## A.9 General resources

- For information on UML see <http://www.uml.org/>
- For information on the Rational tools, see <http://www.rational.com/>
- For information on formal methods and safety critical software development generally, see <http://archive.comlab.ox.ac.uk/safety.html>
- For general links on safety related computer systems, see <http://www.adelard.com/>
- For general guidance on the use of formal methods, see <http://www.ewics.org/>
- For information on the PolySpace static analysis tool for run-time exception checking, see <http://www.polyspace.com/>
- For information on specific formal model checkers, see <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv> (SMV) and <http://www-step.stanford.edu/> (STeP)
- For information on the Dynamic Systems Development Methods (DSDM), see <http://www.dsdm.org/>
- NPL's compact quality manual for T&M software projects, with a tool for producing a software quality plan based on a risk assessment, is available from [http://www.npl.co.uk/ssfm/theme3/project3\\_2/procedure.html](http://www.npl.co.uk/ssfm/theme3/project3_2/procedure.html)
- For information on Bugzilla, see <http://www.mozilla.org/projects/bugzilla/about.html>. Other general links for bug tracking software can be found at [http://dmoz.org/Computers/Software/Configuration\\_Management/Bug\\_Tracking/Fre](http://dmoz.org/Computers/Software/Configuration_Management/Bug_Tracking/Fre)  
[e/](http://dmoz.org/Computers/Software/Configuration_Management/Bug_Tracking/Fre)



## Appendix B Books and references

### B.1 Books

#### B.1.1 LabVIEW

- [1] Gary Johnson, *LabVIEW Graphical Programming*, 2<sup>nd</sup> Edition, McGraw Hill, 1997. ISBN 0-07-032915-X. (With a CD-ROM including a style guide, *LabVIEW with Style*.)
- [2] *Instrupedia*, CD-ROM from National Instruments.
- [3] LabVIEW technical resource, periodical publication on CD-ROM of technical articles, published by LTR publishing, <http://www.ltr.com/>
- [4] Alliance Solutions, Directory of third Party Products and services, published annually by National Instruments.

#### B.1.2 Visual Basic

- [5] Rod Stevens: *Bug Proofing Visual Basic: A Guide to Error Handling and Prevention*, John Wiley & Sons, 1998. ISBN 0471323519.
- [6] John Clark Craig, Webb Craig, Jeff Webb *Microsoft Visual Basic 6.0 Developer's Workshop*, Microsoft Press, 1998. ISBN: 157231883X.
- [7] Jan Axelson & Janet Axelson, *Parallel Port Complete*, Peer-to-Peer Communications, 1997. ISBN 0965081915.
- [8] Jan Axelson & Janet Axelson *Serial Port Complete*, Lakeview Research, 1998. ISBN 0965081923.
- [9] Grier et al: *Visual Basic Programmers' Guide to Serial Communications*, Mabry Software Inc, 1997. ISBN 1890422258.
- [10] *Microsoft Visual Basic 5.0 Programmer's Guide* Microsoft Corporation ISBN: 1-57231-604-71997

#### B.1.3 C/C++

- [11] Campbell J, *C Programmers' Guide to Serial Communications*, Sams, 1993. ISBN 0672302861.
- [12] Dhananjay V. Gadre, *Programming the Parallel Port; Interfacing the PC for Data Acquisition and Process Control*, R&D Books. ISBN 0879305134.
- [13] Les Hatton, *Safer C: Developing Software for High Integrity and Safety Critical Systems*, McGraw Hill, 1995. ISBN 0-07-707640-0. See also <http://www.oakcomp.demon.co.uk/SaferC.html>
- [14] A Koenig, *C Traps and Pitfalls*, Addison Wesley, 1988.

- [15] *MISRA Development Guidelines for Vehicle based Software*, The Motor Industry Software Reliability Association, MIRA, 1994.

#### B.1.4 Java

- [16] *The Elements of Java Style*. Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur & Thompson. Cambridge University Press, 2000. ISBN: 0-521-77768-2.

#### B.1.5 MATLAB

- [17] NAGWare Gateway Generator—Numerical Algorithms Group Ltd, Jordan Hill Road Oxford OX2 8DR.

#### B.1.6 General

- [18] N E Fenton and S L Pleegeer, *Software Metrics: A Rigorous and Practical Approach*, 2nd edition, International Thomson Computer Press, 1996.
- [19] Ian Sommerville and Pete Sawyer, *Requirements Engineering*, John Wiley and Sons, 1997. ISBN 0471974447.
- [20] IChemE, *The Engineer's Responsibility for Computer Based Decisions*, The Institution of Chemical Engineers, Geo. E. Davis Building, 165–171 Railway Terrace, Rugby, CV21 3HQ, UK.
- [21] IEC 1131-3 Programmable controllers, International Electrotechnical Commission, 1992.
- [22] T Gilb and D Graham, *Software Inspection*, Addison-Wesley, 1993. ISBN 0201631814.
- [23] BS 7925, *Software testing*. Part 1: 1998, *Vocabulary*; Part 2: 1998, *Software component testing*.
- [24] Ian Sommerville, *Software Engineering*, 5<sup>th</sup> Edition, Addison-Wesley, 1995. ISBN 0-201-42765-6.
- [25] J A McDermid (editor), *Software Engineer's Reference Book*, Butterworth-Heinemann, 1991. ISBN 0-750-961040-9.
- [26] IEC 61508, Parts 1–7, *Functional Safety: Safety-Related Systems*, 1999.
- [27] DMOZ entry on software testing:  
[http://dmoz.org/Computers/Programming/Software\\_Testing/](http://dmoz.org/Computers/Programming/Software_Testing/)
- [28] G T Anthony and M G Cox, “The National Physical Laboratory's Data Approximation Subroutine Library”, in J C Mason and M G Cox (editors), *Algorithms for Approximation*, Clarendon Press, Oxford, UK, 1987.
- [29] G Booch, *Object-oriented Analysis and Design with Applications*, Benjamin Cummings, 1993, ISBN 0805353402.
- [30] *Rapid Application Development (RAD) issue*, DoD Software Tech News Volume 2, No 1., 1998 (see also <http://www.dacs.dtic.mil/awareness/newsletters/>)

- 
- [31] *Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP)*, Report No: CRR337, HSE Books 2001. ISBN 0 7176 2011 5.  
[www.hse.gov.uk/research/crr\\_pdf/2001/crr01336.pdf](http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf)
- [32] *Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications*, Report No: CRR336, HSE Books 2001. ISBN 0 7176 2010 7.  
[http://www.hse.gov.uk/research/crr\\_pdf/2001/crr01337.pdf](http://www.hse.gov.uk/research/crr_pdf/2001/crr01337.pdf)
- [33] *Dynamic Systems Development Method, version 2*, Tesseract Publishing, Farnham, 1995.
- [34] J. Stapleton, *Dynamic Systems Development Method: The method in practice*, Addison Wesley, 1997.
- [35] R Woodhead, M Atkinson, J Stapleton, M Bray and M Blackman, *Dynamic Systems Development Method and TickIT*, DISC TickIT Office, British Standards Institution, London, 1997.
- [36] *Adelard Safety Case Development Manual*. Adelard, 1998. ISBN 0 9533771 0 5. See <http://www.adelard.co.uk/resources/ascad/>.

### B.1.7 Related guides

- [37] Cathy Thomas, Owen Daly-Jones and Andrew Harry, *Measurement Good Practice Guide No. 8: Human factors in Measurement and Calibration*, National Physical Laboratory, 1998.
- [38] Brian Wichmann, *Software Support for Metrology Best Practice Guide No. 1: Measurement system validation: Validation of measurement software—Revision for safety systems*, Draft. See [http://www.npl.co.uk/ssfm/download/documents/ssfmbpg1\\_draft.pdf](http://www.npl.co.uk/ssfm/download/documents/ssfmbpg1_draft.pdf)
- [39] *Software Support for Metrology Best Practice Guide No. 2: The Development of Virtual Instruments*. <http://www.npl.co.uk/ssfm/download/documents/ssfmbpg2.pdf>
- [40] *Software Support for Metrology Best Practice Guide No. 3: Developing software for metrology*. <http://www.npl.co.uk/ssfm/download/documents/ssfmbpg3.pdf>
- [41] *Software Support for Metrology Best Practice Guide No. 8: Mixed Language Programming*. <http://www.npl.co.uk/ssfm/download/documents/ssfmbpg8.pdf>
- [42] R Barker, *Software Support for Metrology Best Practice Guide No. 5: Software Re-use: Guide to METROS*. See also <http://www.npl.co.uk/ssfm/download/documents/ssfmbpg5.pdf>

### B.2 Other references

- [43] The NAG Fortran library, The Numerical Algorithms Group Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK.
- [44] J J Dongarra, C B Molder, J R Bunch and G W Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.



## Appendix C Visual Basic example

For the Visual Basic implementation of the traffic light example, using the specification from [Section 2.4](#). The main monitoring loop (which is called by a top-level timer control) is listed below.

The main structures of the code that map to the state transition diagram (see [Figure 7](#)) are marked in bold and labelled according.

The program and the source code can be downloaded from Adelard's web site, <http://www.adelard.com/>

```

Sub CheckWorldState()
  'function: This is the overall monitoring state machine.
  ' Refer to the State machine specification for full details.
  ' Basically, requests to open or close the barrier are considered
  ' every time the monitor loop is run, but there is an interlock
  ' whereby the barrier will not close if there is a car in the
  ' barrier zone
  'called by: MonitorWorldTimer
  'history: 17/9/99 LOE created procedure

  'safe to open barrier at any time
If blnRequestOpen Then
  Me.txtBarrierstate = "...opening barrier..."
  Me.txtBarrierstate.Refresh

GotolightState (2)

  Me.txtBarrierstate = "Barrier open"
  blnRequestOpen = False 'no need to request anymore

  'the following line implements slightly different behaviour
  'if it is included or commented out.
  'If included then close requests can be cancelled if there is a
  car in the zone
  'otherwise if commented out then barrier will close eventually if
  the request close
  'is pushed.
  blnRequestClose = False
Else
  'is there a car in the barrier zone?
If blnVehicleInZone Then
  If blnRequestClose Then
    Me.txtBarrierstate = "...waiting for vehicle to pass..."
  Else
    Me.txtBarrierstate = "...vehicle passing...vrm..vrm"
  End If
Else
  'no car in zone so we can close barrier
If blnRequestClose Then
  Me.txtBarrierstate = "...barrier closing..."
  Me.txtBarrierstate.Refresh

```

STEP 1 & 2

STEP 5 & 6

GotolightState (0)

Me.txtBarrierstate = "Barrier closed"

**blnRequestClose = False** *'no need to request anymore*

**Else**

*'no open or close requests in this loop,*

*'so barrier stays in its current alignment*

If iCurLightState = 2 Then Me.txtBarrierstate = "Barrier open"

If iCurLightState = 0 Then Me.txtBarrierstate = "Barrier closed"

**End If**

**End If**

**End If**

**End Sub**

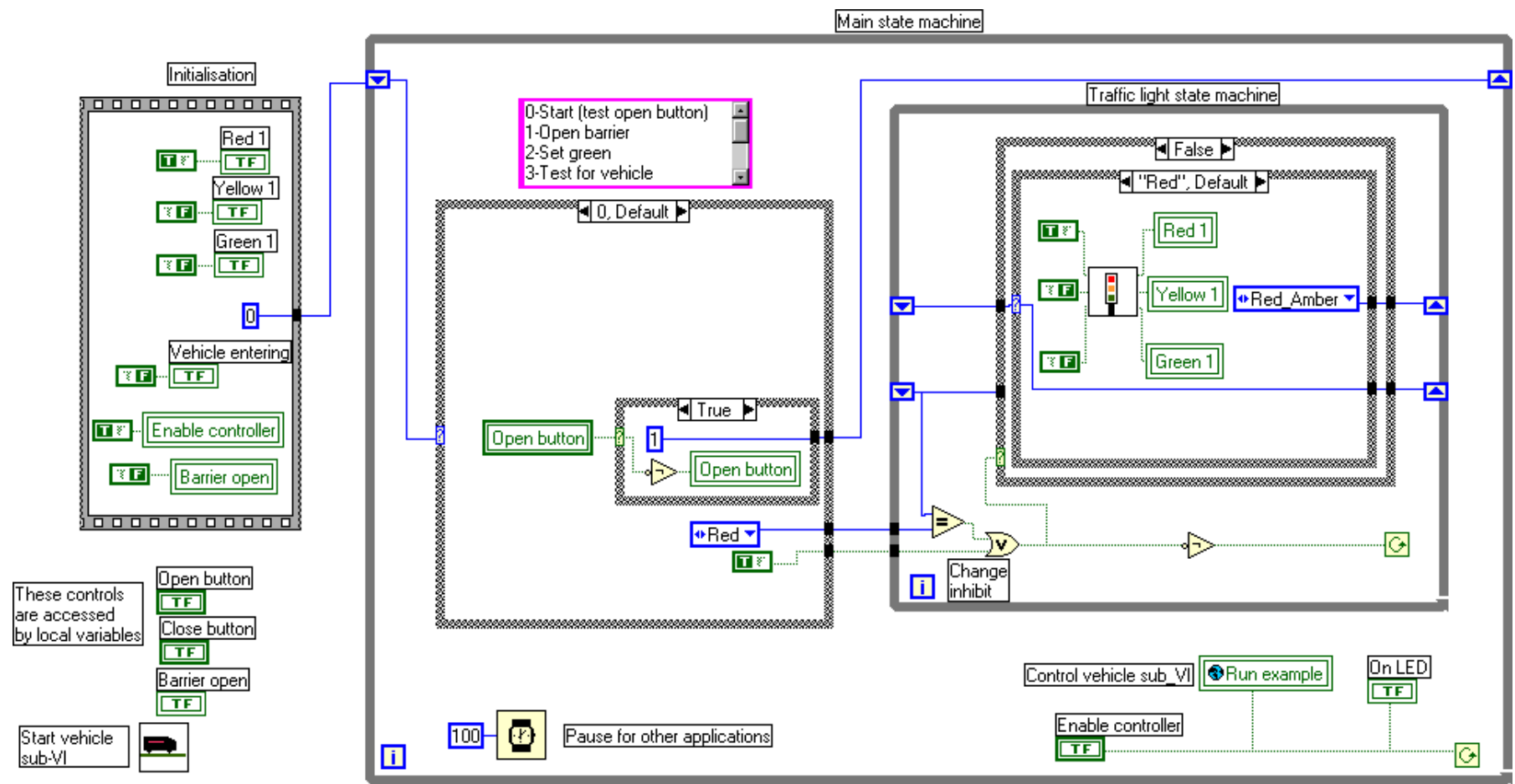
## Appendix D LabVIEW example

This appendix shows the top-level diagram for the site entry barrier example. It shows the initialisation phase, followed by the main state machine (the While Loop and the left-hand Case structure), which determines how to set the barrier on the basis of the buttons and the vehicle sensor. The state machine on the right computes the traffic light aspects.

The complete example includes a free-running sub-VI that models the behaviour of a vehicle passing the barrier when the traffic light is green.

Note the way the diagram is documented by areas of text and a scrolling string constant.

The complete VI hierarchy can be downloaded from Adelard's Web site, <http://www.adelard.com/>





## Appendix E Examples of mixed language programming

In this appendix we provide some example procedures illustrating mixed language programming, specifically calling the NAG Fortran libraries from C++, Delphi, Visual Basic, LabVIEW and MATLAB, and C libraries from Visual Basic. They are taken verbatim from SSfM-1 Best Practice Guide 8.

### E.1 Calling a Fortran DLL from Microsoft Visual C++

The following illustrates how the interface to a NAG Library routine that has multi-dimensional arrays and character strings in the user interface. The example is taken from a PC using the CVF and Microsoft Visual C/C++ compilers.

There are in addition different conventions adhered to by the C language from that of Fortran. For example, Fortran routines assume that array indices start at 1, whereas the usual C convention is that indices start at 0. Care must also be taken with two-dimensional arrays. Fortran conventions stipulate, for example, that element (2,1) is next to element (1,1) in the physical storage of the array. C convention has element (1,2) next to (1,1) (i.e. Fortran stores by column, C by row). The result is that the C programmer often has to provide the transpose array to Fortran routines and to interpret returned array information accordingly.

C character strings and Fortran character strings are handled differently. In the CVF implementation a string argument is passed by value as a structure consisting of a 4-byte argument address, followed by the length of the string.

The following example illustrates the use of the NAG DLLs using Microsoft Visual C++, version 4.2.

```
#include <stdio.h>
#define CONST const
typedef struct { char *str; int val; } Fortran_character_arg;

extern void __stdcall F01QCF(
CONST int *m,
CONST int *n,
double a[ ],
CONST int *lda,
double zeta[ ],
int *ifail
);

extern void __stdcall F01QDF(
CONST Fortran_character_arg trans,
CONST Fortran_character_arg wheret,
CONST int *m,
CONST int *n,
double a[ ],
CONST int *lda,
CONST double zeta[ ],
CONST int *ncolb,
double b[ ],
CONST int *ldb,
double work[ ],
int *ifail
);
```

```
/* Simplified example program for F01QDF */

main( )
{
  int i,ifail,j,m = 5,n = 3,ncolb = 2;
  int lda = m,ldb = m;
  Fortran_character_arg tr = {"Transpose",9};
  Fortran_character_arg se = {"Separate",8};
  /* Initialise arrays in column major order */
  static double a[3][5] =
  {
    2.0, 2.0, 1.6, 2.0, 1.2,
    2.5, 2.5, -0.4, -0.5, -0.3,
    2.5, 2.5, 2.8, 0.5, -2.9
  };
  static double b[2][5] =
  {
    1.1, 0.9, 0.6, 0.0, -0.8,
    0.0, 0.0, 1.32, 1.1, -0.26
  };
  double work[2],zeta[3];

  printf("F01QDF Example Program Results\n\n");
  ifail = 0;
  F01QCF(&m,&n,(double *)a,&lda,zeta,&ifail);
  ifail = 0;
  F01QDF(tr,se,&m,&n,(double *)a,&lda,zeta,&ncolb,(double
  *)b,&ldb,work,&ifail);
  printf("Matrix Q'*B\n");
  for (i=0; i<m; i++)
  {
    for (j=0; j<ncolb; j++)
    {
      printf("%f ",b[j][i]);
    }
    printf("\n");
  }
  return 0;
}
```

## E.2 Calling a Fortran Subroutine from Delphi

As Borland Delphi is a PC product the general rules for best practice may be relaxed provided that Fortran/Delphi inter-calling is the only consideration. The following examples demonstrate how Delphi can be used to call routines from the NAG Library DLL implementation.

One important point to bear in mind when calling the NAG DLLs from Delphi is that the actual parameters must be of type var. This is because the Fortran calling convention requires parameters to be passed by reference and not by value. It is not necessary to include the library itself in the compilation linker list in Delphi; the DLL can be called straight from the code itself and the compiler will link it automatically.

The reference to the DLL is as a procedure or function, defined as external in the Delphi code. This procedure needs to have the same name as the DLL routine called. Delphi is case sensitive, so the NAG name must be in capital letters (the Delphi name construct may be used to change this if desired).

Consider the following example:

```
function S18AEF(var X: Double;
var IFAIL: Integer): Double;
stdcall;
external 'nagsx.dll';
```

The `stdcall` directive is required to ensure that the right-to-left calling convention is used, and to specify that the routine is responsible for cleaning up the stack and not the program. The Fortran DLL expects this calling convention. The function or procedure can then be called as a normal routine, e.g.:

```
WriteLn(S18AEF(X, IFAIL));
```

### *E.2.1 Multi-Dimensional Array*

Arrays of more than one dimension have to be transposed before they can be passed to the Fortran DLL. This is because Fortran assumes that an array such as `A[2,2]` is stored in contiguous locations in column order, i.e. as `A[1,1]`, `A[2,1]`, `A[1,2]`, `A[2,2]`. Pascal, on the other hand, stores the elements in row order as `A[1,1]`, `A[1,2]`, `A[2,1]`, `A[2,2]`. Note that the Pascal arrays handed as actual parameters to a Fortran DLL must be defined as data types as in the example. A Pascal variable array defined in the `var` section, passed as an actual argument, overwrites other parameter values and can cause system failure. See example of function/procedure passing for illustration of multi-dimensional array handling.

### *E.2.2 Passing Functions and Procedures*

Many of the NAG Libraries require subroutines and functions to be passed as parameters. To achieve this in Delphi each procedure or function to be passed needs its own data type, defined under the type heading. This allows it to be passed to the DLL through the parameter list. The type definition needs to have the same number and type of parameters as the subroutine itself. Note that `var` is not required since only a copy of the procedure needs to be made during the passing. Note that `stdcall` is required both on the definition of the function/procedure and on the definition of the datatype to ensure the correct calling convention is used, as before.

### *E.2.3 NAG Library Routine D03PCF Example Program Coded in Delphi*

This code uses the routine `D03PCF`, which integrates a system of linear or nonlinear parabolic partial differential equations (PDEs). This is found in the NAG DLL `NAGD03.DLL`. This program illustrates the use of multi-dimensional arrays and function/procedure passing as described above. Note that it also uses the external function `X01AAF`, found in `NAGSX.DLL`, to find `pi`.

```
unit D03Code;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, TForms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  TForm1: TForm1;

implementation

{$R *.DFM} {Compiler Directive}
```

```
type
U_ArrayType = array [1..20, 1..2] of Double;
UOUT_ArrayType = array [1..1, 1..6, 1..2] of Double;
{Note: the two arrays above are defined as the transpose of the parameter requirements
to ensure compatibility with Fortran DLLs.}

W_ArrayType = array [1..1128] of Double; {1..NW}
X_ArrayType = array [1..20] of Double; {1..NPTS}
XOUT_ArrayType = array [1..6] of Double; {1..INTPTS}
IW_ArrayType = array [1..64] of Integer; {1..NIW}
NPDE_ArrayType = array [1..2] of Double; {1..NPDE}
P_ArrayType = array [1..2] of NPDE_ArrayType;
PDEDEFType = Procedure(var NPDE: Integer;
var T: Double;
var X: Double;
var U: NPDE_ArrayType;
var DUDX: NPDE_ArrayType;
var P: P_ArrayType;
var Q: NPDE_ArrayType;
var R: NPDE_ArrayType;
var IRES: Integer);
stdcall;
BNDARYType = Procedure(var NPDE: Integer;
var T: Double;
var U: NPDE_ArrayType;
var UX: NPDE_ArrayType;
var IBND: Integer;
var BETA: NPDE_ArrayType;
var GAMMA: NPDE_ArrayType;
var IRES: Integer);
stdcall;
{The two types above are Procedure types. These need to be defined so that the
procedures
BNDARY and PDEDEF can be passed as parameters (of type procedure) to the DLL.}
var
NPDE: Integer = 2;
NPTS: Integer = 20;
INTPTS: Integer = 6;
ITYPE: Integer = 1;
NEQN: Integer;
NIW: Integer;
NWK: Integer;
NW: Integer;

I: Integer;
J: Integer;
IFAIL: Integer;

ALPHA: Double;
ACC: Double;
HX: Double;
PI: Double;
PIBY2: Double;
TOUT: Double;
TS: Double;
IND: Integer;
IT: Integer;
ITASK: Integer;
ITRACE: Integer;
M: Integer;
U: U_ArrayType;
UOUT: UOUT_ArrayType;
W: W_ArrayType;
X: X_ArrayType;
XOUT: XOUT_ArrayType = (0.0,0.4,0.6,0.8,0.9,1.0);
IW: IW_ArrayType;

Procedure D03PCF(var NPDE: Integer;
var M: Integer;
var TS: Double;
var TOUT: Double;
PDEDEF: PDEDEFType; {The two procedure parameters,}
```

---

```

BNDARY: BNDARYType; {defined above under type}
var U: U_ArrayType;
var NPTS: Integer;
var X: X_ArrayType;
var ACC: Double;
var W: W_ArrayType;
var NW: Integer;
var IW: IW_ArrayType;
var NIW: Integer;
var ITASK: Integer;
var ITRACE: Integer;
var IND: Integer;
var IFAIL: Integer);
stdcall;
external 'nagD03.dll';

Function X01AAF(var PI: Double): Double; stdcall;
external 'nagsx.dll';

Procedure D03PZF(var NPDE: Integer;
var M: Integer;
var U: U_ArrayType;
var NPTS: Integer;
var X: X_ArrayType;
var XOUT: XOUT_ArrayType;
var INTPTS: Integer;
var ITYPE: Integer;
var UOUT: UOUT_ArrayType;
var IFAIL: Integer);
stdcall;
external 'nagD03.dll';

{PDEDEF--to define the system of PDEs}

Procedure PDEDEF(var NPDE: Integer;
var T: Double;
var X: Double;
var U: NPDE_ArrayType;
var UX: NPDE_ArrayType;
var P: P_ArrayType;
var Q: NPDE_ArrayType;
var R: NPDE_ArrayType;
var IRES: Integer);
stdcall;
begin
Q[1]:= 4.0*ALPHA*(U[2]+X*UX[2]);
Q[2]:= 0.0;
R[1]:= X*UX[1];
R[2]:= UX[2]-U[1]*U[2];
P[1,1]:= 0;
P[1,2]:= 0;
P[2,1]:= 0;
P[2,2]:= 1.0-X*X
end;

Procedure BNDARY(var NPDE: Integer;
var T: Double;
var U: NPDE_ArrayType;
var UX: NPDE_ArrayType;
var IBND: Integer;
var BETA: NPDE_ArrayType;
var GAMMA: NPDE_ArrayType;
var IRES: Integer);
stdcall;
begin
if (IBND=0) then
begin
BETA[1]:= 0;
BETA[2]:= 1;
GAMMA[1]:= U[1];
GAMMA[2]:= -U[1]*U[2];
end
end

```

---

```
else
begin
BETA[1]:= 1;
BETA[2]:= 0;
GAMMA[1]:= -U[1];
GAMMA[2]:= U[2];
end
end;

Procedure SetUp;
var
I: Integer;
begin
NEQN:= NPDE * NPTS;
NIW:= NEQN+24;
NWK:= (10+6*NPDE)*NEQN;
NW:= NWK+(21+3*NPDE)*NPDE+7*NPTS+54;

ACC:= 1.0E-4;
M:= 1;
ITRACE:= 0;
ALPHA:= 1.0;
IND:= 0;
ITASK:= 1;

{Set spatial mesh points}
PIBY2:= 0.5*X01AAF(PI);
HX:= PIby2/(NPTS-1);
X[1]:= 0;
X[NPTS]:= 1;
for I:= 2 to (NPTS-1) Do
begin
X[I]:= SIN(HX*(I-1))
end;

{Set initial conditions}
TS:= 0.0;
TOUT:= 0.1E-4;
end;

{Unit defines the initial PDE condition}

Procedure Uinit(var U: U_ArrayType;
var X: X_ArrayType;
var NPTS: Integer);
var
I: Integer;
begin
for I:= 1 to NPTS Do
begin
U[I,1]:= 2.0*ALPHA*X[I];
U[I,2]:= 1.0;
end;
end;

begin
WriteLn('D03PCF--Example program results');
SetUp;
WriteLn;
WriteLn('Accuracy requirement = ',ACC);
WriteLn('Parameter alpha = ',ALPHA);
Write(' T / X ');
for I:= 1 to 6 Do
Write(XOUT[I]: 6);
WriteLn;

Uinit(U,X,NPTS);
for I:= 1 to 5 Do
begin
IFAIL:= -1;
TOUT:= 10*TOUT;
```

```

D03PCF(NPDE,M,TS,TOUT,PDEDEF,BNDARY,U,NPTS,X,ACC,W,NW,IW,N
IW,
ITASK,ITRACE,IND,IFAIL);

D03PZF(NPDE,M,U,NPTS,X,XOUT,INTPTS,ITYPE,UOUT,IFAIL);
WriteLn;
Write(TOUT: 6, ' U[1]');
for J:= 1 to INTPTS Do
Write(UOUT[1,J,1]: 5, ' ');
WriteLn;
Write(' U[2]');
for J:= 1 to INTPTS Do
Write(UOUT[1,J,2]: 5, ' ');
WriteLn;
end;
WriteLn('Number of integration steps in time',IW[1]);
WriteLn('Number of residual evaluations of resulting ODE
system ',IW[2]);
WriteLn('Number of Jacobian evaluations',IW[3]);
WriteLn('Number of iterations of nonlinear solver',IW[5]);
end.

```

#### *E.2.4 String Handling and Passing*

Several Fortran DLLs require strings or characters to be passed as parameters. The strings need to be null terminated, and defined either with Pchar, or as an array of characters such as:

```
strng = array [ 0.. 2 ] of Char;
```

The example program uses an array of characters for simplicity. Note that the array has to be zero based. The DLL routine will expect a zero based string array and will fail if it does not receive one.

The routines expect the length of the string in characters to be passed after the string itself. The best way to do this is to pass this as an extra integer parameter after each string or character, e.g.:

```

procedure G02EEF(...;
...;
var NAME: Strng_ArrayType;
NAME_Len: Integer;
...;
var NEWVAR: String;
NEWVAR_Len: Integer;
...);
stdcall;
external 'nagG02.dll';

and in calling:
G02EEF(..., NAME, 3, ..., NEWVAR, 3, ...);

```

The extra parameters can be added with no problems because an integer is expected after each string, character or array of strings, such as Strng\_ArrayType.

#### *E.2.5 NAG Library Routine G02EEF Example Program Coded in Delphi*

This code uses the routine G02EEF to carry out one step of a forward selection procedure to enable the “best” linear regression model to be found. This example was chosen to illustrate the problems that arise through passing strings to Fortran DLLs. It also includes another example of multi-dimensional array handling.

```
unit G02Code;
```

```
interface

uses
Forms;

type
TForm1 = class(TForm)
private
{ Private declarations }
public
{ Public declarations }
end;

var
Form1: TForm1;

implementation

{$R *.DFM}
{G02EEF-Example Program in Delphi 2}

type
X_ArrayType = array [1..8, 1..20] of Double;
{X Array, and Q Array below, are defined as the transpose of the parameter
requirements to ensure compatibility with Fortran DLL.}
Strng = array [0..2] of Char;
{A Null terminated string. Note the zero basing of the array of characters.}
Strng_ArrayType = array [1..8] of Strng;
ISX_ArrayType = array [1..8] of Integer;
WTY_ArrayType = array [1..20] of Double;
EP_ArrayType = array [1..9] of Double;
Q_ArrayType = array [1..10, 1..20] of Double;
WK_ArrayType = array [1..16] of Double;

var
I: Integer;
J: Integer;
NMAX: Integer = 20;
MMAX: Integer = 8;
ISTEP: Integer;
MEAN: Char;
WEIGHT: Char;
N: Integer;
M: Integer;
X: X_ArrayType;
NAME: Strng_ArrayType;
ISX: ISX_ArrayType;
Y: WTY_ArrayType;
WT: WTY_ArrayType;
FIN: Double;
ADDVAR: Boolean;
CHRSS: Double;
F: Double;
MODEL: Strng_ArrayType;
NTERM: Integer;
RSS: Double;
IDF: Integer;
IFR: Integer;
FREE: Strng_ArrayType;
EXSS: EP_ArrayType;
Q: Q_ArrayType;
LDQ: Integer;
P: EP_ArrayType;
WK: WK_ArrayType;
IFAIL: Integer;
NEWVAR: Strng;

Procedure G02EEF(var ISTEP: Integer;
var MEAN: Char;
MEANL: Integer;
var WEIGHT: Char;
```



```
WL: Integer;
var N: Integer;
var M: Integer;
var X: X_ArrayType;
var LDX: Integer;
var NAME: Strng_ArrayType;
NAME_L: Integer;
var ISX: ISX_ArrayType;
var MAXIP: Integer;
var Y: WTY_ArrayType;
var WT: WTY_ArrayType;
var FIN: Double;
var ADDVAR: Boolean;
var NEWVAR: Strng;
NVAR_L: Integer;
var CHRSS: Double;
var F: Double;
var MODEL: Strng_ArrayType;
MODL_L: Integer;
var NTERM: Integer;
var RSS: Double;
var IDF: Integer;
var IFR: Integer;
var FREE: Strng_ArrayType;
FREE_L: Integer;
var EXSS: EP_ArrayType;
var Q: Q_ArrayType;
var LDQ: Integer;
var P: EP_ArrayType;
var WK: WK_ArrayType;
var IFAIL: Integer);
stdcall;
external 'nagG02.dll';

Procedure R;
var
Temp: Char;
begin
Read(Temp);
end;

Procedure ReadData;
var
I: Integer;
J: Integer;

begin
ReadLn; {Skip heading in datafile}
Read(N, M);
R; {Skip blank space--See subroutine above}
Read(MEAN,WEIGHT);
If (M<MMAX) and (N<=NMAX) then
begin
for I:= 1 to N Do
begin
for J:= 1 to M Do
begin
Read(X[J,I]);
end;
Read(Y[I]);
If (WEIGHT='W') or (WEIGHT='w') then
Read(WT[I]);
end;
end;
R;
for J:= 1 to M Do
begin
Read(ISX[J]);
end;
R;
for I:= 1 to M Do
begin
```

```
for J:= 0 to 2 Do {note the zero basing of the array and loop}
begin
Read(NAME[I,J]);
end;
R;
end;
Read(FIN);
end;

Procedure FreeVars;
begin
Write('Free variables: ');
for J:= 1 to IFR Do
begin
Write(FREE[J]);
Write(' ');
end;
WriteLn;
WriteLn('Change in residual sum of squares for free variables:');
for J:= 1 to IFR Do
begin
Write(EXSS[J]);
Write(' ');
end;
WriteLn;
WriteLn;
end;

begin
WriteLn('G02EEF Example Program Results');
ISTEP:= 0;
IFAIL:= 0;
ReadData;
for I:=1 to M Do
begin
IFAIL:=0;

G02EEF(ISTEP,MEAN,1,WEIGHT,1,N,M,X,NMAX,NAME,3,ISX,MMAX,Y,WT,
FIN,ADDVAR,NEWVAR,3,CHRSS,F,MODEL,3,NTERM,RSS,IDF,
IFR,FREE,3,EXSS,Q,NMAX,P,WK,IFAIL);
{NB Fortran requires the length of the strings to be passed immediately following the
strings themselves.
Therefore it expects an integer after every string parameter.}
if (IFAIL<>0) then
begin
WriteLn('IFAIL = ',IFAIL);
Exit;
end;
WriteLn;
WriteLn('Step ',ISTEP);
if (ADDVAR<>TRUE) then
begin
WriteLn('No further variables added maximum F =',F);
FreeVars;
Exit;
end
else
begin
WriteLn('Added variable is ',NEWVAR);
WriteLn('Change in residual sum of squares =',CHRSS);
WriteLn('F Statistic = ',F);
WriteLn;
Write('Variables in model: ');
for J:= 1 to NTERM Do
begin
Write(MODEL[J]);
Write(' ');
end;
WriteLn;
WriteLn;
WriteLn('Residual sum of squares = ',RSS);
WriteLn('Degrees of freedom = ',IDF);
```

```
WriteLn;  
if (IFR=0) then  
begin  
WriteLn('No free variables remaining');  
Exit;  
end;  
FreeVars;  
end;  
end;  
end.
```

## E.3 Calling C procedures from Visual Basic

### E.3.1 Multi-Dimensional Array

In Visual Basic, multi-dimensional arrays are stored by columns (as in Fortran) rather than by rows, which is the C convention. This means that care must be taken when a C procedure has two-dimensional array (matrix) arguments.

For example, assume that a 3 by 2 matrix:

```
11 12  
21 22  
31 32
```

is stored in a Visual Basic 2-dimensional array `a` in the natural manner, as in the following code fragment:

```
Dim a(2, 1) As Double  
a(0, 0) = 11  
a(1, 0) = 21  
a(2, 0) = 31  
a(0, 1) = 12  
a(1, 1) = 22  
a(2, 1) = 32
```

The array `a` consists of 6 elements stored in column order, as follows:

```
11 21 31 12 22 32.
```

However, the C convention dictates that two-dimensional arrays are stored in row order. Suppose the array `a` (above) were passed to a NAG C procedure, say the NAG C library routine `f02wec` which computes the Singular Value Decomposition (SVD) of the matrix:

```
Call f02wec(3, 2, a(0, 0), . . . . )
```

where the first two arguments specify the number of rows and columns in the matrix. The NAG C Library procedure would treat the array as representing a 3 by 2 matrix stored in row order:

```
11 21  
31 12  
22 32
```

which is not the intended matrix  $a$ .

One solution to this problem is to store the matrix in a one-dimensional array  $a1$ , with the element  $a1(i, j)$  stored in  $a1(i-1) * (tda + j-1)$ , where  $tda$  is the trailing dimension of the matrix (in this case 2).

```
Dim a1(5) As Double
Dim tda As Long
tda = 2
a1(0) = 11
a1(1) = 12
a1(2) = 21
a1(3) = 22
a1(4) = 31
a1(5) = 32
Call f02wec(3, 2, a1(0), tda.... )
```

Another solution is to store the transpose of the matrix  $a$  in a two-dimensional array  $at$ , with  $tda$  now being the leading dimension of the array  $at$ :

```
Dim at(1, 2) As Double
Dim tda As Long
tda = 2
at(0, 0) = 11
at(0, 1) = 21
at(0, 2) = 31
at(1, 0) = 12
at(1, 1) = 22
at(1, 2) = 32
Call f02wec(3, 2, at(0, 0), tda,.... )
```

The Visual Basic array  $at$  can be larger than is needed to store the 2 by 3 matrix  $A^T$ ; in order that the C routine accesses the correct array elements it is essential that  $tda$  is set to the correct value, i.e. the actual size of the trailing dimension of the array rather than the size of the matrix.

## E.4 Visual Basic calling Fortran components

Visual Basic users may use DLLs to boost the capabilities of their application. The secret lies in inserting the appropriate “Declare” statements in a Visual Basic module.

The following example applies to Microsoft Excel, but most of it applies equally to Visual Basic generally.

Open the Excel workbook and create a module by clicking on the Tools, Macro, Visual Basic menus of Excel 97. Insert the “Declare” statement into the module. For illustration we will use NAG Library routine S14AAF (a double precision function that takes two scalar arguments, the first of which is double precision and the second is integer).

If the source code for the DLL is written in Fortran it is worth making sure that that the command “Option Base 1” is at the top of the Module. This ensures that any VBA arrays declared start their indices at 1, making them compatible with the Fortran routines. It is also good practice to ensure that “Option Explicit” is also present. Unlike C, VBA arrays store by

column and are thus compatible with Fortran. Experienced VBA programmers may now use the Fortran routine as though it had been written in VBA, subject to the conventions contained in the “Declare” statement.

The following code appears in a module of the workbook:

```
Option Base 1
Option Explicit
Declare Function S14AAF Lib "NAGSX.DLL" (x As Double, ifail As Long) As Double
```

The simplest functions may be used directly. To see this, move to an ordinary Worksheet in this Workbook and select a cell before clicking upon the Function Wizard (fx) on the Excel Ribbon. In the “User Defined” section, you will find the NAG S14AAF routine. Proceed as prompted by the Wizard, putting the dummy value 0 for IFAIL when finally prompted for this. If you have typed in valid input for the parameters, the function is now evaluated and placed in the cell selected. (You might wish to type in the value 1.25 for X, 0 for IFAIL and verify that the cell value is now 0.9064.)

From the above it can be seen that interfacing with subprograms that only utilise basic scalar types is relatively trivial. In many cases though data must be obtained from a worksheet into a VBA array. This example also shows how Fortran CHARACTER data is handled even though, in general its use is discouraged. The example shows a real matrix multiply operation using 2 BLAS routines, DGEMM and DGEMV, which compute a matrix-matrix product and a matrix-vector product respectively.

The specification of DGEMM is as follows:

```
SUBROUTINE DGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)
CHARACTER*1 TRANSA, TRANSB
INTEGER M, N, K, LDA, LDB, LDC
REAL ALPHA, A(LDA,*), B(LDB,*), BETA, C(LDC,*)
```

The specification of DGEMV is as follows:

```
SUBROUTINE DGEMV(TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
CHARACTER*1 TRANS
INTEGER M, N, LDA, INCX, INCY
REAL ALPHA, A(LDA,*), X(*), BETA, Y(*)
```

The following code appears in a module of the workbook:

```
Option Explicit
'Make explicit declaration of variables compulsory.
'Necessary because a Variant will not suffice as an argument to a
FORTRAN routine
Option Base 1
'Useful because FORTRAN array start at 1, whereas VBA array index from 0
by default

Declare Sub DGEMV Lib "NAGF06.DLL" (ByVal trans As String, _
ByVal length_trans As Long, m As Long, n As Long, alpha As Double, _
a As Double, lda As Long, x As Double, incx As Long, beta As Double, _
y As Double, incy As Long)

Declare Sub DGEMM Lib "NAGF06.DLL" (ByVal transa As String, _
```

```
ByVal length_transa As Long, ByVal transb As String, _  
ByVal length_transb As Long, _  
m As Long, n As Long, k As Long, alpha As Double, a As Double, _  
lda As Long, b As Double, ldb As Long, beta As Double, c As Double, _  
ldc As Long)
```

```
Private Function dimension(myVariantArray) As Long  
'Determines the dimension of an array or vector  
Dim retval As Long, i As Long  
On Error Resume Next  
Do Until retval = -1  
    retval = -1  
    i = i + 1  
    retval = UBound(myVariantArray, i)  
Loop  
dimension = i-1  
End Function
```

```
Sub Assemble(x, a)  
' Takes an argument X and form a VB array A (ReDimmed)  
Dim s As String  
Dim bits As Long, i As Long, j As Long  
Dim m As Long, n As Long  
Dim myX As Range  
Dim dks As Integer  
  
s = TypeName(x) 'Since x might be a NAME, or a selection or range
```

```
Select Case s  
Case "String"  
Set myX = Range(x)
```

```
Case "Variant()"  
dks = dimension(x)
```

```
If dks = 1 Then  
n = UBound(x)  
m = 1  
ReDim a(m, n)  
For j = 1 To n  
a(1, j) = x(j)  
Next j  
Exit Sub
```

```
Else  
If dks = 2 Then  
n = UBound(x, 2)  
m = UBound(x, 1)  
ReDim a(m, n)  
For j = 1 To n  
For i = 1 To m  
a(i, j) = x(i, j)  
Next i  
Next j  
Exit Sub  
End If  
End If
```

```
Case Else  
Set myX = x  
End Select
```

```
bits = myX.Areas.Count ' For this simple example bits should be 1
```

```
'Now get the overall dimension of the matrix  
m = myX.Rows.Count  
n = myX.Columns.Count  
ReDim a(m, n) ' VB Array redimensioned
```

```
' Assemble the matrix A  
For j = 1 To n  
For i = 1 To m
```

---

```

a(i, j) = myX.Cells(i, j).Value
Next i
Next j
Exit Sub

End Sub

Sub Assemblev(x, a)
` Takes an argument X and form a VB array vector (ReDimmed)
Dim s As String
Dim bits As Long, i As Long, j As Long, k As Long
Dim m As Long, n As Long, p As Long
Dim myX As Range

s = TypeName(x) ` Since x might be a NAME, selection or range

Select Case s
Case "String"
Set myX = Range(x)

Case "Variant()"
n = UBound(x)
ReDim a(n)
For j = 1 To n
a(j) = x(j)
Next j
Exit Sub

Case Else
Set myX = x
End Select

bits = myX.Areas.Count ` For this simple example bits should be 1
`Now get the overall dimension of the matrix
m = myX.Rows.Count
n = myX.Columns.Count
`Cater for possibility of row vector
If m > n Then
p = m
Else
p = n
End If

ReDim a(p) ` VB Array redimensioned
` Assemble the matrix A
k = 0
For j = 1 To n
For i = 1 To m
k = k + 1
a(k) = myX.Cells(i, j).Value
Next i
Next j
End Sub

Function NAGDgemm(a, b, Optional c, Optional alpha, Optional beta, _
Optional transa, Optional transb) As Variant
`The matrix multiply routine
Dim m As Long, k As Long, n As Long
Dim i As Long, j As Long
Dim myTransa As String * 1, myTransb As String * 1
Dim myAlpha As Double, myBeta As Double
Dim myA() As Double, myB() As Double, myC() As Double

` Entry-deal with optional parameters

If IsMissing(transa) Or VarType(transa) = vbError Or IsNull(transa) Then
myTransa = "N"
Else
myTransa = transa
End If

If IsMissing(transb) Or VarType(transb) = vbError Or IsNull(transb) Then

```

```
myTransb = "N"
Else
myTransb = transb
End If

If IsMissing(alpha) Or VarType(alpha) = vbError Or IsNull(alpha) Then
myAlpha = 1#
Else
If IsNumeric(alpha) Then
myAlpha = alpha
Else
myAlpha = alpha.Value
End If
End If

If IsMissing(beta) Or VarType(beta) = vbError Or IsNull(beta) Then
myBeta = 0#
Else
If IsNumeric(beta) Then
myBeta = beta
Else
myBeta = beta.Value
End If
End If

Call Assemble(a, myA) ` copy contents of a into VBA array myA
Call Assemble(b, myB) ` copy contents of b into VBA array myB

Select Case myTransa `whether to transpose A
Case "N", "n"

Select Case myTransb `whether to transpose B

Case "N", "n"
m = UBound(myA, 1)
k = UBound(myA, 2)
n = UBound(myB, 2)

If IsMissing(c) Or VarType(c) = vbError Or IsNull(c) Then
ReDim myC(m, n)
Else
Call Assemble(c, myC) ` copy contents of c into VBA array myC
End If

If ((UBound(myB, 1) <> k) Or (UBound(myC, 1) <> m) Or (UBound(myC, 2)
<> n)) Then

`Array dimensioning error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

Else

Call DGEMM(myTransa, 1, myTransb, 1, m, n, k, myAlpha, myA(1, 1), m,
-
myB(1, 1), k, myBeta, myC(1, 1), m)
NAGDgemm = myC
Exit Function

End If

Case "T", "t", "C", "c"

m = UBound(myA, 1)
k = UBound(myA, 2)
n = UBound(myB, 1)
If IsMissing(c) Or VarType(c) = vbError Or IsNull(c) Then
ReDim myC(m, n)
Else
Call Assemble(c, myC)
```



```
End If

If ((UBound(myB, 1) <> n) Or (UBound(myC, 1) <> m) Or (UBound(myC, 2)
<> n)) Then
'Array dimensioning error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

Else

Call DGEMM(myTransa, 1, myTransb, 1, m, n, k, myAlpha, myA(1, 1), m,
-
myB(1, 1), n, myBeta, myC(1, 1), m)
NAGDgemm = myC
Exit Function

End If

Case Else

'Input argument error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

End Select

Case "T", "t", "C", "c"

Select Case myTransb

Case "N", "n"
m = UBound(myA, 2)
k = UBound(myA, 1)
n = UBound(myB, 2)

If IsMissing(c) Or VarType(c) = vbError Or IsNull(c) Then
ReDim myC(m, n)
Else
Call Assemble(c, myC)
End If

If ((UBound(myB, 1) <> k) Or (UBound(myC, 1) <> m) Or (UBound(myC, 2)
<> n)) Then

'Array dimensioning error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

Else

Call DGEMM(myTransa, 1, myTransb, 1, m, n, k, myAlpha, myA(1, 1), k,
-
myB(1, 1), k, myBeta, myC(1, 1), m)

NAGDgemm = myC
Exit Function

End If

Case "T", "t", "C", "c"

m = UBound(myA, 2)
k = UBound(myA, 1)
n = UBound(myB, 1)
If IsMissing(c) Or VarType(c) = vbError Or IsNull(c) Then
```

```
ReDim myC(m, n)
Else
Call Assemble(c, myC)
End If

If ((UBound(myB, 1) <> n) Or (UBound(myC, 1) <> m) Or (UBound(myC, 2)
<> n)) Then

`Array dimensioning error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

Else

Call DGEMM(myTransa, 1, myTransb, 1, m, n, k, myAlpha, myA(1, 1), k,
_
myB(1, 1), n, myBeta, myC(1, 1), m)
NAGDgemm = myC
Exit Function
End If

Case Else

`Input argument error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

End Select

Case Else

`Input argument error
ReDim myC(1, 1)
myC(1, 1) = CVErr(xlErrValue)
NAGDgemm = myC
Exit Function

End Select

End Function

Function NAGDgemv(a, x, Optional y, Optional alpha, Optional beta, _
Optional trans) As Variant
` Matrix times vector routine
Dim m As Long, n As Long
Dim i As Long, j As Long
Dim myTrans As String * 1
Dim myAlpha As Double, myBeta As Double
Dim myA() As Double, myX() As Double, myY() As Double

If IsMissing(trans) Or VarType(trans) = vbError Or IsNull(trans) Then
myTrans = "N"
Else
myTrans = trans
End If

If IsMissing(alpha) Or VarType(alpha) = vbError Or IsNull(alpha) Then
myAlpha = 1#
Else
If IsNumeric(alpha) Then
myAlpha = alpha
Else
myAlpha = alpha.Value
End If
End If

If IsMissing(beta) Or VarType(beta) = vbError Or IsNull(beta) Then
myBeta = 0#
```

```
Else

If IsNumeric(beta) Then
myBeta = beta
Else
myBeta = beta.Value
End If
End If

Call Assemble(a, myA)
Call Assemblev(x, myX)

Select Case myTrans

Case "N", "n"
m = UBound(myA, 1)
n = UBound(myA, 2)

If IsMissing(y) Or VarType(y) = vbError Or IsNull(y) Then
ReDim myY(m)
Else
Call Assemblev(y, myY)
End If

If (UBound(myY, 1) <> m) Then

'Array dimensioning error
ReDim myY(1)
myY(1) = CVErr(xlErrValue)
NAGDgemv = myY
Exit Function

Else

Call DGEMV(myTrans, 1, m, n, myAlpha, myA(1, 1), m, _
myX(1), 1, myBeta, myY(1), 1)
NAGDgemv = myY
Exit Function

End If

Case "T", "t", "C", "c"

m = UBound(myA, 1)
n = UBound(myA, 2)
If IsMissing(y) Or VarType(y) = vbError Or IsNull(y) Then
ReDim myY(n)
Else
Call Assemblev(y, myY)
End If

If (UBound(myY, 1) <> n) Then

'Array dimensioning error
ReDim myY(1)
myY(1) = CVErr(xlErrValue)
NAGDgemv = myY
Exit Function

Else

Call DGEMV(myTrans, 1, m, n, myAlpha, myA(1, 1), m, _
myX(1), 1, myBeta, myY(1), 1)

NAGDgemv = myY
Exit Function

End If

Case Else

'Input argument error
```

```
myY(1) = CVerErr(xlErrValue)
NAGDgemv = myY
Exit Function
End Select
End Function
```

## E.5 Calling Fortran Subroutines from LabVIEW

The following example is based on an earlier release of LabVIEW.

- Select the icon for the “CALL LIBRARY FUNCTION” VI from the appropriate menu.
- Fill in the VI
  - Library Name
  - Function Name, which is the Fortran subroutine name
  - Calling Conventions, e.g. “C”
- Match the parameters appropriately as follows:
  - return type is “void” for subroutines
  - each arguments has
    - type: numeric/array
    - Datatype: Signed 32-bit Integer/8-byte double Pointer to value

## E.6 Incorporation of a Fortran Subroutine into MATLAB

The following example shows a Fortran gateway to call the subroutine CUBE. Cube takes two arguments; X is the input and Y the output:

```
SUBROUTINE CUBE(X,Y)
C.. Scalar Arguments..
DOUBLE PRECISION X,Y
C.. Executable Statements..
Y = X*X*X
END
```

The gateway first checks that the right number of input and output arguments has been passed and that these are compatible with the routine. MATLAB storage is then allocated for the output argument. The input argument is copied from MATLAB storage to the Fortran scalar X. CUBE is then called with its output being stored in Fortran scalar Y which is copied back to MATLAB storage.

```
C Example of a Fortran gateway to call the above subroutine CUBE
SUBROUTINE MEXFUNCTION(NLHS,PLHS,NRHS,PRHS)
C.. Scalar Arguments..
INTEGER NLHS,NRHS
C.. Array Arguments..
```

---

```

C This assumes 32 bit pointers
INTEGER PLHS(*),PRHS(*)
C.. Local Scalars..
DOUBLE PRECISION X,Y
INTEGER M,N,SIZE,X_PR,Y_PR
C.. External Functions..
INTEGER MXCREATEFULL,MXGETM,MXGETN,MXGETPR,MXISCOMPLEX,
+ MXISNUMERIC
EXTERNAL MXCREATEFULL,MXGETM,MXGETN,MXGETPR,MXISCOMPLEX,
+ MXISNUMERIC
C.. External Subroutines..
EXTERNAL CUBE,MEXERRMSGTXT,MXCOPYPTRTOREAL8,MXCOPYREAL8TOPTR
C.. Executable Statements..
C Check correct number, structure and type of input and output
C arguments
C Note: MEXERRMSGTXT does not return
IF (NRHS.NE.1) THEN
CALL MEXERRMSGTXT('One input argument required.')
ELSE IF (NLHS.NE.1) THEN
CALL MEXERRMSGTXT('One output argument required.')
END IF
M = MXGETM(PRHS(1))
N = MXGETN(PRHS(1))
SIZE = M*N
IF (SIZE.NE.1)
+ CALL MEXERRMSGTXT('Input argument must be scalar.')
IF (MXISNUMERIC(PRHS(1)).EQ.0)
+ CALL MEXERRMSGTXT('Input argument must be a number.')
IF (MXISCOMPLEX(PRHS(1)).EQ.1)
+ CALL MEXERRMSGTXT('Input argument must not be COMPLEX.')

C Allocate space for the output argument.
PLHS(1) = MXCREATEFULL(M,N,0)
C Get pointers to the LHS and RHS arguments real parts
X_PR = MXGETPR(PRHS(1))
Y_PR = MXGETPR(PLHS(1))

C Copy input argument from MATLAB to Fortran storage
CALL MXCOPYPTRTOREAL8(X_PR,X,SIZE)

C Finally call our Fortran routine.
CALL CUBE(X,Y)

C And copy the output argument from Fortran storage back to MATLAB
CALL MXCOPYREAL8TOPTR(Y,Y_PR,SIZE)
END

```

This gateway can then be compiled and linked using the “fmex” script, the directory containing the gateway added to the MATLAB path and the function cube can be called directly from MATLAB, e.g.:

```
>> a=cube(7)
```

```
a =
```

```
343
```