RAL-93-055

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-93-055

A Programming Guide for the Development of Engineering Applications Software in Fortran

C J Fitzsimons and C Greenough

July 1993

A Programming Guide for the Development of Engineering Applications Software in Fortran

C.J. Fitzsimons and C. Greenough

June 1993

Abstract

The need to develop good quality software is well recognised by both the academic and industrial communities. In many areas of industry, exacting and very detailed standards of software quality control are imposed in development projects. However in the academic research environment, although the development of quality software is a goal, the restriction and burden placed on the developer by these formal standards is considered unacceptable.

This guide attempts to suggest an approach to software development that involves the use of a practical subset of quality assurance (QA) methods for the academic community. The goal is to ensure that the software is easier to develop, maintain, re-use and distribute to colleagues without an excessive overhead in time and effort.

The guide covers a simple approach to software design, implementation and documentation. It also suggests a number of inexpensive software tools that can help the developer produce good software.

An example of the application of the proposed methods is given to demonstrate the approach.

Mathematical Software Group
Computational Modelling Division
Rutherford Appleton Laboratory
Chilton, Didcot
Oxfordshire OX11 0QX

Contents

1 INTRODUCTION

This guide sets out an approach for the development of engineering analysis software written in Fortran in an academic research environment. This involves the use of a practical subset of quality assurance (QA) methods to help ensure that the software is easier to develop, maintain, re-use and distribute to colleagues. Engineering applications software falls into one of three categories [5]:

1. **one-off**, i.e. written and used by the originator for the sole purpose of generating a set of results, or demonstrating a particular algorithm,

2. **re-usable**, i.e. written in portable form with some documentation, with a view to use not only by the originator, but also by others who wish to produce results for related applications or employ the algorithms developed, and

3. **commercial**, i.e. made to be portable, robust for a wide range of applications, with added features to make it easier to use, and fully documented, to the extent that it can be marketed. Many such codes have gone through a gestation period as re-usable software before being brought to a commercial standard.

The working practices necessary to produce fully quality assured software are not usually considered conducive to conducting research. Therefore a balance between current practice and best practice must be found in order that re-usable software can be produced in a research environment, without interfering too much with the primary aim of the research.

QA methods are used to help ensure the development of software which has a number of positive attributes [1], e.g. portability, readability, usability, testability, integrity, flexibility, reliability, extensibility and maintainability. Attributes such as integrity (how does the program perform against its specification), flexibility (does the program have the full potential of the range of application of the method) and reliability (how does the program perform with minor changes in the input data) will normally be addressed during algorithm evaluation. Of the remaining six attributes, by paying more attention to portability, readability (can the workings of the program be broadly understood) and usability (is data preparation relatively straightforward and well-documented) the developer can greatly improve the overall quality of the software. This in turn greatly simplifies the achievement of testability, extensibility and maintainability.

An important consideration in the choice of approach is the cost of some of the software tools available for assuring the quality of developed software. Often, the leading tools are far too expensive for consideration by a research group, where resources are limited and directed towards the research. The tools chosen are inexpensive, or freely available. In time, when the benefits of using QA methods are seen, some researchers might decide to investigate the use of more expensive and comprehensive software tools.

In the next Section we describe a practical approach to the design and implementation of engineering applications software. This is followed by a brief description of the software tools mentioned, contact addresses for which are in Appendix C. In Section 4 we comment on some of the Fortran 77 compilers and debuggers available. In Appendix A we give some guidelines on writing Fortran programs based on those given in [10]. In Appendix B we provide a detailed worked example of the different stages in designing and writing a piece of software. This particular program, `flux`, may be obtained from the Higher Education National Software Archive (HENSA) at the University of Kent.

2 AN APPROACH TO DESIGN AND IMPLEMENTATION

Design and implementation methodologies are commonly viewed as exercises that get in the way of writing software. A rigorous application of any design and implementation methodology, it is believed, adds an unacceptable overhead to the development of re-usable applications software. However, certain elements of these methodologies can be used beneficially in a research environment and they greatly help the production of re-usable software. The methods and tools discussed in this guide can actually save development time in many instances by either helping reduce the number of bugs in the software as it is being written or else through providing tools which help to identify common sources of error quickly and easily.

There are many software design and implementation methodologies, e.g. Jackson Structured Design [2], HIPO (Hierarchical Input Processing Output) [4], the IEE Model Procedures [6]. These advocate a formal procedure for software development, use and maintenance. Some of their features, drawn mainly from [6] and adapted for a research environment, will be discussed here; for a more detailed explanation of each we recommend the stated references.

Verification is a vital component of good software practice and occurs at several stages of the software development process. We will highlight its presence under the appropriate subsections which follow.

2.1 Problem Definition

Software design is supposed to begin with a clearly-written statement of what problem the software is supposed to be addressing. In research, the user is usually also the developer, or else next door to the developer. Therefore, the target problem is not always clearly stated, because it is "obvious". This clarity of perception invariably wears off in time. A minimum contribution to software design is a statement of what problem the software is aiming to solve: a problem specification document. This includes the equations and their dimensionality, the types of boundary conditions, the geometric configurations, the physical models, and a statement on the type of hardware on which it is intended to run. Any likely extensions to the problem envisaged in the future, e.g. the level of increased geometrical or physical complexity, should also be clearly stated. This exercise is not very time-consuming but leaves behind a clear picture of where the software is aiming today, and where future versions are likely to go. It often has a large influence on certain choices made in the first version of the software, to make sure that it will be upwardly-compatible. It also serves to clear up any woolliness about the precise problem being solved.

2.2 Algorithm Selection

The work required at this stage of the design process depends on the developer's aims: to produce software using existing algorithms or to facilitate the design and development of new algorithms.

2.2.1 Using Existing Algorithms

A second simple document which should be produced at this stage is a technical report that details the algorithms which will be employed in the software (e.g. choice of element or volume shape, choice of problem discretisation, method for solving any linear or nonlinear systems which arise in the solution

of the problem) and which data formats and interfaces will be used. This document should also contain a list of any external packages (e.g. FEMGEN, FEMVIEW, IDEAS, or PATRAN) or software libraries (e.g. NAG, BLAS, or LAPACK) which will be used, or any routines from previously-developed in-house software.

2.2.2 Algorithmic Research

Software used to investigate new algorithms is, of necessity, unpolished. However, with some care and planning much of what is written can be re-used in subsequent work. Differentiating between what is known or standard in the program and what is new can clarify many issues and also lead to the ready identification of which parts of the program can make use of existing subroutines or functions.

Many of the comments from the previous subsection still hold true. Much of what is going to be used is standard and can be easily written down. Often what is different is the manner in which it going to be used. For example, software written to investigate a new discretisation scheme will often differ only in the portion of the program where the discrete equations are computed and assembled; the remainder of the program is standard.

2.3 Software Design

Once the problem definition and numerical algorithm selection is completed, the software design phase can commence. There are many possible methods to choose from, and the choice will depend on personal preference. When the software is being designed, it is useful to keep in mind existing software which can be used, e.g. routines already written by the author for a previous piece of software, or well-tested and widely available libraries such as NAG or BLAS. Their re-use reduces the amount of validation required in order to produce a working piece of software and greatly simplifies any bug-fixing needed.

There are many software design methodologies available today. Unfortunately, they have not been written with the needs of the scientific engineering applications software developer in mind. Many researchers find them inappropriate to their needs. However, software which is written before it is designed usually becomes unwieldy and a nightmare to either update or maintain.

We suggest the use of a top-down approach leading to a structured program. The program is decomposed into functions, e.g. data initialisation, problem input, problem solution, and problem output. Each of these is, in turn, decomposed into its constituents. After a few stages of this process (the precise number will depend on the size and complexity of the algorithms being implemented), the designer will have a description of what each subroutine is for, the data it requires and how it operates on the data. Moreover, this makes it easier to see what existing — presumably tested! — routines can be used in developing this program.

At this stage we perform the first stage of the verification process and appraise how well the software design meets the initially-stated user's requirements.

2.4 Software Implementation

This is where some people are tempted to start! The software should be implemented module by module, and each individually tested. The use of strict compilation options, and analysis tools such as

f2c and Toolpack reduces the number of holdups at this stage of development, by checking declarations and subroutine calling sequences, the two most common sources of error in Fortran software.

Programming style is personal. However, all good programming styles incorporate the use of common features such as a structured style, explicit typing of variables, adherence to the ANSI standard, and extensive use of (illuminating) comments. In Appendix A we give a set of programming guidelines based on those in [10]. Kernighan and Plauger [7] also provides a very readable and useful introduction to the subject.

Often insufficient attention is paid to exceptional data, i.e. data outside the ranges for which the subroutine was designed. It is important that mechanisms to trap these exceptions are incorporated in the implementation.

An important part of software verification is its testing. This comprises two stages: at a subroutine level and integration testing. Typical tests of a subroutine include: any valid input produces the expected output and invalid input is handled without the program crashing. The results of these tests should form part of the system documentation, so that they can be re-used in the future. Integration testing involves checking that each subroutine integrates into the full program.

The final stage in the verification process is software validation or acceptance testing. Validation is ensuring that the software is an accurate implementation of the mathematical model specified to meet the user's requirements. This involves the solution of agreed tests which satisfy the required specification. They could be, for example, well-known and documented test cases for the given application. The results of these tests should form part of the system documentation.

2.5 User Guide

While the software is being developed, the user guide should be written. This will contain the previously written documents, as well as information on how to use the software (including all input command syntax) and prepare input data, a description of output and any error reporting instructions.

2.6 Software Release

The finished software should come complete with a set of test case input files and their corresponding output files, which should be user-reproducible. These serve to help ensure that the software has been mounted correctly on a new system, or after a change of operating system version. Any other documented test cases should also form part of the software release.

3 VALIDATION, EVALUATION, VERIFICATION AND TESTING TOOLS

The cost of many of the best tools in current use is beyond the budget of most academic research groups. However, effective use can be made of public domain or inexpensive verification and testing tools which provide a similar functionality to the more expensive tools, but at the cost of being not quite as easy to use. We use f2c and Toolpack in our day-to-day work, in conjunction with the Salford compiler (running on an 386-based PC). These provide a wide range of static and dynamic checks on the software and find the commonly-made mistakes when software is being written.

The words verification, validation and evaluation mean different things to different people. Before continuing we will say what we mean by each of these:

- **Validation** is testing whether the software is an accurate implementation of the mathematical model,
- **Evaluation** is ascertaining how the implemented model reflects the problem physics, i.e. how the results compare to those anticipated for a comparable real life system,
- **Verification** is the overall checking procedure which includes validation and evaluation. It contains five main identifiable stages:
    1. **Design Reviews** Design reviews are carried out at the end of each design stage. These appraise how well the system and the upper levels of system decomposition meet the initially-stated user's requirements.
    2. **Code Walk-Through** Each module is walked-through when the programmer is satisfied that it is ready. This checks that, formally, the module performs to specification.
    3. **Module Testing** Testing is demonstrating that the module meets the requirement specification under all circumstances. Tests include: any valid input produces the expected output, and invalid input is handled without the program crashing. The results of these tests form part of the system documentation, so that they can be reused in the future.
    4. **Integration Testing** Integration tests ensure that each module will integrate satisfactorily with the full system.
    5. **Acceptance Testing** Software acceptance tests are agreed tests which will demonstrate that the software satisfies the required specification. Examples of these include well-documented benchmark test cases for given CFD applications. The results from these tests form part of the system documentation.

3.1 Some Available Tools

Several hundred software tools are available for use in different parts of the software lifecycle. Clearly, it is beyond the scope of this guide to survey them all. The use of appropriate tools enhances the software quality, and reduces the amount of maintenance required. Compilers and debuggers perform some verification and testing of the software; they are dealt with separately in the next subsection.

Verification tools perform a static analysis of the software and typically offer:

- **control flow analysis** which highlights unreachable software, multiple entry loops, and breaches in a predefined programming standard,
- **data flow analysis** which looks for errors in variables, e.g. identification of variables which are initialised but never used,
- **information flow analysis** which checks the relationship between the data input to a module, the computation, and the output from the module, and
- **complexity analysis**, e.g. McCabe's Cyclomatic Complexity [8] which measures the number of decision points in a module.

Testing tools offer a range of facilities including some of the following: capture (each keystroke is recorded), playback (the recorded keystrokes are replayed, the speed of which can be altered), comparison of files, input generation (automatic generation of large volumes of random input — no such tools are commercially available), simulators and emulators (testing for hardware configurations not available locally), a testing harness, coverage analysis (what percentage of statements, branches, etc are covered by a given test) and debugging facilities (see the next subsection).

Goh and Sinclair [3] survey a wide range of verification and testing tools. Here we list and describe briefly a few of the more commonly-used tools and draw on their descriptions and the comments of others who have used these tools. We begin by describing two tools, Forchek and f2c, which are in the public domain and thus suitable for use by all developers. These should form the basis of software development tools used. The remaining tools described here are commercial products whose use will enhance the quality of the developed software, but whose cost limit their availability. The market for software development tools is changing rapidly and it is certain that you will find tools other than those described here which will aid your work.

The contact for each of the companies referred to here may be found in Appendix A. Prices and products change over time, so we suggest that you contact the supplier for up to date information about any product in which you are interested.

3.1.1 Higher Education National Software Archive

Some of the tools described below are freely available from the Higher Education National Software Archive (HENSA) at the University of Kent. For up to date instructions on how to use HENSA send the two word e-mail message

send index

to the JANET e-mail address netlib@unix.hensa.ac.uk

In reply you will receive full details of how to use netlib and how to acquire f2c and Forchek.

3.1.2 f2c

f2c, available from HENSA, is – as its name suggests – a Fortran to C conversion utility. While it produces a C translation which sometimes appears ungainly, it is a useful utility for checking whether variables and arrays have been declared properly and whether subroutine calling sequences are compatible with the subroutine definition.

3.1.3 Forchek

Forchek is available from the HENSA, where it is in the fortran chapter of netlib.

It is not primarily intended to detect syntax errors. Forchek is useful in finding semantic errors, i.e. those which pass the compiler but may cause problems during execution: variables which are never used, uninitialised variables and undeclared variables.

3.1.4 Toolpack/NagWare

Toolpack, from NAG Limited, is public domain software for which you just pay a distribution fee. Toolpack is a collection of more than 50 Fortran tools which perform a range of tasks from precision conversion to the pretty printing of source software. For example, source code is checked for compliance with a variety of standards including ANSI and its DoD extensions, source code is restructured, its portability is verified and a procedure calling graph is produced.

It contains a verifier tool which performs most of the static analysis checks of QA Fortran and Forchek. However, the output is just a list of error messages which the user has to match back to the source, unlike the other two tools where the error messages are embedded in the source. Another tool produces call trees.

NagWare Fortran 77 tools are a reworking of the basic Toolpack tools into a more consistent and easier to use form, e.g. the error and warning reports are more helpful! As a consequence, this costs money. For example, an academic licence for a SUN3 workstation costs £720, with an annual maintenance fee of £130. The cost for a Sparcstation is twice that.

3.1.5 QA Fortran

QA Fortran, from Programming Research Limited, is a verification and test coverage tool, with many useful features. The tool checks for adherence to the ANSI standard, and the user may also reconfigure the system so that it tests for adherence to any in-house programming standard. It also verifies compliance with over 4000 subroutines from standard packages, e.g. the NAG library, and IMSL. Features which will be obsolescent under Fortran 90 are flagged, as well as Fortran 77 features known to have poor efficiency.

The tool handles maintenance problems associated with large software projects: performing a fan-in/fan-out analysis, working directly on the source software, producing enhanced calling and called-by trees, performing a structured path analysis, and checking argument passing to subprograms for type and number taking account of direction of passing. It also applies a comprehensive range of complexity metrics which are very useful in identifying software likely to be a maintenance problem. QA Fortran is available on most Unix and VMS platforms, and has an X-Windows interface.

The company is willing to mount demonstrations and to allow a potential customer to apply it to his own software. One annoying feature we found was that although it highlighted problems in the software, it was not possible to fix these from within the QA Fortran environment and then resubmit the software. The second annoying feature is that it costs £6000 per user.

This software is used by — for example — the European Space Agency, Shell Chemicals, British Petroleum, UKAEA, and the University of Kent.

3.1.6 Softest

Softest, from Information Processing Limited, is a test harness and coverage tool which is used during unit and system integration testing. It offers test data generation, expected results generation, test driving and results analysis, run time assertion verification (to check the correctness of the program as it executes) and performance analysis. It has been assessed as probably having a steep learning curve, and costs from £7500 (with an academic discount of 90%).

4 COMPILERS AND DEBUGGERS

Nearly all Fortran compilers have an ANSI option, which allows the programmer to check that the software complies with the ANSI standard. Some compilers have additional features which simplify the sometimes laborious task of tracking down simple errors. For example, some compilers check whether a variable has been assigned a value by the program before its first use, or whether any array accesses occur outside the declared range of the array. The most comprehensive Fortran compiler available, of which we are aware, is the Salford compiler (from Salford Software), which is available on 386-based and 486-based PCs. An academic licence for the 386 version costs £604, and the 486 version costs £716.

The source level debugger is an essential software development tool. No matter how much care is taken in the design and implementation of software, there are usually some bugs whose detection and correction requires the use of a source level debugger, i.e. a software tool which allows the programmer to track the flow of data through the program by inspecting — and in some cases, changing — the values of selected variables and arrays in the program. The use of such tools is a great advance over old methods that involved writing out selected values to data files at selected points in the program. While such tools greatly improve the lot of the programmer, there is a capability disparity between those of different compiler writers.

The debugger available under Primos on Prime computers or the VAX debugger contain all the features required by the software developer. The SUN debugger has a narrower functionality; its major defect is that the software developer cannot easily inspect the values in a user-specified subset of an array in the program, one of the main uses of a debugger in practice. There is also a debugger called XRAY, from Microtec Research Limited, which runs on a range of workstations and PCs for Fortran, C Pascal, and assembler. An added feature is that it is possible to recreate sessions and to implement automated test sequences. There are approximately 200 users in the UK and its price starts at £2000.

5 SUMMARY

We have presented a development guide for software which is easier to maintain, re-use and distribute. The intention behind this guide is to help researchers to developsoftware which is re-usabled, either by others or in their own future software developments.

The method of software development has been broken down into six stages. First the problem which will be solved is defined. Then the algorithms and numerical methods are chosen, e.g. solution strategy, discretisation procedure, linear and nonlinear solution techniques. Thirdly, the software is designed using the developer's preferred method. The design is then implemented using standard-conforming Fortran. While the software is under development, its user guide should be written. Finally, the software is released, i.e. it is completed, applied to a number of documented test cases, and ready for use.

6 ACKNOWLEDGEMENTS

The authors would like to acknowledge the members of the SERC CFD Community Club Steering Group, Prof. Geoff Hammond (Bath), Richard Bond (SERC, Swindon), Prof. Derek Causon (Manch-

ester P), Dr Bruce Dean (WS Atkins), Dr Dave Emerson (SERC, Daresbury), Prof. David Gosman (Imperial), Prof. Phil Hutchinson (Cranfield), Dr Brian Williams (DRA, Aerospace), whose helpful comments and suggestions improved the content and scope of this document.

Steve Fiddes (Bristol) originally provided flux for use in a CFD Community Club Summer School in 1991 and very kindly allowed it to appear in its current form in this document.

References

[1] **C. Albone** "Report of the AGCFM Working Party on Software Quality Assurance" Royal Aircraft Establishment, Tech. Memo Aero 2105, June (1987)

[2] **J.R. Cameron** "An Overview of JSD" *IEEE Softw. Engng.* **SE–12** No. 2, 222–240 (1986)

[3] **T. Goh, C. Sinclair** *Report of Research into Quality Assurance Software and Tools* EASE Technical Report **ETR 9/91** Rutherford Appleton Laboratory (1991)

[4] **International Business Machines Corporation** *HIPO — A Design Aid and Documentation Technique* IBM Manual GC20–1851–1 (1975)

[5] **P. Hutchinson** *SERC CFD Advisory Group Final Report: Recommendations for Research Related to Computational Fluid Dynamics* May (1988)

[6] **The Institution of Electrical Engineers** *Software Quality Assurance Model Procedures* The Institution of Electrical Engineers (1990)

[7] **B.W. Kernighan, P.J. Plauger** *The Elements of Programming Style* (2nd Edition) McGraw-Hill (1978)

[8] **T.J. McCabe** "A Complexity Measure" *IEEE Trans. Softw. Engng.* **SE–2** No. 4, 308–320 (1976)

[9] **NATO** "NATO Software Quality Control System Requirements" Allied Quality Assurance Publication **AQAP–13** August (1981)

[10] **SIGFE** "Finite Element Software — Development Standards"

A General Guidelines on the Writing of Fortran Programs

A.1 Introduction

While it is recognised that individual style can play a large part in writing software, some guidelines for new programs have been drawn up. It is the intention that these should be considered not as a rigid set of enforceable rules but as points to bear in mind when writing. In particular cases there may indeed be good reasons for going against the guideline, but the author should be aware of these reasons.

A.2 General Comments

1. The routine should be well designed, and the code should reflect this design.
2. Every variable used should have a specific meaning.
3. Avoid non-standard constructs.
4. Increased efficiency should be attempted after the routine has been proved correct, and when definite knowledge of the major inefficient areas has been obtained.
5. If it is necessary for a variable to lie within a certain range for the program to behave correctly, this range should either be checked or be stated as an assumption in a comment.
6. Clarity and simplicity are good aims.
7. Compilers should be left to do much of the 'dirty work'. For example, avoid writing X + X + X + X for 4.*X just because the addition is faster, unless the code is in an efficiency-sensitive area.
8. Don't put in the obvious. Use guidelines sensibly.

A.3 Variables

1. Give all variables good mnemonic names.
2. Declare every variable explicitly.
3. Decide on naming conventions and stick to them. For example:
   (a) use single letter variables for loop control and temporary storage, but not otherwise;
   (b) have conventions for local/global names; such as: names beginning with I-K or O-Z are local, and names beginning with L-N or A-H are global.
4. Use a variable for a specific purpose. If the purpose changes, use another variable.
5. Use EQUIVALENCE only when really necessary.
6. Do not DATA initialise variables whose values will be altered later - particularly if the variable is local.
7. Initialise all variables before accessing their values.
8. Use LOGICAL types for variables which are used essentially as binary switches.

A.4 Labels

1. Make them increasing throughout the routine, initially with a minimum step length of at least 10 to allow for later amendments.
2. Put labels in columns 2-5 (if possible) so that they are clearly visible between comments.
3. Have a separate label convention for FORMAT statements.

A.5 Structure

1. Consider using indentation to emphasise the design structure.
2. Avoid unnecessary GOTO's.
3. Use separate labels for each DO termination.
4. Only put labels on CONTINUE statements - particularly when terminating DO loops. This allows easy insertion of extra code (possibly for debug prints).
5. Avoid arithmetic IF.
6. Emphasise loops that are not immediately apparent (such as those implemented with IF and GOTO). This could be done by comment or indentation.
7. Use functions or subroutines for repetitive sections. Statement functions can add clarity.
8. Use library routines if applicable, rather than writing your own copy.
9. Choose data representations that lead to simple programs. For example, use multi-dimensioned arrays if the logic calls for it.
10. Never test real numbers for equality. Be aware of (and comment) the number of significant figures you expect, e.g. A-B against a constant: say, that the difference is required to be within a specified significance. This can change when moving a program to another machine.
11. Take special care when handling characters, particularly if the code is to be transportable.
12. Construct the program in logical units. Do not break the structure just to save one or two orders.

A.6 Subroutines and Functions

1. Do not write routines with arguments that are neither read nor written.
2. Avoid side effects (e.g. in functions).
3. Do not re-order similar argument lists in different routines unless necessary.
4. Avoid internal subroutines (driven by GOTOs).
5. Check ranges of arguments, or comment on those untested.
6. Don't alter input arguments.

7. In a FUNCTION, the function name should only occur on the left hand side of an assignment. Use a local variable instead if the value is required elsewhere within the routine.

8. Each argument should have only one logical function, not many.

A.7 Input/Output

1. If possible, choose input layouts which are easy to prepare and check.

2. Check input for validity. Routines should be able to cope with random input.

3. Try to keep all I/O for a particular device in one routine.

4. Use variables for unit numbers.

5. Human-readable output must be clear. Machine readable output must be easy to re-input.

A.8 Testing

1. Instrument the program carefully.

2. After testing, be able to state explicitly what was tested.

A.9 A Commenting Standard for Fortran Software

A.9.1 Introduction

The main aim of software commenting is to provide sufficient information with the software to aid maintenance. The standard will be imposed on programs written by the group, and on anyone wishing to provide programs for us to mount and maintain.

A.9.2 The Standard

1. `SUBROUTINE` (or equivalent) statements or programme titles should be underlined. Suggested characters are - or =.

2. There should be a header section of comments immediately following the underlined SUBROUTINE statement or programme title.

3. The header section should be divided into sections. Each section should start with a non-indented (i.e. col 3) heading, which may optionally be underlined. The body of the section should be indented sufficiently to make the heading stand out (it is suggested that such comments begin in column 7).

4. Sections on Purpose and History must appear in each routine. Sections on Locals and Package are optional. All other sections must appear if there is some information other than "None" to be included. Otherwise the whole section may be omitted. (see below for details of sections).

5. The following defines section headings, order of sections and section body layout:

PURPOSE Brief description of what routine does.

METHOD Description of the tools used (e.g. algorithms) and how they are used. It is not necessary to give a full description of the program, including variable names.

HISTORY

AUTHOR initials, date, MACHINE — original machine

CHANGES initials, date, short comment. Change of machine if applicable.

ARGUMENTS IN List of arguments whose value is read by (for subroutines) the routine, or by inferior routines, together with description and constraints/assumptions.

ARGUMENTS OUT List of arguments whose value is set by (for subroutines) the routine, or by inferior routines, together with description and constraints/assumptions.

LOCALS List of important local variables, together with description. Important labels can be included here. Temporary variables and DO loop control variables need not in general be included.

ROUTINES CALLED

SYSTEM system library routines,

PACKAGE non-ANSI routines from the same package as this routine,

Library Name routines from the named library (such as NAGLIB, GINO, GKS, etc).

INPUT/OUTPUT Fortran channel number device type status change (or variable name). (e.g. MT re-winding).

ERROR ACTION Exception conditions and action taken. This applies only to errors explicitly determined by this routine, and not to errors which may possibly occur in inferior routines. Calling a specific error routine will, however, be noted here.

SPECIAL FEATURES Important points not covered above, such as details of any machine dependency or any personal naming conventions.

6. The following section should occur in one subprogram, at most, of a package:

   PACKAGE Overall comments on machine dependency, conventions, etc for the whole package. References to other documentation connected with the package. The chosen routine will normally be the most prominent (in some senses, the first).

7. If the header section is longer than 45 lines, it should be terminated by a repeat of the SUBROUTINE statement as a comment, underlined with a different character, (suggestion is *). This termination is optional for shorter headings.

8. The writer must be consistent in any conventions used.

9. In-line comments should follow these guidelines:

   (a) Code sections should be distinguished by some easily identifiable COMMENT form. Some method of differentiating different lexical levels (or subsections) may be used. Some suggested methods are:

      i. Line of characters such as -. Line is of different length for subsections.

      ii. As above, but different characters used for differentiating subsections.

iii. Blank comments between sections.

iv. Combinations of the above.

(b) Comments should be delimited in some way, and should not become confused with the code. They should not interfere with the visibility of labels. (e.g. single or double line underlimited comments starting in column 7 must be avoided).

(c) Comments must not just give syntactic descriptions of code. They should be concise.

B Example of Use of the Procedure

In this appendix we present a complete example developed using the methods outlined in this document. We follow the stages outlined in Section 2. The software chosen has been developed mainly as a tool to introduce users to a variety of numerical schemes used in the solution of fluid dynamics problems. This example uses the program flux which was developed by Mr. S.P. Fiddes of the University of Bristol to illustrate some of the numerical techniques used in solving CFD problems.

B.1 Problem Definition

We require a computer program to run on a SUN3 workstation running under Unix which will permit the user to compare a variety of discretisation schemes for the one-dimensional linear advection equation with unit wave speed or the inviscid Burgers equation. The program will normally be used for teaching purposes. Therefore it should be written so that the student can introduce additional discretisation schemes or starting conditions.

The one-dimensional linear advection equation is

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{1}$$

where c is the wave speed. The solution of this equation is a wave of fixed shape travelling at speed c. Here we consider $c = 1$. The one-dimensional inviscid Burgers' equation is

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{2}$$

The program should include the following discretisation schemes: forward-time centred-space (FTCS), upwind differencing, MacCormack, Backward Euler with central differencing, Backward Euler with upwind differencing, Beam and Warming (2nd order accurate in time) and Lax-Wendroff.

The program should include the following starting profiles: a ramp function for both equations and a smooth bump for the linear advection equation.

The user should be able to specify the number of grid points in the solution domain of $[0, 3]$ and also the Courant-Friedrichs-Lewy (CFL) number for the problem. The program should solve the problem using a time interval of $[0, 3]$ also.

B.2 Algorithm Selection

The user has specified the majority of the algorithms required by the program in the Problem Definition. However, the implicit methods will require the solution of a tridiagonal system of linear equations, which we will solve using a tridiagonal Gaussian Elimination algorithm. A GKS version of VGA graphical display routines originally written by Steve Fiddes (Bristol) for an IBM-PC will also be used.

B.3 Software Design

The software is intended to be educative, therefore we will make use of graphical display. The problems chosen by the user admit analytical solutions, therefore we design the software so that the user can compare the numerical and exact solutions at each step of the solution process, visually.

The program comprises the following segments:

1. **User Problem and Data Selection** The user selects his choice of discretisation scheme, equation and initial data.
2. **Preparation** The initial solution is computed and the graphics is initialised.
3. **Problem Solution** The problem is solved and after each time-step the updated solution is shown on the graphics screen.
   (a) **Computation** The scheme is applied to march the solution from its current position in time to the next position in time.
   (b) **Display** The numerically-computed and exact solutions are displayed for purposes of comparison.
4. **Program Termination** The graphics is closed before the program finishes.

B.3.1 User Problem and Data Selection

For simplicity, the user will enter all input to the program from the keyboard. Where the user has a choice in input, e.g. in method selection, the full list of options will be displayed.

B.3.2 Preparation

The initial solution is computed. This is governed by the user's choice of initial data. The necessary GKS calls are made to set up the graphics and to open the display window.

B.3.3 Problem Solution

The solution procedure involves marching the solution forward in time in well-defined time steps. The computed solution and exact solution will be displayed at each time step, for comparison.

This procedure suggests the following loop structure for the main body of the program:

1. Compute the time step and increment the total time.
2. Compute the exact solution at the new time level.
3. Compute the numerical solution at the new time level, using the method selected by the user.
4. Display the exact and computed solutions at the new time level.

This loop is executed as long as the new time level is less than the specified maximum. The program will exit from the loop if the solution is known to be diverging and go directly to the segment of the software where the program is terminated. While it might be tempting to just stop the program at this point, from the point of view of software maintenance it is much simpler if there is only one point of entry and one point of exit to each subprogram.

B.3.4 Program Termination

The program will terminate when the main computation loop is finished. The necessary GKS calls to close the graphics window will be made first.

B.4 Software Implementation

We make the following choices to aid portability and robustness. Three parameters are defined: the screen input and output stream numbers, and the maximum problem size. The streams are system dependent. The user only has to change one parameter statement when moving the software from one platform to another. The maximum number of grid points allowed (NMAX) is declared as a parameter so that it can be used in the dimensioning of arrays. Again, this simplifies altering the software to choose either a larger or smaller problem size.

The variable IPROB, which indicates the method chosen by the user is stored in the COMMON block PROB.

The graphical calls are to RAL GKS.

The coding deviates from the strict ANSI standard in the use of the INCLUDE statement. This statement is used for ease of software maintenance. It can be replaced by the explicit insertion of the PARAMETER statements and COMMON block using a utility from TOOLPACK.

B.5 User Guide

B.5.1 How to Use flux

The user gives the command flux to run the program. He is then prompted for the required input data: the numerical method, the CFL number and the problem to be solved. The solution is displayed on the screen as it is being computed and the user hits the return key when the solution is finished.

B.5.2 Description of Output

The output from this program is entirely graphical. Once the user has specified the input data, a graphics window is opened to which two solutions are drawn at each time step in the solution process. The upper is the exact solution to the selected problem and the lower is the solution computed by the numerical method.

B.5.3 Error Reporting Instructions

The user input is checked as it is entered; any inconsistent values are trapped and the user is asked to re-enter suitable values. The permitted data is:

- **method type** This must have a value between 1 and 7, since there are seven methods to choose from.
- **CFL Number** This must have a value greater than zero.

- **Mesh Points** This must have a value between 2 and NMAX, the specified maximum number of nodes. For the current value of NMAX, see the implementation details below.

At each time step the solution is tested and if it found to be diverging the program is stopped. The test is performed in subroutine BANG and it checks if the numerical solution lies outside the range -3 to 3.

B.5.4 Implementation Details

There are three files for the SUN 3 implementation: flux.f, prob.common and Makefile.

flux.f contains all the Fortran source for the program.

prob.common contains the machine dependent variables and the only common block used in the program, PROB, which stores the variables IPROB (the user selected problem). The machine and implementation dependent variables are:

NMAX The maximum number of mesh points accepted by the program. It is currently set to 500.

TERMIN The stream number for terminal input. It is currently set to 5 for the SUN3 implementation.

TEROUT The stream number for terminal output. It is currently set to 6 for the SUN3 implementation.

Makefile The makefile will have to be modified so that it loads the GKS graphics from wherever they are stored on your own computing system.

B.6 Software Release

Since the software plots the exact solution in addition to the computed solution at each time step, it is not appropriate to include a set of test problems and their sample outputs for the program.

B.7 Software Source

B.7.1 The Fortran Source

The documented source of parts of flux are given below. The subroutines have been documented with the standard header and comments are given to document the programs operations.

```
      PROGRAM FLUX
*-----------------------------------------------------------------------*
*                                                                       *
* PURPOSE                                                               *
*     To provide an algorithmic testbench for the one-dimensional       *
*     linear advection and Burgers' equations.                          *
*                                                                       *
* HISTORY                                                               *
*     S P Fiddes (Bristol)    21 July 1991        SUN3                  *
*     C J Fitzsimons (RAL)    29 November 1991    SUN3                  *
*                                                                       *
* ARGUMENTS                                                             *
*     None.                                                             *
*                                                                       *
* LOCAL VARIABLES                                                       *
*     I       INTEGER. Loop counter.                                    *
*     J       INTEGER. Dummy variable. Used in a loop to slow down the  *
*             solution process to help the visual display.              *
*     IMETH   INTEGER. Identifies the discretisation scheme chosen. For *
*             a list of its possible values, see the comments to        *
*             routine TITLES.                                           *
*     N       INTEGER. The number of grid points chosen by the user.    *
*     X       REAL. The grid points on which the problem is solves.     *
*     UOLD    REAL. The computed solution at the current time.          *
*     UNEW    REAL. The computed solution at the next time step.        *
*     UENEW   REAL. The exact solution at the next time step.           *
*     UEOLD   REAL. The exact solution at the current time.             *
*     CMAX    REAL. Function which computes the maximum of the absolute *
*             value of the derivative of the flux with respect to u.    *
*     T       REAL. The current time.                                   *
*     DT      REAL. The time step being taken in solving the problem.   *
*     DTDX    REAL. The time step divided by the grid spacing, i.e.     *
*             dt/dx.                                                    *
*     DX      REAL. The distance between grid points.                   *
*     CFL     REAL. The Courant-Friedrichs-Lewy number (specified by    *
*             the user).                                                *
*     BANG    LOGICAL. When set to .TRUE. this indicates that the       *
*             solution is growing unphysically.                         *
*                                                                       *
* ROUTINES CALLED                                                       *
*   SYSTEM: FLOAT                                                       *
*           SQRT                                                        *
*     FLUX: BMWM                                                        *
*           CHECK                                                       *
*           CMAX                                                        *
*           DRAW                                                        *
*           ENDDEV                                                      *
*           EXACT                                                       *
*           FTCS                                                        *
*           IMP1                                                        *
*           IMP2                                                        *
*           INIT                                                        *
*           LAXWEN                                                      *
*           PICCLE                                                      *
*           MAC                                                         *
*           SETDEV                                                      *
*           TITLES                                                      *
*           UPWIND                                                      *
*      GKS: GCLSG                                                       *
*           GCRSG                                                       *
*           GDSG                                                        *
*           GUWK                                                        *
*                                                                       *
* INPUT/OUTPUT                                                          *
```

```
*     TERMIN  Unit number for terminal input. Parameter set in
*             prob.common.
*     TEROUT  Unit number for terminal output. Paramater set in
*             prob.common.
*
* ERROR ACTION
*     If the solution increases beyond an acceptable size (this is
*     checked in CHECK) the main iteration loop of the program is
*     exited.
*
* BUG FIXES
*     None.
*
* SPECIAL FEATURES
*     None.
*
*     PROGRAM FLUX
************************************************************************
      INCLUDE 'prob.common'
      INTEGER I, J, IMETH, N
      REAL X, UOLD, UNEW, UENEW, UEOLD, CMAX, T, DT, DTDX, DX, CFL
      LOGICAL BANG
      DIMENSION X (NMAX), UOLD (NMAX), UNEW (NMAX), UENEW (NMAX),
     1          UEOLD (NMAX)
      EXTERNAL BMWM, CHECK, CMAX, DRAW, ENDDEV, EXACT, FTCS, GCLSG,
     1         GCRSG, GDSG, GUWK, IMP1, IMP2, INIT, LAXWEN, MAC,
     2         PICCLE, SETDEV, TITLES, UPWIND
      INTRINSIC FLOAT, SQRT

*
* 1. USER PROBLEM AND DATA SELECTION
*
      CALL TITLES(CFL, IMETH, N)
*
* 2. PREPARATION
*
* Compute the inital solution and initalise the time.
*
      CALL INIT (X, DX, UENEW, UNEW, N)
      T = 0.0
*
* Set up the graphics window to display the results.
*
      CALL SETDEV
      CALL PICCLE
      CALL GCRSG(1)
*
* Display the initial data for both the numerical and exact solutions.
*
      CALL DRAW (X, UENEW, 2.0, 1, N)
      CALL DRAW (X, UNEW, 0.5, 2, N)
      CALL GCLSG
      CALL GUWK (1, 1)
*
* 3. PROBLEM SOLUTION
*
  10  IF (T .LT. 3.1) THEN
*
* Compute the time step and increment the time level.
*
        DT = CFL * DX / CMAX (UNEW, N)
        T  = T + DT
```

```
      DTDX = DT / DX
*
* Compute the exact solution at the new time level.
*
      CALL EXACT (T, X, UEOLD, UENEW, N)
*
* Compute the numerical solution at the new time level using the method
* chosen by the user.
*
* NOTE: It is in this part of the software that the user can introduce
* additional methods of his own choosing.
*
      IF (IMETH .EQ. 1) THEN
        CALL FTCS (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 2) THEN
        CALL MAC (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 3) THEN
        CALL UPWIND (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 4) THEN
        CALL IMP1 (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 5) THEN
        CALL IMP2 (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 6) THEN
        CALL BMWM (DTDX, UOLD, UNEW, UENEW, N)
      ELSE IF (IMETH .EQ. 7) THEN
        CALL LAXWEN (DTDX, UOLD, UNEW, UENEW, N)
      END IF
*
* Draw the exact and numerical solutions at the new time level.
*
      CALL GDSG (1)
      CALL GUWK (1, 1)
      CALL GCRSG (1)
      CALL DRAW (X, UNEW, 0.5, 2, N)
      CALL DRAW (X, UENEW, 2.0, 1, N)
      CALL GCLSG
      CALL GUWK (1, 1)
*
* This is included to slow down the program slightly, so that the
* user has time to study the solution at each time step.
*
      DO 1000 I = 1, 5000
        J = SQRT (FLOAT (I))
 1000 CONTINUE
*
* If the solution is diverging we want to stop the computation and break
* out of this loop.
*
      CALL CHECK (UNEW, N, BANG)
      IF (BANG) THEN
        WRITE (TEROUT, 6000)
        GO TO 20
      END IF
      GOTO 10
    END IF
*
* 4. TERMINATION
*
   20 CONTINUE
      CALL ENDDEV
      STOP
 6000 FORMAT('SOLUTION HAS DIVERGED')
      END
```

```
      SUBROUTINE TITLES (CFL, IMETH, N)
*-----------------------------------------------------------------------*
*                                                                       *
* PURPOSE                                                               *
*     To enter the user data to the program, i.e. the numerical method  *
*     to be used, the CFL number for the problem, the number of points  *
*     in space, and the choice of initial data.                         *
*                                                                       *
* HISTORY                                                               *
*     S P Fiddes (Bristol)     21 July 1991        SUN3                 *
*     C J Fitzsimons (RAL)     29 November 1991    SUN3                 *
*                                                                       *
* ARGUMENTS OUT                                                         *
*     CFL      REAL. The Courant-Friedrichs-Lewy number. Input by the   *
*              user. Must be positive.                                  *
*     IMETH    INTEGER. The number of the method selected by the user.  *
*              It takes the following values:                           *
*                                                                       *
*                1. Forward-time Centred-Space                          *
*                2. MacCormack                                          *
*                3. (Cole-Murman) Upwind Scheme                         *
*                4. Backward Euler Implicit with Centred Differencing   *
*                5. Backward Euler Implicit with Upwind Differencing    *
*                6. Beam-Warming                                        *
*                7. Lax-Wendroff                                        *
*                                                                       *
*     N        INTEGER. The number of space points in the problem. Input*
*              by the user. It must be less than NMAX and greater than  *
*              1.                                                       *
*                                                                       *
* LOCAL VARIABLES                                                       *
*     None.                                                             *
*                                                                       *
* ROUTINES CALLED                                                       *
*     None.                                                             *
*                                                                       *
* INPUT/OUTPUT                                                          *
*     TERMIN  Terminal input stream.                                    *
*     TEROUT  Terminal output stream.                                   *
*                                                                       *
* ERROR ACTION                                                          *
*     The user input is checked that it is within the permitted ranges.*
*     The user is asked to re-enter values for data outside the         *
*     following ranges:                                                 *
*                                                                       *
*       1 <= IMETH <= 7                                                 *
*       0 < CFL                                                         *
*       2 <= N <= NMAX                                                  *
*                                                                       *
*                                                                       *
* BUG FIXES                                                             *
*     None.                                                             *
*                                                                       *
* SPECIAL FEATURES                                                      *
*     None.                                                             *
*                                                                       *
*     SUBROUTINE TITLES (CFL, IMETH, N)                                 *
*************************************************************************
      INCLUDE 'prob.common'
      INTEGER IMETH, N
      REAL CFL

*
* User selects the method.
```

```
*
      WRITE (TEROUT, 6000)
  10  WRITE (TEROUT, 6001)
      READ (TERMIN, *) IMETH
      IF (IMETH.LT.1 .OR. IMETH.GT.7) THEN
        WRITE (TEROUT, 6002)
        GO TO 10
      END IF
*
* User selects the CFL number.
*
  20  WRITE (TEROUT ,6003)
      READ (TERMIN, *) CFL
      IF (CFL .LT. 0.0) THEN
        WRITE (TEROUT, 6004)
        GO TO 20
      END IF
*
* User selects the number of points in space.
*
   30 WRITE (TEROUT, 6005)
      READ (TERMIN, *) N
      IF (N.LT.2 .OR. N.GT.NMAX) THEN
        WRITE (TEROUT, 6002)
        GO TO 30
      END IF
*
* User chooses the equation and initial data.
*
      WRITE (TEROUT, 6006)
      WRITE (TEROUT, 6007)
   40 READ (TERMIN, *) IPROB
      IF (IPROB.LT.1 .OR. IPROB.GT.2) THEN
        WRITE (TEROUT, 6002)
        GO TO 40
      END IF
*
      RETURN
*
 6000 FORMAT (1X,//////////,
     1'   FLUX:  ALGORITHM TESTBENCH',
     2////,
     3'   METHODS AVAILABLE :-',/,
     4'                       1.     FTCS        ',/,
     5'                       2.     MACCORMACK  ',/,
     6'                       3.     UPWIND      ',/,
     7'                       4.     IMPLICIT1   ',/,
     8'                       5.     IMPLICIT2   ',/,
     9'                       6.     BEAM-WARMING',/,
     1'                       7.     LAX-WENDROFF',/,
     2///)
 6001 FORMAT ('  Please select method            :  ',$)
 6002 FORMAT ('  The number you entered is out of range.')
 6003 FORMAT ('  Please select CFL number        :  ',$)
 6004 FORMAT ('  The CFL number should be positive.')
 6005 FORMAT ('  PLEASE SELECT NUMBER OF POINTS  :  ',$)
 6006 FORMAT (1X,//////////////////////,
     1'  PROBLEM TYPES AVAILABLE :-',/,
     2////,
     3//,
     4'       1.     LINEAR ADVECTION EQUATION (RAMP)',/,
     5'       2.     NONLINEAR BURGERS''EQUATION (RAMP)'/,
     6'       3.     LINEAR ADVECTION EQUATION (4X(1-X))'/,
     7///)
```

```
 6007 FORMAT ('  PLEASE SELECT PROBLEM TYPE      : ',$)
*
      END

      SUBROUTINE CHECK(U,N,BANG)
*-----------------------------------------------------------------------*
*                                                                       *
* PURPOSE                                                               *
*     To determine if any of the first N entries of the array U lie     *
*     outside the range [-3,3]. If any does, BANG is set to .TRUE.      *
*     This routine is used to check if the solution is diverging.       *
*                                                                       *
* HISTORY                                                               *
*     S P Fiddes (Bristol)    21 July 1991        SUN3                  *
*     C J Fitzsimons (RAL)    29 November 1991    SUN3                  *
*                                                                       *
* ARGUMENTS IN                                                          *
*     U       REAL. The array the magnitude of whose entries is being   *
*             checked.                                                  *
*     N       INTEGER. The number of entries in U to be checked.        *
*                                                                       *
* ARGUMENTS OUT                                                         *
*     BANG    LOGICAL. Set to .TRUE. if any of the entries in U lies    *
*             outside the range [-3,3].                                 *
*                                                                       *
* LOCAL VARIABLES                                                       *
*     I       INTEGER. Loop counter over the entries in U.              *
*                                                                       *
* ROUTINES CALLED                                                       *
*   SYSTEM: ABS                                                         *
*                                                                       *
* ERROR ACTION                                                          *
*     None.                                                             *
*                                                                       *
* BUG FIXES                                                             *
*     None.                                                             *
*                                                                       *
* SPECIAL FEATURES                                                      *
*     None.                                                             *
*                                                                       *
*************************************************************************
      INTEGER N, I
      REAL U
      LOGICAL BANG
      INTRINSIC ABS
      DIMENSION U (N)

      BANG = .FALSE.
      I=1

   10 IF (I .GT. N) GO TO 20
      IF (ABS(U(I)) .GT. 3.0) THEN
        BANG = .TRUE.
        GO TO 20
      END IF
      I = I + 1
      GO TO 10
   20 CONTINUE

      RETURN
      END
```

B.7.2 The Include File

As part of the standard, although not part of the formal ANSI Standard, the use of *include files* has been suggested. The syntax of the *include* does vary between implementations, however it is an important aid in reducing the errors in data definitions. Below are shown the *include* files used in the flux program.

```
*-----------------------------------------------------------------------*
* INCLUDE file prob.common                                              *
*-----------------------------------------------------------------------*
*                                                                       *
* PURPOSE                                                               *
*     PROB Common Block and Parameter Statements                        *
*                                                                       *
* VARIABLES                                                             *
*     NMAX    Maximum number of intervals in solution                   *
*     IPROB   Problem number - indicates method                         *
*     TERMIN  Channel number for keyboard input                         *
*     TEROUT  Channel number for output to screen                       *
*                                                                       *
*-----------------------------------------------------------------------*
*
      INTEGER NMAX, IPROB, TERMIN, TEROUT
      PARAMETER (NMAX = 500, TERMIN = 5, TEROUT = 6)
*
      COMMON /PROB/ IPROB
*
*-----------------------------------------------------------------------*
* INCLUDE File graphic.common                                           *
*-----------------------------------------------------------------------*
*                                                                       *
* PURPOSE                                                               *
*     GRCONS Common Block. Contains information about graphics          *
*     output characteristics.                                           *
*                                                                       *
* VARIABLES                                                             *
*     ISTART                                                            *
*     ICOL    Current output colour                                     *
*     XNOW    Current value of x pen position                           *
*     YNOW    Current value of y pen position                           *
*     XMIN    Minimum x co-ordinate value                               *
*     XMAX    Maximum x co-ordinate value                               *
*     YMIN    Minimum y co-ordinate value                               *
*     YMAX    Maximum x co-ordinate value                               *
*     SCAL    Global scaling value                                      *
*                                                                       *
```

```
*-----------------------------------------------------------------------*
*
      INTEGER ISTART,ICOL
      REAL XNOW,YNOW,XMIN,XMAX,YMIN,YMAX,SCAL
      COMMON /GRCONS/ XNOW, YNOW, XMIN, XMAX, YMIN, YMAX, ISTART, SCAL,
     *               ICOL
*
```

B.7.3 The Makefile

During software development on UNIX (or similar) based systems it is advisable to construct and use a *Makefile* to automate the compilation and building of programs. The *Makefile* used for the flux program is given below.

```
#
# Make file for FLUX - Algorithmic testbench for 1D equations
#
# Program uses GKS to produce graphical output
#

OBJ_FLUX = flux.o

GRAPHICS=$(GKSPATH)/libgks-1.35.a -lsuntool -lsunwindow -lpixrect


FFLAGS = -O2 -fswitch
LFLAGS = -O2 -fswitch

.f.o:
f77 -c $(FFLAGS) $<

flux:   $(OBJ_FLUX)
f77 $(LFLAGS) -o flux $(OBJ_FLUX) $(GRAPHICS)
```

C Contacts

1. **Information Processing Limited** Eveleigh House, Grove Street, Bath BA1 5LR.
2. **NAG Limited** Wilkinson House, Jordan Hill Road, Oxford OX2 8DR.
3. **Programming Research Limited** Waynflete House, 74–76 High Street, Esher, Surrey KT10 9QS.
4. **Salford Software Limited** Venables Building, 5 Cockroft Road, Salford M5 4NT.