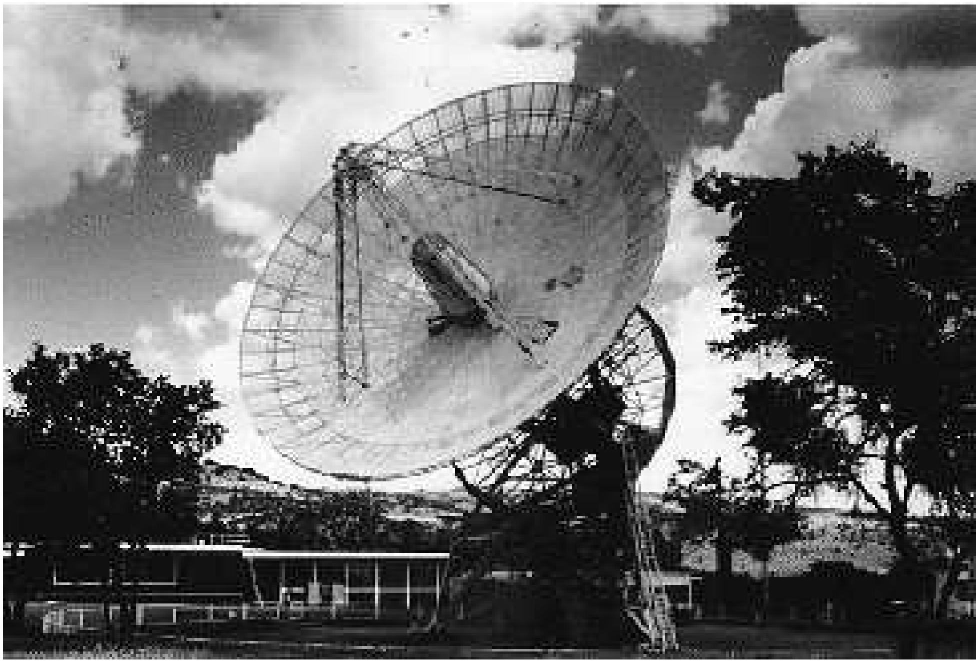
Hartebeesthoek 26m
New Computer Control System (NCCS)
CODE DEVELOPMENT ENVIRONMENT
version 1.0

Andy Götz

December 1, 1995



CODE DEVELOPMENT ENVIRONMENT

CODE DEVELOPMENT ENVIRONMENT
CODE DEVELOPMENT ENVIRONMENT
CODE DEVELOPMENT ENVIRONMENT
CODE DEVELOPMENT ENVIRONMENT
CODE DEVELOPMENT ENVIRONMENT
CODE DEVELOPMENT ENVIRONMENT

1 Introduction

The Hartebeesthoek 26m telescope computer control system is presently being redesigned and rebuilt - the so-called New Control Computer System (NCCS). In this process a lot of new code will developed and old code from the previous control system ported to the new operating system which has been chosen for the NCCS - Linux. This document makes a proposal of how to develop code under Linux and how to organise it. The proposals presented here apply equally to analysis programs and control programs.

2 Overview

The process of developing code for doing data analysis or control can be broken down into a number of well-defined steps. The aim of this document is to describe these steps and to propose a standard way of doing them. Standardising procedures will make it easier for all programmers to share and exchange software.

The basic processes which can be identified in developing software are :

1. **editing** - entering and modifying source code in human readable format in files,
2. **source code organisation** - how to organise source code into subdirectories
3. **compiling** - converting source code from human to machine readable format,
4. **make** - a tool used mainly for managing the compiling process,
5. **debugging** - that age old art of removing the creepy-crawlys from your software,
6. **source code management** - keeping track of multiple versions of source code files,
7. **distributing binaries** - the altruistic action of letting others benefit from your work,
8. **distributing source code** - the even more altruistic action of letting others continue from where you left off,
9. **documentation** - the hardest part of all (so they say) of putting into a human language what you have already put into machine language,
10. **backups** - saving all that hard work.

This document will now treat each of these topics individually and make proposals of how they should be handled at the Hartebeesthoek Radio Astronomy Observatory (HartRAO) in the context of the NCCS and Linux. The proposals are valid for programs written for doing scientific data analysis, programs written for controlling the telescope, utility programs, test programs and programs written elsewhere and which are ported to HartRAO. Where differences exist between the different environments this will be noted.

3 Editing

Editing under Linux is basically the same as editing under Unix. Under Linux the following editors (at least) exist :

- `vi` - the vanilla Unix editor from the 60's, a screen-based modal editor with cryptic commands, no online help, on Linux it is cursor sensitive when running under X11,
- `emacs` - the super duper configurable editor from the GNU project, includes an online help, fully screen based, LISP interpreter, lots of language sensitive configurations are available, somewhat cryptic commands,
- `joe` - a mode-less editor similar to PC editors, fully featured screen editor, users of Wordstar and Turbo language type editors will feel at home with this one,
- `xedit` - simple text editor for X.

Seeing as editing is such a personal matter [1]. it is not advisable to standardise on editors, every programmer should be free to make their own choice.

File names should follow the standard Unix conventions for suffixes e.g. C/C++ code must use the `.c` suffix, Fortran the `.f` suffix, include files the `.h` suffix and so on. Refer to the compiler man page when in doubt.

4 Source Code Organisation

Source code management refers to the standard procedure of organising source code files into different subdirectories according to type and purpose.

Code is normally organised into packets or units referred to as a **project** or a **package**. Once a project or package is identified it is given a software code name e.g. the control software for the NCCS can be grouped together under the code name `nccs`, another example is the plotting package from Caltech code named `pgplot`.

Every project must have it own directory structure and the same root directory. In this document we will refer to this as the `~project` directory.

Source code should be organised in identical subdirectory tree structures with respect to the root directory. This facilitates the task of program developers and installers. A proposed subdirectory tree structure is :

```
                            ~project
                                |
                                |
  ------------------------------------------------------------
  |        |       |       |        |            |          |
  |        |       |       |        |            |          |
README    bin     doc   include    lib          src     subproject
           |               |        |            |          |
           |               |        |            |          |
         --------         RCS   --------        RCS   -------------
         |      |               |      |              | | | | | | |
         |      |               |      |
       linux   sun4           linux   sun4
```

[1] As a system programmers at the MPIfR once said *"editing is a very personal matter, I wrote my own"*

Where the different subdirectories refer to the following :

- `~project` is the root directory for the project,
- `README` contains a description of the project and a brief description of the source code which can be found in this directory tree,
- `bin` contains the binaries (if any) of the sources; a separate subdirectory exists for each platform supported e.g. in the case of HartRAO this will be `linux` and `sun4`,
- `doc` contains all documentation related to this project e.g. white papers, users guide, porting guides, man pages etc.,
- `lib` contains all libraries generated and maintained by the project; a separate subdirectory exists for each platform supported e.g. in the case of HartRAO this will be `linux` and `sun4`,
- `src` is used for storing all source files; the golden rule is *"one source file for all platforms supported"*, differences between platforms are managed by conditional statements,
- `~subproject` is used for setting up a sub-project inside the main project, sub-projects have the identical subdirectory tree structure as the main project, there can be more than one sub-project,

The project root directory should be stored on file server accessible by all member of the project. In the case of HartRAO this could be `bootes:/usr/local/project`. Alternatively for a very big project it might be worthwhile creating a dedicated project directory in the `/home` area e.g. for the NCCS project a root directory `/home/nccs` coould be created on bootes.

All members of a project must belong to a common group. For a medium to large project a special group should be created for the project (usually with the same name as the project). The root directory and all subdirectories must have read, write and execute permission for all members of the project group. Files must have read permission for the group.

5 Compiling

Linux supports most of the GNU compilers. It has been decided at HartRAO to stick to two compilers : C and Fortran. Both compilers exist on the Sun and there should be no problem supporting both languages on both platforms.

6 Make

All projects must be managed with the wonderful Unix tool - `make`.

Make requires an input file (called `Makefile` or `makefile`) which describes the dependencies and rules for compiling. This file must be stored in the `~project/src` directory. For every target which has to be made list a dependency. Make maximum use of environment variables and implicit rules. If the implicit rules are not sufficient the define new ones (there are many implicit rules - to find out a list of implicit rules type `make -p | less`).

The following standard targets must exist in every `Makefile` :

- `all` - main target to do everything
- `install` - to install a finished product
- `clean` - to clean up after doing an all or an install

Multi-platform support can be handled in one of two ways :

1. **One master Makefile** for all platforms with conditional statements in the Makefile to handle differences between platforms; for this to work one has to use the GNU make (sometimes referred to as gmake),

   A simple example Makefile using this method is :

   ```
   Makefile
   --------

   #
   # Makefile for compiling the program "main" on multiple platforms
   # (sun and linux)
   #
   # author : andy gotz
   #
   # history : 7nov95 - created
   #
   #ifdef "sun4"
   CC = cc
   CFLAGS = -suncflags
   LFLAGS = -sunlflags
   INSTALLDIR = /usr/local/bin
   #else
   CC = gcc
   CFLAGS = -linuxcflags
   LFLAGS = -linuxlflags
   INSTALLDIR = /bootes/usr/local/bin
   #endif

   all : main

   main : main.o
           $(CC) $(LFLAGS) -o main main.o

   install : all
           cp main $(INSTALLDIR)

   clean :
           rm -rf *.o
   ```

2. **One Makefile per platform**, this is the simple way out but can lead to more work because there are more Makefiles to deal with.

   Using this approach the above example would be two files :

   ```
   Makefile.sun4
   -------------
   ```

```
#
# Makefile for comiling the program "main" on a sun4
#
# author : andy gotz
#
# history : 7nov95 - created
#
CC = cc
CFLAGS = -suncflags
LFLAGS = -sunlflags
INSTALLDIR = /usr/local/bin

all : main

main : main.o
	$(CC) $(LFLAGS) -o main main.o

install : all
	cp main $(INSTALLDIR)

clean :
	rm -rf *.o

Makefile.linux
-------------

#
# Makefile for comiling the program "main" on a sun4
#
# author : andy gotz
#
# history : 7nov95 - created
#
CC = cc
CFLAGS = -linuxcflags
LFLAGS = -linuxlflags
INSTALLDIR = /bootes/usr/local/bin

all : main

main : main.o
	$(CC) $(LFLAGS) -o main main.o

install : all
	cp main $(INSTALLDIR)

clean :
	rm -rf *.o
```

My own preference is for the first method.

7 Debugging

For linux use GNU's gdb or X based xgdb for C. Xxgdb is preferred in a windowing environment. There is no known solution for Fortran.

On the Sun use the dbxtool for C and Fortran programs.

8 Source Code Management

Use Revision Control System (RCS) to create master capies of all source files. Every time a modification is made to a source file then do a check in, check out. All source code must be under version control. Use minor version numbers to distinguish between different versions of the same file and major version numbers to distinguish between different major versions of the project.

For documentation of RCS do `man rcs`. Note for RCS to work there must be only ONE master version of the RCS source repository.

Example for the basic commands of rcs are :

1. **starting up** - before putting any file under revision control create a directory called `RCS` where all version files will be stored (`mkdir RCS`).
2. **checking in a file** - checkin and update the current version (`ci -u file`).
3. **making an intermediate checking in a file** - checkin and check the current version out with locking (`ci -l file`).
4. **checking out a specific version** - where the required version is $m.n$ (`co -rm.n file`).
5. **comparing the current version with the last checked in version** - prints the differences between the checked out and the latest checked in version (`rcsdiff file`).

Two golden rules for knowing when to checkin are (1) checkin a file when a significant modification has been made, and (2) checkin when you have typed in so much that you don't want to lose it.

Note RCS is not available for the sun's at present. If needed it can be easily got from the public domain and installed on the sun's.

9 Distributing Binaries

Binaries must be distributed to a major file server or to individual machines. I would suggest putting all binaries in `/usr/local/~project/bin` and `/usr/local/~project/lib` for libraries.

Distributed machines on the network can mount these directories to make sure that they always have the latest versions.

10 Distributing Source Code

This should be done via anonoymous ftp and WWW.

11 Documentation

Every project worth doing is worth documenting.
The following documents at least should exist for all software projects :

1. *User's Guide* - a pedestrian's guide to how to use the software.
2. *Programmer's Guide* - a programmer's guide to how the software has been written and how it works.
3. *Manual Pages* - online descriptions of how to start the software and do basic things.

The accepted format for these documents is LateX and/or HTML.

12 Backups

Distributed development (i.e. on local Linux boxes) will only work if there is an efficient and working backup system.
Need a central backup facility with a well-defined policy of when and how backups are done ... who is best qualified to write this - JQ ?

13 Recommended Reading

1. *Make: The GNU make Utility*
2. *GCC: Information about the gcc Compiler*
3. *GDB: The GNU Debugger*
4. *Emacs: The extensible self-documenting text editor*
5. `man rcs`