Starlink Project
Starlink System Note 78.1

Norman Gray, Peter W Draper, Mark B Taylor, Steven E Rankin

11 April 2005

The Starlink Build System

Abstract

This document provides an introduction to the Starlink build system. It describes how to use the Starlink versions of the GNU autotools (autoconf, automake and libtool), how to build the software set from a checkout, how to add and configure new components, and acts as a reference manual for the Starlink-specific autoconf macros and Starlink automake features.

It does not describe the management of the CVS repository in detail, nor any other source maintainance patterns.

It should be read in conjunction with the detailed build instructions in the `README` file at the top of the source tree (which takes precedence over any instructions in this document, though there should be no major disagreements), and with sun248, which additionally includes platform-specific notes.

Contents

Chapter 1
Introduction

The Starlink build system consists of

- the Starlink CVS repository;
- build tools, consisting of modified versions of the GNU autotools `autoconf` `automake` and `libtool`;
- configuration and bootstrap code, consisting of the `starconf` system and the scripts at the top level of the Starlink source tree; and
- the conventions for branching and tagging which are at present only documented in the wiki, on the pages `http://wiki.starlink.ac.uk/twiki/bin/view/Starlink/BranchingPolicy` and `http://wiki.starlink.ac.uk/twiki/bin/view/Starlink/CvsTagging` (you will currently need a wiki account to see these pages).

The repository is on the machine `cvs.starlink.ac.uk` – see Sec. 2.6 for further details.

The build system is intended to cover two main cases. Firstly, it covers the problem of building code directly from the CVS repository, both when building the entire software set from scratch (as during the nightly test build for example), and when working on a development version of a particular component, within the context of an otherwise built software set. This includes the problem of abiding by long-standing Starlink conventions on documentation, auxiliary tools and installation locations. The build system at present is concerned primarily with the 'classic' applications, which are the large volume of legacy Starlink code which, though it is no longer being developed, must still be maintained and be buildable by the community. It has no contact with the large volume of Java code, apart from having the same tagging and branching conventions, since the building of the Java applications is currently completely handled by the `ant` system.

Each component in the CVS repository is intended to be built separately, as opposed to being built only by a make running at the top of the tree. Many components require other components to be installed, and the role of the top-level makefile is to manage these dependencies, when it is necessary to build the entire tree from scratch. After a fresh CVS checkout, and presuming that any dependencies are in place, each component in the tree should be built with the sequence of commands

```
% ./bootstrap
% ./configure
% make
% make install
```

The 'install' target generated by Starlink automake installs the component as usual, but additionally creates a manifest file which is then installed in the `manifests/` subdirectory under the Starlink installation directory. The building of the whole tree from scratch is described in Sec. 3.2, and the procedure to build a single component, though little more complicated than described here, is discussed in more detail in Sec. 3.3. You should read both of these sections before trying the commands above.

Secondly, the build system handles the construction of portable source-code sets – distribution tarballs – so that individual components can be built by users, from source, as straightforwardly as possible. Since we are using the GNU autotools, much of this functionality comes for free, and we simply have to do the relatively small amount of configuration work required to ensure that the source set is complete (rather than requiring non-distributed build tools) and has clearly expressed dependencies between packages. It is an absolute requirement that the build process for users must consist of no more than

```
% ./configure
% make
% make install
```

with the usual options available for `./configure` and the usual relocations available for GNU-style makefiles. This document does not discuss building from a distribution tarball in any more detail than this, since there is really very little more detail to discuss.

The system does not cover the precise mechanism by which these tarballs are distributed to users, nor the means by which they are informed of, or helped to satisfy, the dependencies between components; but in generating manifests and assembling the component dependency graph, it provides the information which such a distribution portal would need.

In other sections of this document, we discuss the tools used to support developing within the CVS tree, including the specific starconf tools and an overview of the GNU autotools autoconf, automake and libtool (Sec. 2); we discuss the procedure for building applications within the repository (Sec. 3); and we describe the process of adapting a currently working package to the build system's conventions (Sec. 4).

1.1 Quick entry-points

The build system is intended to be a fairly integrated whole, and so this document is intended to be read straight through, rather than act purely as a reference manual. However it should be fairly well cross-referenced, and so depending on your motivations for reading this document, there is more than one reasonable entry-point.

- If you want to build and install a distributed source tarball, then you are probably reading the wrong manual, since this one contains rather more information than you need. To build the distribution you should not need to do more than `./configure; make; make install` – if you do need more than that, that's either a bug, or there's a readme you didn't read. If you need to know more about the options to `./configure`, then `./configure -help` should provide that. And if you really want or need the gory details, then Sec. 2.1.5 has them all.
- If you want to start building and using software from the repository, then start with Sec. 2.6 or Sec. 3.3 and work outwards from there.
- If you want to add another component to the repository, then you are probably going to have to read quite a lot of the document, but Sec. 4.1 and Sec. 4.5 (and indeed most of Sec. 4) are good places to start.
- There are some FAQs in Sec. 5.
- There is a reference for the Starlink autoconf macros in Sec. A.

Chapter 2
Tools

This section is concerned with the tools which make up the Starlink build system. There are generally very few adjustments you need to make to the program source code (though the autotools can help you manage such things as installation directories and platform dependencies), and the changes you will make are to the auxiliary files which manage the build, most prominently the `configure.ac` file which organises the configuration, and the `Makefile.am` file which controls generating the makefile which does the work.

The build system as a whole consists of a number of components:

The Starlink autotools These consist of versions of the GNU autotools, which are part of the Starlink CVS repository, and which are built and installed as part of the top-level bootstrap. They should be installed in `/star/buildsupport/bin`, or its equivalent if you have installed a set of tools in a tree other than `/star`. Since they have been extended and customised, they are essential; you cannot build the Starlink collection without them. There is an introduction to the autotools in Sec. 2.1, covering their interrelationships and basic usage.

We use (at the time of writing) an unmodified libtool, a moderately extended autoconf, and an extended and customised automake. The description below explains the use of these tools as modified; see Sec. B for details of the differences from the stock autotools.

The Starlink autoconf macros There is an extensive set of autoconf macros which support building Starlink applications. You use these as you would any of the standard autoconf macros, by invoking them within the `configure.ac` file.

The macros are described in detail in Sec. A.

The autoconf program which is used is not itself customised for Starlink use. Instead, the macros are installed using the standard autoconf extension mechanism, which uses the application *aclocal*. This is discussed in passing in Sec. 2.1, and we mention it only in passing here.

starconf This is the program which handles installing required macros, and configuring your directory ready to use the build system. Associated with *starconf* is the *starconf-validate* program which checks that your directory is configured correctly, and the program *./starconf.status*, which unpacks required files, and allows you to make the link back to the configuration used to build the software. All these are described in detail in Sec. 2.2.

2.1 Overview of the Autotools

The autotools as a group help you create software distributions which will build reliably on a large variety of platforms. They are most obviously associated with unix systems, but in fact support the larger class of POSIX systems which have an implementation of `/bin/sh` available – a class which includes Windows machines, through the cygwin environment.

The minimum required contact with the autotools is to generate the files which will let you configure a freshly checked-out directory before working on it, since these configuration files are not checked in to the repository. The `./bootstrap` script (see Sec. 2.2) will do this for you, but it can only do this if the autotools are installed (see Sec. 3.5).

The aim of the notes below is to give you an overview of autoconf and automake functionality, enough to make it possible for you to roughly understand the existing configuration files, and to make it easier to approach the full autoconf and automake documentation. We start off with a broad overview, and provide a few further details in the subsections which follow.

The two most obvious and important files are `configure.ac` and `Makefile.am`. The first tells autoconf how to make the `./configure` script which does the actual configuration work, and which will be distributed with your package, the second (assuming you're using automake rather than hand-maintaining the input Makefile) tells automake how to create the template makefile `Makefile.in`, which is also distributed with your package, and which is edited by the `./configure` script to (finally!) produce the `Makefile` which (finally!) builds the code on the user's system.

It's useful to illustrate the inputs and outputs of the various tools. First are autoconf and automake:

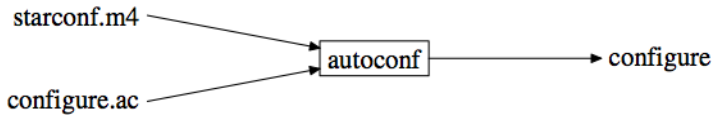

Figure 2.1: Inputs to, and outputs from, autoconf

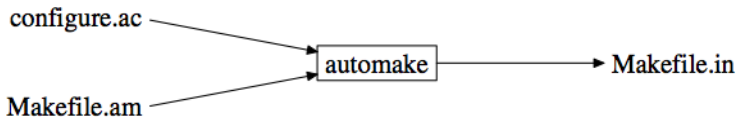

Figure 2.2: Inputs to, and outputs from, automake

As you can see, although `configure.ac` is most associated with autoconf, automake reads it, too, to discover for example whether you have invoked `AC_PROG_LIBTOOL` or `STAR_MONOLITHS`, and so whether it needs to add the appropriate support in the generated `Makefile.in`.

Once the `./configure` script is created, it is able to interrogate the system (the user's system, that is, not the developer's), and based on that edit the various `.in` files to produce their corresponding outputs.

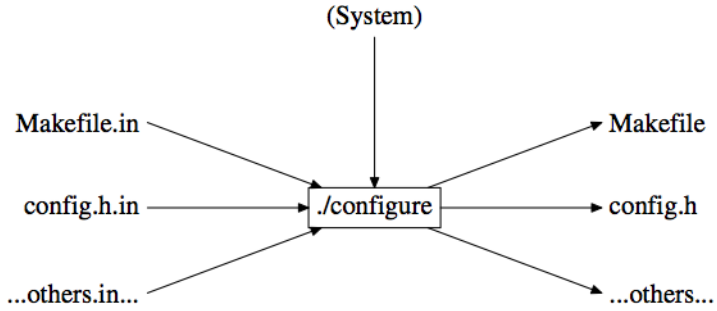

Figure 2.3: Inputs and outputs to configure

This diagram for `./configure` shows a file `config.h.in` being edited. This file – which we will have more to say about in Sec. 2.1.1 – is a template list of C preprocessor definitions, which can be `#included` in a C source file, and so used to configure the use of functions and definitions there. Like any of the other `.in` files, it is possible to maintain this by hand, but it is more reliable to maintain it using another application *autoheader*, which looks as follows.

The other file we have not mentioned yet is `starconf.m4`. This file contains macros (in the implementation language of autoconf, m4) which are used to extend autoconf's functionality, supplying the Starlink-specific autoconf support. You will not typically have anything to do with this file, and we mention it only to indicate schematically where (some of) the Starlink magic comes from. This file is not in your

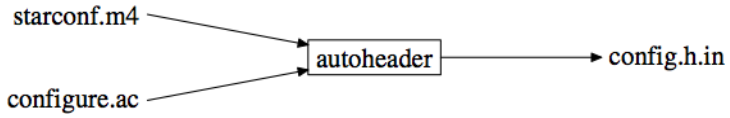

Figure 2.4: Inputs and outputs to autoheader

working directory, but is installed in the place the autotools expect to find it, by the *starconf* application described in Sec. 2.2. You may also notice a file `aclocal.m4` in your directory. This is a cache of autoconf macros, maintained by the autoreconf program; it should not be checked in.

Of the applications mentioned above, automake, autoconf and (usually implicitly) autoheader are run on the developer's machine before or while making a distribution; thus the products of these programs, `./configure`, `config.h.in` and `Makefile.in`, are included in the distribution, and the `.in` files edited by the `./configure` script into the `config.h` and `Makefile` which actually control the build.

We have elided some details here, for simplicity, and we could possibly have elided more, since you generally don't have to concern yourself with autoheader or `aclocal.m4`. You don't even have to worry about the dependencies between these files and applications, since the `Makefile.in` which automake generates knows about these dependencies and will generally keep things up to date for you. If you wish to be sure things are up to date, then you can simply run application *autoreconf* (see Sec. 2.1.4), which knows enough about the relationships between them to run each of the required ones in the correct order. This is likely the only one of the autotools commands which you will run yourself.

For more details on the relationships between these applications, and fuller diagrams, see the online copy of the book 'GNU Autoconf, Automake, and Libtool', and in particular Appendix C of that book, 'Generated file dependencies'.

Comprehensive details of the autotools are to be found in their respective manuals, which are available online at

- `http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html`
- `http://www.gnu.org/software/automake/manual/`
- `http://www.gnu.org/software/libtool/manual.html`

The autoconf manual is relatively clear, once you have a basic idea of what it's trying to do, but it's more effective as a reference manual than as a tutorial. The automake manual should be regarded as a reference manual only: you might not guess from the manual, that automake will make your life simpler, rather than full of anguish, suffering and confusion. It doesn't really matter how good or bad the libtool manual is, since if you discover you have to look at it, you already know your future holds pain, and the manual isn't going to make things better or worse.

It's also worth looking at the 'Release Process' section of the GNU Coding Standards document. Though we are not necessarily following these standards, this section both describes what is conventional packaging in the open-source world, and outlines the conventions which the autotools are designed to support.

2.1.1 Autoconf

We use a slightly extended version of autoconf. See `autoconf -version` for the base version number.

The goal of autoconf is to help you make your program portable, by allowing you, at build time, to adapt the program to the facilities available on the machine on which the program is being built.

You control this through a script called `configure.ac` (which was called `configure.in` in older versions of autoconf). The autoconf program produces from this a very portable `/bin/sh` script called `configure`,

which is distributed as part of your package, and which the person building the package runs as the first part of the `./configure; make; make install` incantation.

The `./configure` script probes the system it is running on, finding compilers, testing the behaviour of the local unix system, testing whether specific include files exist or not, testing whether required functions are available and working as expected, and managing some configuration by the user, such as allowing them to specify where the package is to be installed.

This information is only useful if it can be communicated to your program in some way. This is done by the configure script by editing the values of certain 'substitution variables' into a list of template files.

For example, you might have a version header file `version.h.in`, containing the line

```
const char version[] = "@PACKAGE_VERSION@";
```

The configure variable `PACKAGE_VERSION` is one of those substituted by default, and if this file were listed as one of those to be substituted (by mentioning it in the autoconf macro `AC_CONFIG_FILES(version.h)`), then a file `version.h` would be created containing the contents of the `version.h.in` file with the `@PACKAGE_VERSION@` substituted by the version number declared within the `configure` file.

Although this substitution process can be done for any template file, there are two template files which are used particularly often.

The first is `Makefile.in`, which is a skeleton makefile which you might write by hand (though see the discussion of automake in Sec. 2.1.2). There is an example of this at the top level of the Starlink build tree. A typical `Makefile.in` might include lines like

```
LN_S = @LN_S@
CC = @CC@
myfile.o: myfile.c
        $(CC) -o myfile.o myfile.c
```

That, combined with autoconf macros `AC_PROG_CC` and `AC_PROG_LN_S`, would allow you to produce a file `Makefile` which was customised to use the C compiler appropriate to the local environment, and which had, in the makefile variable `{$(LN_S)` a command which makes links between files if the local platform supports that, or makes hard links or simple copies if that is all that is possible.

As well as configuring the makefile, you may also want to configure the source code, so that you can take different actions in your code depending on whether certain functions or headers are available. That is is most often done via a particular configured file, conventionally named `config.h`. This second way of communicating with your source code has the `config.h` file substituted with a number of C preprocessor `#define` statements. For example, if you included in your `configure.ac` file the lines:

```
AC_CHECK_HEADERS([sys/wait.h])
AC_CHECK_FUNCS([strchr])
```

then the configure script generated from that source would include a test of whether the `sys/wait.h` header file was available, and whether the `strchr` function was available in the C library. If so, the resulting `config.h` file would include the lines

```
#define HAVE_SYS_WAIT_H 1
#define HAVE_STRCHR 1
```

Whereas most configured files are substituted as a result of being mentioned in a `AC_CONFIG_FILES()` macro, the `config.h.in` input file is configured through `AC_CONFIG_HEADERS(config.h)`, which does its configuration in a slightly different way.

After including this file into your source code with:

```
#include <config.h>
```

you can adjust the compiled code with suitable `#if` preprocessor conditionals, such as

```
#if HAVE_SYS_WAIT_H
#include <sys/wait.h>
#endif
```

It is possible to maintain the `config.h.in` file by hand, but generally better to generate it by using the *autoheader* application which is part of autoconf. This scans your `configure.ac` and extracts into `config.h.in` mentions of all those preprocessor defines it finds. Autoheader knows about macros such as `AC_CHECK_HEADERS` above; if you wish to add further switches to the `config.h.in` file, you should do so by calling the autoconf macro `AC_DEFINE`. See the section 7.1 Defining C Preprocessor Symbols in the autoconf manual for further details. Autoheader is one of those applications run on your behalf by autoreconf (see Sec. 2.1.4).

Note the angle brackets round `config.h`: this is preferred to double quotes, as it gives the makefile the option of controlling where the source code is; in the simplest case this is handed by simply adding `-I.` to the compile line. We don't take take advantage of this particular flexibility, but it is a good habit to get into.

An illustrative `configure.ac` might look like this (with line numbers inserted):

```
1   AC_INIT(mypack, 1.2, starlink@jiscmail.ac.uk)
2   AC_CONFIG_SRCDIR(src.c)
3   STAR_DEFAULTS
4   AC_PROG_CC
5   AC_PATH_PROG(PERL, perl)
6   AC_CHECK_HEADERS([sys/time.h])
7   STAR_MESSGEN(mypack_err.msg)
8   AC_CONFIG_HEADERS(config.h)
9   AC_CONFIG_FILES(Makefile)
10  AC_CONFIG_FILES(myscript.pl, [chmod +x myscript.pl])
11  AC_OUTPUT
```

Line 1: This line is required. It declares the package name, version number, and bug-report address. Each of these is available for substitution via the substitution variables `PACKAGE_NAME`, `PACKAGE_VERSION` and `PACKAGE_BUGREPORT`.See autoconf 4.1 Initializing configure

Line 2: This is largely a sanity check, and produces an error if the named file (in this case `src.c`) is not in the source directory. The source directory is typically the current directory, but you can specify a different one using the `-srcdir` option to `./configure`, if you have a good reason for doing that. See autoconf 4.3 Finding configure Input

Line 3: This macro is one of the Starlink extensions, and the only required Starlink macro. It sets up the defaults required for building Starlink applications, and assures the `starconf` program that it's being run in the correct directory. See Sec. 2.2 for a description of the starconf application, and Sec. A for details of the associated macros.

Line 4: This finds a working C compiler, and prepares to substitute it in the substitution variable `CC`. Presumably the Makefile.in has a line `CC=@CC@`. See autoconf 5.10.3 C Compiler Characteristics

Line 5: Find a binary called `perl` in the current `PATH` and assign the substitution variable `PERL` the full path to it. The most common way of using this would be for the file `myscript.pl.in` to start off with the line

```
#! @PERL@ -w
```

so that the script ends up with the correct full path. See line 10 and autoconf 5.2.2 Generic Program and File Checks

Line 6: check that the system has an include file `sys/time.h` in the include path, and if it has, make sure that the cpp variable `HAVE_SYS_TIME_H` is defined. If this were a real configure file, you would likely have several other header tests here (in a space-separated list, surrounded by square brackets), and cpp branching inside your source code to handle the various cases. See line 8 and autoconf 5.6.3 Generic Header Checks.

Line 7: Another Starlink extension. It declares that this package has a set of ERR messages in the given file, and that autoconf should check the location of the messgen application. The argument is in fact optional (it merely causes the files to be declared as pre-distribution files – see Sec. A.27; this line should be partnered by a declaration of the variable `include_MESSAGES` in the corresponding `Makefile.am`. See Sec. A.23 for fuller details.

Line 8: This is the macro that makes cpp configuration information available, by editing the header file `config.h.in` (this file name is conventional, but not required). See autoconf 4.8 Configuration Header Files. If you want to put extra information into this file, use the `AC_DEFINE` macro: a declaration like `AC_DEFINE(SPECIALCASE,1)` would insert `#define SPECIALCASE 1` into the `config.h`; autoheader also spots this and puts a suitable template into `config.h.in`. See the autoconf manual, section Defining C preprocessor symbols, for further details.

Line 9: This does the work of substituting the results of the various tests into the files being configured. For each of the files named in the (space-separated list) argument, which most typically includes `Makefile`, autoconf expects to find a file of the same name but with `.in` appended, and this is the file which has the substitutions edited in. (Automake also looks at this line, and if it sees a `Makefile` mentioned here, it looks to see if there is a corresponding `Makefile.am` already present, and if so, recurses)

Line 10: This is a variant of line 9. The `AC_CONFIG_FILES` macro takes a second argument consisting of one or more lines of shell script to post-process the file in question, in this case making sure that the generated file is executable.

Line 11: This is the line which *really* does the work.See 4.4 Outputting Files.

Of this script, it is lines 1, 2 and 11 which are absolutely required, along with something like line 9 to make the configure script do something useful.

It is useful to think of the `configure.ac` file as being the template for the final `./configure` script, with the autoconf macros simply expanding to (large chunks of) shell script during translation by the autoconf program. This isn't the whole truth, but it suffices for almost all purposes. About the only place where this view might deceive you is if you wished to modify a file after it was generated by `AC_CONFIG_FILES` for example; if you did that immediately after the `AC_CONFIG_FILES` line it would fail, since the code which generates the files is in a different place from the `AC_CONFIG_FILES` line – that is why `AC_CONFIG_FILES` has its second argument. With this view it is natural that you can add any other shell scripting to your configure.ac, adding any tests and switches you fancy. You communicate the results of your scripting to the output files by making shell variables into substitution variables, and the way you do that is by calling `AC_SUBST` on them. Thus, after

```
wibble=''
... # lots of clever scripting
AC_SUBST(wibble)
```

any occurrences of the string `@wibble@` in the substituted files will be replaced by the value of `wibble` at the *end* of the `./configure` script.

2.1.1.1 M4 syntax and traps

Autoconf does its work by running the `configure.ac` file through the text processor GNU `m4`, and this can cause occasional surprises.

M4 is a macro language, intended for very much the sort of rewriting that autoconf does. When m4 processes a file, anything at all that looks like an m4 macro is substituted, so that there is no macro invocation character like the backslash in TEX;. Macros can take arguments, delimited by round brackets and separated by commas, as illustrated above.

The first oddity is to do with comment characters. The default m4 comment character is `#`, but since this is also the shell script and makefile comment character, autoconf makes it an ordinary character. Thus to add comments to a `configure.ac` file which won't make it into the `./configure` file, you should prefix them with the m4 *macro* `dnl` (which means 'delete to next newline'), as in

```
dnl   This is a comment
AC_INIT(...)
```

There's no harm in including `#`-style comments in your `configure.ac` file and allowing these to be passed through to the output file, since 'got-here' type comments can sometimes help you debug the `./configure` script.

The default m4 quote characters are the left and right single quotes, but autoconf changes these to left and right square brackets. You need to use these in two circumstances, firstly when you have a multi-word argument to a macro, and particularly when that argument contains a comma, and secondly if a piece of text looks like a current m4 macro. In general, if you need to care about m4 quoting rules, you're in trouble, but see section 8.1 M4 Quotation in the autoconf manual, for some advice.

2.1.2 Automake

The most typical use of autoconf is to configure a file `Makefile.in`. You can write this yourself, as described in the discussion of autoconf, but since so much of a typical makefile is boilerplate, automake exists to write this boilerplate for you. This has the additional advantage that we can support and enforce Starlink-specific conventions with a customised installation of automake. We (currently) use an adapted version of automake – see `automake -version` for the version number, and Sec. B for a summary of the differences.

The file which controls automake is `Makefile.am`. Automake reads both this file and `configure.ac`, and based on these emits a `Makefile.in`. It is this latter file which you distribute, and which is substituted at build time by the `./configure` script which autoconf generates in turn.

The resulting `Makefile` has rules to do all the required building and installation in a very portable fashion, as well as targets to make distributions (`make dist`), do tests (`make check`), clean up (`make clean`, `make distclean` and `make maintainer-clean`) and, in the case of Starlink automake, do an install along with an installation manifest.

The `Makefile.am` script consists, in its simplest form, of a sequence of declarations of the relationships between the source files in your distribution and the programs and libraries they are intended to produce. For example, here is the `Makefile.am` for the PAR library, slightly edited:

```
## Process this file with automake to produce Makefile.in

lib_LTLIBRARIES = libpar_adam.la
libpar_adam_la_SOURCES = $(F_ROUTINES)

include_HEADERS = $(PUBLIC_INCLUDES)

F_ROUTINES = \
     par1_menu.f \
     par_cancl.f \
     [blah...]
```

```
PUBLIC_INCLUDES = \
    PAR_ERR \
    PAR_PAR \
    par.h \
    parwrap.h \
    par_err.h \
    par_par.h

BUILT_SOURCES = PAR_ERR par_err.h
```

Overall, you can see that this automake source file is syntactically a makefile – the statements in this example look like makefile variable settings, and it is possible to put makefile rules in the `Makefile.am` file, though that is not illustrated here. This is slightly deceptive, however, and while it was useful to think, above, of the `configure.ac` file as being the template of the eventual `./configure` script, for automake you should think of the match between the input syntax and the makefile as a happy coincidence. You provide information to automake through the `Makefile.am` file, and based on that it then emits the `Makefile.in` it thinks you want (or need, at least).

The first line is a conventional comment. It starts with a doubled hash mark `##`, which causes automake to discard the text after it; lines beginning with a single comment character are copied verbatim into the generated `Makefile.in`.

The next stanza declares that there is to be a library `libpar_adam.la`, and that the sources for this file are in the 'makefile variable' `F_ROUTINES`.

Though the variables `F_ROUTINES` and `PUBLIC_INCLUDES` are specific to this makefile, and arbitrary, the other variable names have both structure and meaning for automake.

The variable `lib_LTLIBRARIES` consists of the 'prefix' `lib` and the 'primary' `LTLIBRARIES`. The primary tells automake that you wish to build the libtool library `libpar_adam.la`, and the prefix indicates that this is to be installed with the other libraries (the variable `pkglib_LTLIBRARIES`, for example, would tell automake that you wanted to install the result in a package-specific library). For each of the libraries listed in this variable, separated by spaces, there must be a corresponding `_SOURCES` variable, which has the 'primary' `SOURCES` and a prefix formed from the library name, which lists the set of source files for that library. The prefix must be canonicalised by replacing with underscores everything other than letters and numbers. As well as declaring the files which are to be compiled into the library, this indicates to automake that these source files are to be distributed as part of the final tarball, and that it must emit makefile rules to install the library in the correct place, with the correct installation commands. For the list of such associated primaries, see section Program and Library Variables of the automake documentation.

By default, libtool builds both static and shared libraries. You can control this if necessary with the `-enable-shared` and `-disable-shared` options to `./configure`, and with the `AC_DISABLE_SHARED` autoconf variable. For further details see the documentation for `AC_PROG_LIBTOOL` in the Libtool manual.

If we only wanted to build static libraries, we would replace this line with `lib_LIBRARIES = libpar_adam.a`, and the given library would be built in the usual way, without involving libtool.

The `LIBRARIES` and `LTLIBRARIES` primaries can have any one of the prefixes `libdir`, `pkglibdir`, `check` or `noinst`, with the latter indicating that the library should be built but not installed. Having non-installed libraries can be useful when you are building a library conditionally, or in stages. Libtool refers to these as 'convenience libraries', and they are discussed in section Libtool Convenience Libraries of the automake manual.

The other very important primary (not shown here) is `PROGRAMS`, which describes programs which are to be built and installed (the special Starlink case of monoliths is described below). This can have the prefixes `bin`, `sbin`, `libexec`, `pkglib`, `check` or `noinst`: `check` is for tests, and is described in Sec. 4.4; `noinst` indicates that the program should be built but not installed, and is useful for programs which are used during the build – for generating header files for example – but which are not part of the final product. There is no single standard list of prefixes, since each primary supports only a subset of them

(you cannot declare `bin_LIBRARIES`, for example), but several are mentioned in automake's What Gets Installed, and the directories in question are discussed in autoconf's 4.7.2 Installation Directory Variables.

The `include_HEADERS` line is similar: it indicates that the listed files are to be distributed, and are to be installed in the correct place for include files.

The final line which is significant to automake is the `BUILT_SOURCES` line. This says that, even though `PAR_ERR` and `par_err.h` are to be installed and distributed, they are not actually genuine source files, but must be built; adding the `BUILT_SOURCES` line forces automake to add a dependency for these files at an artificially early stage. It is only rather rarely necessary to include a line like this. If one of the source files were in this category, then it would naturally be built when it was required, without any need to add it to `BUILT_SOURCES`. As it happens, there is no rule in this file for building these targets; that is added automatically by automake when it spots that the `STAR_MESSGEN` macro has been used in the `configure.ac` file which partners this `Makefile.am`. In a more general case, however, you would add a make-style rule to the `Makefile.am` file to build these files from their (presumably undistributed) sources. See also Built sources in the automake manual.

For a second example, we can look at the `Makefile.am` for the `sst` application (this is a non-distributed application for building documentation).

```
## Process this file with automake to produce Makefile.in

bin_SCRIPTS = start

bin_MONOLITHS = sst_mon
sst_mon_SOURCES = \
    sst_mon.f \
    $(sst_mon_TASKS:=.f) \
    $(SUBSRC) \
    $(SUBCSRC) \
    $(PRIVATE_INCLUDES)
sst_mon_TASKS = forstats procvt prohlp prolat propak prohtml
sst_mon_LDADD = $(LDADD) `fio_link`


SUBSRC = sst_clean.f sst_fwild.f sst_latex.f sst_puts.f  sst_trcvt.f \
         sst_cntac.f sst_get.f sst_latp.f sst_rdad1.f sst_trhlp.f \
         [blah...]

SUBCSRC = find_file.c

PRIVATE_INCLUDES = SST_PAR SST_SCB

# special installation directory (but see discussion below)
sstsupportdir = $(bindir)
sstsupport_DATA = sun.tex sst.tex layout.tex forstats.dat html.sty

# The `start' script needs to have installation locations edited into it
edit = sed \
    -e 's,@bindir\@,$(bindir),g' \
    -e 's,@VERSION\@,$(VERSION),g'
start: start.in
    rm -f start.tmp start
    $(edit) \
        -e 's,@edited_input\@,start: produced from start.in by Makefile.am,' \
        $(srcdir)/start.in >start.tmp
    mv start.tmp start

EXTRA_DIST = start.in $(sstsupport_DATA)
```

This makefile configures and installs a script, builds and installs a monolith, and adds some supporting data files in a non-standard place.

The SCRIPTS primary indicates to automake where and how to install the `start` script. The `start` script must be generated, since it is to include the version number and installation location. Since it includes the installation location, it should *not* be generated at configure time or install time, but instead at make time, so that the user is free to specify one installation prefix at configure time (through `./configure -prefix=www`), override this with another prefix at build time (through `make prefix=xxx`) and specify a *different* one – presumably a staging location or something similar – at installation time (through `make prefix=yyy install`). It is the prefix at *build* time that is to be baked into the code, if any such baking has to be done. This is one of the GNU conventions mentioned in Sec. 2.1, and is discussed in a little more detail in section 4.7.2 Installation Directory Variables of the autoconf manual. This is why we have to include a makefile-style rule for deriving the file `start` from its template `start.in`. This substitutes in the values of the makefile variables `bindir` and `VERSION`; these get their values, in the resulting `Makefile`, by having those values substituted in to the generated `Makefile.in` by the generated `./configure` script; the careful escaping of the @-signs in the sed command is to match the `@...@` in `start.in` while *not* matching this in `Makefile.am` or the resulting `Makefile.in` (yes, this can get a little confusing).

This `Makefile.am` also declares a single monolith (this and the TASKS primary below are obviously part of the Starlink extensions to automake) and its associated SOURCES, along with its component tasks. For fuller details of the monoliths support, see Sec. 3.4. This incidentally illustrates that automake allows variable reuse and rewriting very similar to that supported by make.

When we are linking this monolith, we need to add a couple of link options, and we do this with a LDADD primary associated with the `sst_mon` prefix. The extra link options we declare here are added to the eventual link command, replacing the default set of option options `$(LDADD)`, so we include that variable in our version. We also add `‘fio_link‘`, because this monolith needs to be linked against the FIO library (automake constrains what can appear in the LDADD variable, and the `‘fio_link‘` is permissable only in Starlink automake – see Sec. B for details).

The variables SUBSRC, SUBCSRC and PRIVATE_INCLUDES are purely user variables, with no special meaning to automake.

The next bit of magic in this file is `sstsupport_DATA`. In this particular case, we want to install data files in the same location as the binaries (why, and whether this is a good thing, is not for us to worry about here). The obvious thing would be to say `bin_DATA = sun.tex ...`, but automake forbids that particular combination of prefix and primary, precisely because it wouldn't generally make sense (see Architecture-independent data files in the automake manual). Instead, we can take advantage of an escape-hatch that automake provides: a given prefix, such as `sstsupport`, is valid if you also define a variable of the same name, but with `dir` appended. Thus we define `sstsupport_DATA` and `sstsupportdir`, and define the latter to have the same value as the `$(bindir)` makefile variable, as eventually substituted (see section The Uniform Naming Scheme in the automake manual).

If you look at the real `sst` component's `Makefile.am` file, you will see that it does in fact use `bin_DATA` rather than the `sstsupport_DATA` illustrated above. This is because the `configure.ac` in the `sst` component declares the option `STAR_DEFAULTS(per-package-dirs)` (see Sec. A.18) which, amongst a couple of other magic features, causes the variable `bin_DATA` to become legal.

This is a rather special case, and you should not have to do this sort of semi-licensed hacking very often. In particular, you do not need to do it in the case of the Starlink standard directories `/star/docs`, `/star/etc`, `/star/examples` and `/star/help`, since you can give these directories as prefixes to the DATA primary. As an example, the HDS makefile installs a file containing the information it has determined about the build machine's floating point model, and this is declared there as follows:

```
noinst_PROGRAMS = hds_machine
starhelp_DATA = hds_machine.txt

hds_machine_SOURCES = hds_machine.f
```

```
hds_machine_LDADD = libhds.la ‘ems_link‘ ‘chr_link‘ ‘cnf_link‘

hds_machine.txt: hds_machine
        ./hds_machine >$@
```

This is enough to build the `hds_machine` application at install time, run it, and install the results in `/star/help` (with the obvious changes for prefixes `stardocs`, `staretc` and `starexamples`).

The remaining magic in this file is the `EXTRA_DIST` variable. For all its cleverness, automake cannot work out that the `start.in` source, nor any of the `sstsupport_DATA` files are to be distributed, and you need to declare that explicitly by setting the `EXTRA_DIST` variable. Though it is of course deterministic (see section What Goes in a Distribution of the manual), I find the most straightforward way to work out whether anything needs to go here is to build a distribution and spot what was missed out.

2.1.3 Libtool

We are using unmodified Libtool. See `libtool -version` for the actual version.

Libtool contains the darkest magic of the autotools, so it is fortunate that we need barely concern ourselves with it. This is because all of the technicalities of calling libtool are handled for us by the makefile generated by automake.

The function of libtool is to build libraries. While this is generally straightforward for static libraries, usually involving little more than working out whether `ranlib` is necessary, doing the same thing for dynamic libraries is extremely platform dependent. Libtool consists of a large body of platform-dependent code, ported to a broad range of operating systems, which implements a single platform-independent interface.

For more details, see the libtool manual at `http://www.gnu.org/software/libtool/manual.html`.

Unfortunately, libtool's library magic introduces a minor complication when you wish to run a program under a debugger : plain `gdb myprog` won't work, and you must instead use `libtool -mode=execute gdb myprog`. See section 'Debugging executables' of the libtool manual for discussion.

2.1.4 Autoreconf: why you don't need to know about aclocal

You don't have to know anything very much about `autoreconf`, other than that, if you change one of the autotools files `Makefile.am` or `configure.ac.`, you should probably run `autoreconf` to bring everything up to date. In fact, you probably don't even need that, since the generated makefiles have these dependencies explicit. The problem that `autoreconf` addresses is that when one of these files is updated, there are several commands which might need to be re-run, including `aclocal`, `autoheader`, `libtoolize` and others, and it's a headache trying to remember which ones are which.

2.1.5 Running ./configure

Running the configure script is basically very simple: `./configure`. There are, however, options and arguments which can help you, or catch you out. You can see the full list of options with the command `./configure -help`.

You set the place where the configured component will be installed with the `-prefix` option. This names a directory which will end up with the standard directories `bin`, `manifests` and so on. The default is reported by `./configure -help`, and is ultimately controlled by *starconf* (see Sec. 3.5 and Sec. 2.4; or Sec. 2.2.1 for a means of setting it to a per-directory default, which can be useful in certain circumstances).

Because configure's tests potentially take quite a long time to run, it is possible to cache the results between runs. If you add an option `-C`, then these results will be cached in a file `config.cache` in the current directory. When you do the tree-wide configure from the top level, it is important to give this

option, otherwise the configure step takes an unconscionably long time. When this is happening, the configure runs in subdirectories use the cache file in the top-level directory, and when you are running configure in subdirectories, you can use this global cache, too. If you are in a directory two levels down from the top (say), then you can configure slightly faster using the command

```
% ./configure --config-cache=../../config.cache
```

This reads and updates the given cache file. Sometimes, when things get a little confused, you might need to delete this cache file – this is always safe.

The Starlink autoconf adds a couple of extra standard options.

`-with-starlink[=location]` If no `location` is provided, this does nothing. This is effectively the default.

If a `location` is provided, it overrides the STARLINK environment variable which will be used. Very rarely needed.

Option `-without-starlink` causes the `STARLINK` shell variable to be unset. Some packages might with to configure themselves differently in this case.

`-without-stardocs` Controls whether documentation is built and installed. You might want to turn this off, since it can take quite a long time. The default is `-with-stardocs`, and so you can disable this with the configure option `-without-stardocs`.

A *few* components have extra options on the `./configure` command. For example, the component `docs/ssn/078` (the component which holds this document) has a `-with-pstoimg` option, the value of which must be the full path to a copy of the `pstoimg` program, to help in the cases where this cannot be discovered automatically (this may be a temporary feature of this component).

All of the components have "Some influential environment variables" listed in the help message. This will include at least `STARLINK`, and in the (very common!) case of components which include a compiler, an environment variable such as `CC` or `FC` which allows you to override the compiler which the configure script will find. This is useful if you want to avoid a default compiler and use a specific one instead. For example, if you wished to use the Sun C++ compiler specifically (while on a Sun, obviously), you would put `/opt/SUNWspro/bin` in your path, and set the `CXX` variable:

```
% export CXX=CC   # best avoided
% ./configure
```

or

```
% env CXX=CC ./configure   # best avoided
```

where the latter is probably preferable, inasmuch as it does not leave this important variable set, in such a way that it can make a difference unexpectedly.

Better than either of these is

```
% ./configure CXX=CC
```

This doesn't set `CXX` as an environment variable, but sets it in a similar-looking way as one of the `./configure` arguments. This way is preferable to either of the above for two reasons: firstly, it does not leave the variable set; secondly, this way `./configure` 'knows' that the compiler has been overridden, so that if you are using a configure cache, and you *fail* to do this when the directory is reconfigured, `./configure` can warn you of this, in case this is not deliberate.

If you change one of these variables between runs, and are using a configure cache, then the `./configure` script will warn you like this:

```
% ./configure -C
configure: loading cache config.cache
configure: error: 'STARLINK' has changed since the previous run:
configure:   former value:  /somewhere/else
configure:   current value: /export3/sun
configure: error: changes in the environment can compromise the build
configure: error: run 'make distclean' and/or 'rm config.cache' and start over
```

To deal with this, simply remove the `config.cache` file, check that the environment variable is indeed set as you wish, and rerun `./configure`.

You can pass options to the compiler using environment variables, but you will *not* need to do this in general, other than perhaps to set `CFLAGS=-g` to turn on debugging in the code you are building. The variables `CFLAGS` and `LDFLAGS` are variables you might potentially set and export in the environment, for example to point `./configure` to software installed in non-standard places (perhaps you are on a Mac and have installed Fink in `/sw` or OpenDarwin in `/opt/local`, or are on a Sun and have extra software in `/opt`). In this case you might set `CFLAGS=-I/opt/local/include` and `LDFLAGS=-L/opt/local/lib` in the environment to help configure and the compiler find libraries. Note that this is rather a blunt instrument, and because you cannot really control where the respective flags appear in generated compiler commands, you can end up picking up inconsistent versions of software. That is, this is a mechanism for getting yourself out of a fix, not a recommended way of building the software.

The important point of this is that these environment variables *do matter*, and implicitly *change the behaviour of `./configure`*. You should not have them set in your environment if you don't want this to happen.

The values you set here act as defaults in the Makefile, and can be overriden at make time by giving arguments to make:

```
% make 'CFLAGS=-g -I/somewhere/else'
```

2.2 Starconf, and the Starlink autoconf macros

The starconf application does some of the work of ensuring that your build directory is correctly organised. It does the following:

- ensures that you have a `./bootstrap` script;
- runs the *starconf-validate* application to check that your `configure.ac` and `Makefile.am` files look at least plausible by checking, amongst other things, for the presence of a `STAR_DEFAULTS` invocation in the former, and to check that the right files have been checked in to the repository;
- creates the script `./starconf.status`, which allows you to inspect some of the starconf parameters.

The `./bootstrap` script is 'owned' by starconf, and you should not change it, since it will be overritten by starconf if a newer release of starconf includes an updated bootstrap script. If you *do* have some pressing reason to change it, then remove the word 'original' from the second line of the bootstrap file, which signals to starconf that it should leave the file untouched. The standard bootstrap script:

- runs *starconf*, using `./starconf.status` if necessary to find the path to the program;
- bootstraps any subdirectories named in the `AC_CONFIG_SUBDIRS` macro within `configure.ac`;
- runs *autoreconf* to regenerate configuration files if necessary.

You need to run starconf explicitly only once, when you first prepare a directory for configuration. The `./bootstrap` file which this creates itself runs starconf, so that each time you run the bootstrap script, you run starconf also. This has no effect unless either the bootstrap script or the macro set has been updated in the starconf installation, in which case starconf will propagate the updates to your directory. The `./starconf.status` script should not be checked into the repository. The command `starconf-validate` (which is invoked by `starconf` in passing but which may be invoked explicitly also) will tell you what should and shouldn't be checked in.

You might on other occasions run the `./starconf.status` script. You will do this firstly to query important locations, such as the location of the starconf templates:

```
% ./starconf.status --show buildsupportdata
/export3/sun/buildsupport/share
```

See `./starconf.status -show -all` or `./starconf.status -help` for the list of all the things which you can show in this way (though be warned that this list is not yet completely stable, and may yet change).

These variables are fixed for a particular installation of starconf (you can in principle have more than one installation of starconf, and choose which one to invoke, but there is unlikely any need for that).

Two very important variables are `STARCONF_DEFAULT_PREFIX` and `STARCONF_DEFAULT_STARLINK`, discussed in Sec. 2.2.1 below.

A companion to the starconf application is the *starconf-validate* application. When run, this examines the current directory, checking that all required files are present, and checking that you have the correct files checked in to the repository. The command `starconf-validate -help` shows which files are in which category. Note that this applies only to directories which are fully Starlink applications – those in the `libraries` and `applications` directories; components in the `thirdparty` tree, on the other hand, have some starconf features such as a `component.xml` file, but are not valid according to *starconf-validate*; also 'bundle' directories such as `libraries/pcs`, which have no actual code in them, are not valid in this sense.

There are templates available for the most important starconf files. See Sec. 2.2.3.

The final component of the starconf system is the file `component.xml`. This is an XML file containing information about the particular component in the directory. The information in this file is redundant with some of the information you specify in `configure.ac`, and so the best way to ensure the two are consistent is to configure `component.xml` from `configure.ac`. To this end, there is a template `component.xml.in` file in the starconf 'buildsupportdata' directory. When you are preparing a directory to be build with the Starlink tools, copy this `template-component.xml.in` into the current directory under the name `component.xml.in`, and fill in those field which are not configured. Remember to uncomment the elements you fill in! See Sec. 2.2.4 for details. The files `component.xml` and `component.xml.in` should both be checked in.

2.2.1 `STARCONF_DEFAULT_PREFIX` and `STARCONF_DEFAULT_STARLINK`

Two very important variables are `STARCONF_DEFAULT_PREFIX` and `STARCONF_DEFAULT_STARLINK`. The first is the default installation prefix of the software you are building; the second is the location of a currently built Starlink tree, which will be used for applications and include files if they are not found under `STARCONF_DEFAULT_PREFIX`.

The value for each of these was frozen when starconf was itself built and installed, most typically at the time of the tree-wide bootstrap, and you can see the values for these with the command `starconf -show STARCONF_DEFAULT_PREFIX` for example. You can also see the results of this configuration in the `./configure` script itself, as the command `./configure -help` indicates within its output where material will be installed by default. It is not unreasonable to have more than one starconf installation,

depending on your path, if you wish to have different frozen-in defaults here. The value of each of these two variables is typically something like `/star`.

It is important to emphasise that these parameters are *frozen* when starconf is built, and their values are ignored when a component is itself bootstrapped. If you need to change either value, then there are two ways you can do this.

The first, much more common, way is to provide the `-prefix` option when you run `./configure` in a component (this overrides the frozen-in value of `STARCONF_DEFAULT_PREFIX`), or you can set the `STARLINK` variable as a `./configure` argument line (or less securely default it from the environment, see Sec. 2.1.5 for discussion), overriding the frozen-in value of `STARCONF_DEFAULT_STARLINK`.

Specifying `-prefix` or `STARLINK` each time you run `./configure` might be troublesome when you are working on a component's configuration script, especially as doing this inconsistently would produce very confusing results. You might want to do this if you are working on a repository branch, and so want built material from the current directory to be installed in one branch-specific place, while using a Starlink tree based on the trunk. You can default this on a per-directory basis by using a m4 back-door. Create a file `acinclude.m4` in the directory in question, and include in it a line like:

```
m4_define([OVERRIDE_PREFIX],[/my-branch/star])
```

(or `OVERRIDE_STARLINK` to override that variable). Then run `./bootstrap`, which invokes 'autoreconf' in turn, and the generated `./configure` file will have the named directory as its default prefix. This method is in principle fragile, and uses a partly-deprecated autoconf interface, and so is not guaranteed to work for all time. It is at present adequately robust in practice, however, and so is a respectable technique as long as you are aware that it is to some extent a rather special effect.

In general, however, we recommend that you do not adjust `-prefix`, and that you leave the `STARLINK` variable unset. Also, since they are ignored at all times except when the whole tree is being configures, it might be wise *not* to have the `STARCONF_DEFAULT_...` variables set, which could trick you into believing that they are (not) having some effect.

2.2.2 Components

Throughout this documentation, the term 'component' is used rather than 'package'. The distinction is that the components are the units of organisation within the CVS tree, and the packages are the units which are distributed as tarballs. These will generally overlap, but the mapping may not be one-to-one, so that the components within the CVS tree might not be reflected in ultimately distributed packages.

A 'component directory' is a directory which has a `component.xml.in` file in it. All component directories will have a manifest file created and installed in `.../manifests`; non-component directories will not have manifest files. Everything that's installed must be installed as part of some component or other. This helps when you want to remove a component (a *very* primitive de-installer is therefore `rm -f 'sed '1,/<;files>;/d;/<;files>;/,$d' .../manifests/XXX'`), and packaging for distribution should be easy.

The starconf `./bootstrap` scripts, which are installed by the starconf application, recurse into the directories listed in a `configure.ac` file's `AC_CONFIG_SRCDIR` macro, and will stop recursing when they find a `component.xml.in` file. They'll warn if they find a `component.xml.in` file in any `AC_CONFIG_SUBDIRS` directory, but ignore it, and exit with an error if they do not find a `component.xml.in` file and there are no `AC_CONFIG_SUBDIRS` directories in which to search further. That is, the tree of directories which the top-level bootstrap traverses should have `component.xml.in` files at or above all its leaves.

This further implies that a component 'owns' all the tree beneath the component directory, and thus that you cannot have one component located within another.

This means that bootstrap files only have to appear within component directories, but not their children, and starconf/autoreconf and starconf-validate only have to be run in component directories. Macros

like `STAR_DEFAULTS` are still usable in the subdirectories' component.ac files (because auto(re)conf in the parent handles those macros when it remakes the child configures).

Automake spots `component.xml.in` files and will only arrange to install a component manifest if it does fine a `component.xml.in` file. It also demands that the `STAR_SPECIAL_INSTALL_COMMAND` macro (see Sec. A.29) appear only in configure.ac files in a component directory (that macro is a sufficiently magic blunt instrument that it shouldn't be allowed to lurk where it can't have an eye kept on it).

The `starconf-validate` script checks these various assertions. As usual, `starconf-validate` should be regarded as pickily authoritative about how things are ideally configured, and if it objects to something, that means either that the the directory in question is non-ideally configured (and should be fixed), or the part of SSN/78 that suggested you set things up that way is out-of-date or unclear and should be clarified.

2.2.3 File templates

There are three templates available as part of the starconf system. They are located in the starconf 'buildsupportdir': given that the starconf application is in your path, you can find this location with `starconf -show buildsupportdata`, and if you have a `starconf.status` script in your local directory, you can find this directory with `./starconf.status -show buildsupportdata`.

The three templates present are `template-Makefile.am`, `template-configure.ac` and `template-component.xml.in`.

2.2.4 The format of the component.xml file

The meaning of the elements in the component.xml DTD is as follows.

component: attribute 'id' The ID for this component. This must match the package name in `configure.ac`, and is the name under which the manifest file is installed. Let this remain under the control of `configure.ac`

component: attribute 'status' The status of this component, which can be `current` (the default, and assumed if no such attribute is present), or `obsolete`. When the `Makefile.dependencies` file is regenerated, the program doing that will warn if any component depends on a component marked obsolete.

version The version number of this component. This must match the version number in `configure.ac`.

path This is the path to this package within the CVS repository, without any leading slash. Make sure this is correct, or else the top-level build will probably fail.

description A brief description of the component.

abstract A fuller description of the component.

dependencies This is the set of component IDs which this component depends on. This is managed by the `STAR_DECLARE_DEPENDENCIES` macros, and should not be adjusted by hand.

developers This is a non-exhaustive list of those who have worked on this package. Little use is made of this at present, but its use is likely to extend, for example to be the source of recipients for CVS commit messages. The fields within it are `<name>`, which is the real name of the individual; `<uname>`, which is the username of the developer on the CVS system; and `<email>`, which is the person's email address. The last two elements are not required.

documentation A list of SUNs and the like. At this stage the format of this element is not completely finalised, and it is best to leave its maintenance to the `STAR_LATEX_DOCUMENTATION` macro.

bugreports The email address for bug reports.

notes Any other notes that might be of interest or utility.

Note that, since this is an XML file, you will have to escape the ampersand, left- and right-angle-bracket characters as `&`, `<` and `>` respectively (well, you don't actually *have* to escape the right angle bracket character, but it looks marginally neater if you do). There are no other entities defined in this context.

The DTD for this file is included in the repository, at `buildsupport/starconf/componentinfo.dtd`, with a RelaxNG version at `buildsupport/starconf/componentinfo.rnc`.

2.3 Version numbers

The only place where you should specify a version number is as the second argument to the `AC_INIT` macro; if you need the version number anywhere else, such as a bit of code which reports the package version number, you should put it there with the `@PACKAGE_VERSION@` substitution variable, or one of the variants of this defined by `STAR_DEFAULTS` (see Sec. A.18).

Starlink follows the usual practice of indicating release versions with a three-part version number, major.minor-release. We have no particularly precise rules controlling these, but the major version number is typically incremented for a release which contains a major change in functionality or implementation, the minor is incremented for changes of functionality less significant than this, and the release number is incremented for bugfix releases, involving no functionality changes at all.

If a component contains a shared library, then the maintainer may elect to maintain shared library versioning information [XXX: should we make this a requirement for libraries?]. Since we use libtool, we can use its `-version-info` interface to maintain library versions. This is documented in section 'Updating library version information' of the libtool manual, but the explanation is repeated and elaborated below.

The crucial thing to remember is that the versioning information encoded in this `-version-info` option is *not* the component version number, but instead an encoding of which *interface* versions a given shared library implements, in the form of a triple `current:revision:age`. 'Current' is the interface version, as a monotonically increasing integer, and 'age' is a statement of how many interfaces are compatible with it: if a library has a triple `c:r:a`, then it implements all the interfaces between versions *c-a* and *c*, inclusive. When you're making a new release of a library which had the `-version-info c:r:a`,

(1) If the interface is unchanged, but the implementation has changed or been fixed, then increment `r`.

(2) Otherwise, increment `c` and zero `r`.

   (a) If the interface has grown (that is, the new library is compatible with old code), increment `a`.

   (b) If the interface has changed in an incompatible way (that is, functions have changed or been removed), then zero `a`.

This is illustrated in Sec. 2.3.

The libtool documentation suggests starting the version-info specifications at 0:0:0, and this is the default if no `-version-info` option is present. Since only some Starlink libraries have this versioning maintained, it is best if you start the specification at 1:0:0, to distinguish this from those libraries which have no version-info defined.

You add the `-version-info` specification to the libtool command using the library's `LDFLAGS` variable, as follows. First the `Makefile.am` (from the `prm` component):

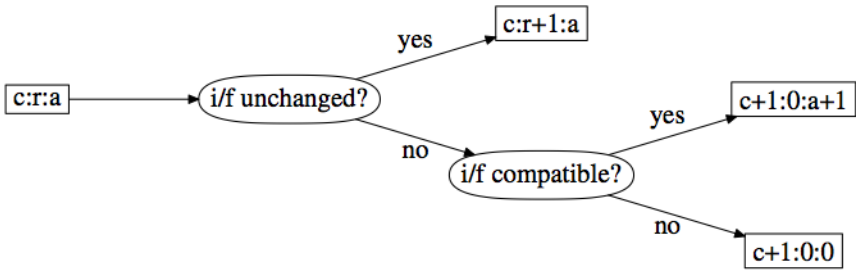

Figure 2.5: Updating `-version-info` specifications

```
libprm_la_SOURCES = \
    $(F_ROUTINES)
    $(PUBLIC_INCLUDES) \
    $(PRIVATE_INCLUDES) \
    PRM_PAR
libprm_la_LDFLAGS = -version-info $(libprm_la_version_info)

libprm_a_la_SOURCES = \
    $(PLATFORM_C)
libprm_a_la_LDFLAGS = -version-info $(libprm_a_la_version_info)
[...]
```

and then the corresponding section from `configure.ac`:

```
AC_INIT(prm, 1.3-1, starlink@jiscmail.ac.uk)
# Version-info specifications.  See SSN/78 for guidelines, and update the table
# below for ANY change of version number.
#
# Release    libprm.la      libprm_a.la
#  1.3-1       1:0:0          1:0:0
AC_SUBST(libprm_la_version_info,   1:0:0)
AC_SUBST(libprm_a_la_version_info, 1:0:0)

[...]
```

There is no automake magic to the `libprm_la_version_info` variable name – it is just mnemonic, and you are free to change it if you have a good reason to do so. Since there is no fixed relationship between the component version number and the `-version-info` specification, it is important to maintain a simple table of associations between the two, and the `configure.ac` file is a sensible place to do that.

2.4 A few remarks on state

There is quite a lot of state held within the starconf build tree. This section contains a few notes on where this state is, and how to handle it.

Since most of the details here are rather intricate, you might save time by looking at the list of FAQs on state in Sec. 5.3.

Autotool state: The most obvious state within the tree is the existence of the `configure` and `Makefile.in` files which are the generated by autoconf and automake. Files `config.h.in` and `aclocal.m4`, are also part of this process, inasmuch as they are generated by autoreconf: `config.h.in` is based on the declarations within `configure.ac`; `aclocal.m4` is a local cache of m4 macros, drawn from the

starconf, autoconf, automake and libtool installations. Other objects generated by the autotools are the cache directory `autom4te.cache` and the tool links `config.guess`, `config.sub`, `depcomp`, `install-sh`, `ltmain.sh`, `missing` and `stamp-h1`. All of this can be happily blown away by hand, and a lot of it is removed by `make maintainer-clean`, though this doesn't remove things like `config.h.in` which are required for `./configure` to run. Most of this state is maintained reliably by the dependencies within `Makefile.in` – for example if you update `Makefile.am`, then a simple `make` will remake `Makefile.in` and regenerate `Makefile` before using the new `Makefile` to do the build. If you have removed this state or suspect that it may be out of date, then autoreconf will always regenerate it – it's always safe to rerun autoreconf.

Configuration state: When `./configure` runs, its most obvious effect is to transform the `.in` files into their corresponding configured results. However at the same time, it creates a log file in `config.log` (which can be handy to examine if a configure script fails or appears to give wrong answers), a cache in `config.cache` if you ran `./configure` with the `-C` option, it creates the directory-specific `./libtool` script, and it creates a file `config.status`. This last file holds most of `./configure`'s state, and can be used to either regenerate a particular file (`./config.status Makefile`) or else to update itself (`./config.status -recheck` has the effect of rerunning `./configure` with the same options such as `-prefix` which were given to the `./configure` script last time). You won't often have to run `./config.status` yourself, but it's reassuring to know what it's doing when the makefile runs it for you. It is always regenerated afresh by running `./configure`.

Build state: The compiled files are obviously part of the build state, and can be removed by `make clean` and regenerated with plain `make`. Less obviously part of this state are the directory `.libs`, which is libtool's' workspace and holds different types of objects and libraries, and `.deps`, which holds dependency information gleaned as part of the last build. Less obviously still are those (few) `.in` files which are configured as part of the build. As mentioned in Sec. 2.1.2, so-called 'installation directory variables' should be substituted at make time rather than build time, with a hand-written makefile rule in `Makefile.am`.

Starconf state: There is essentially no starconf state, since the starconf system's only role is to manage the `bootstrap` file and provide the extra autoconf macros (installed in the buildsupport part of the tree when starconf itself is installed). The `starconf.status` file is purely a cache of starconf variables, and allows you to locate the starconf installation which was used most recently. At one time, the `starconf.status` file did hold state, and allowed you to manipulate it, and was even checked in; this is no longer the case.

Build tree and installation state: The last noteworthy state is that in the build tree as a whole. The top-level makefile requires some state in order to manage its bootstrapping 'make world' build. This build is organised by representing the build and link dependencies between components by makefile dependencies between the components' installed manifests, which are installed in the `.../manifests` directory alongside, and at the same time, as the component is installed by the `install` Makefile target. Thus this is state held *outside of the build tree*. If the 'make world' build sees that a manifest file is up-to-date with respect to the dependencies expressed in `Makefile.dependencies` , it makes *no attempt to build it*, even if the component itself has been updated and itself needs rebuilding and reinstalling. A slight wrinkle here is that the 'buildsupport' tools – namely starconf and the autotools – are not explicitly listed as being a dependency of anything, since they are in fact a dependency of everything. Since they are rather a special case, these are built within the top-level bootstrap script, and the best way to run that 'by hand' is via the command `./bootstrap -buildsupport`, noting that the remarks about dependencies above apply to this process also. Thus, if you update a component – including one of the buildsupport components – and wish the top-level 'make world' build to notice this, the best way to do this is to first delete that component's manifest file from the installation tree. This process might change slightly with starconf developments (see Sec. C).

CVS state: This isn't really part of the build system as such, but this seems a good place to point out, or reassure you, that all the state for a CVS checkout directory is in the files in the CVS subdirectory, and that all the repository state for the directory is in the files in the corresponding directory within the repository.

2.5 Preprocessable Fortran

The Starlink build system includes support for preprocessable Fortran, in both autoconf and automake.

As described in Sec. B, the installed version of autoconf is extended with support for preprocessable Fortran, by adding the two macros `AC_PROG_FC` and `AC_PROG_FPP`.

You should use `AC_PROG_FC` in preference to the macro `AC_PROG_F77` described in the autoconf manual, since the 'FC' support in autoconf is more flexible and more developed than the 'F77' support, and the older macro is likely to be deprecated in coming autoconf releases. However a potential problem with the `AC_PROG_FC` macro is that it searches for Fortran 9x compilers before Fortran 77 ones. Fortran 9x incorporates all of *strict* Fortran 77 as a subset, but no more, so if you have used any of the common Fortran extensions (which is sometimes unavoidable), you might find the Fortran compiler objecting. In this case, you should use the second optional argument to `AC_PROG_FC` to specify Fortran 77 as the required dialect:

```
AC_PROG_FC([], 77)
```

See Sec. A.12 for full details.

Unlike C, there is no preprocessor defined as part of the Fortran standard, and this is inconvenient if you wish to use a Fortran extension if it is available, but not have the compilation fail if it is absent. This most commonly applies to the `READONLY` keyword on the Fortran `OPEN` statement. It is possible to use the C preprocessor, *cpp*, with Fortran, though not completely reliably, since the underlying syntaxes of C and Fortran are so different that *cpp* is capable of emitting invalid Fortran in some circumstances.

Both the Sun and GNU Fortran compilers can handle cpp-style preprocessor constructs in a Fortran file, avoiding any separate preprocessing stage. Sun have produced a Fortran preprocessor, *fpp*, which is available at `http://www.netlib.org/fortran/`: it's freely available, but not open-source. And more often than not we can in fact fall back on simple *cpp*, as long as we have confined ourselves to the `#include`, `#define` and `#if...` constructs. If it comes to that, such simple preprocessing could be mostly handled with a simple Perl script.

Starlink autoconf and automake between them add support for preprocessable Fortran. Autoconf supplies the `AC_PROG_FPP` macro described in Sec. A.13, to investigate what compiler and preprocessor are available, and automake works with the results of that test to add appropriate support to the generated `Makefile.in`.

For example, you might have in your `configure.ac` file the lines

```
AC_PROG_FC
AC_PROG_FPP
AC_FC_OPEN_SPECIFIERS(readonly,[access='sequential',recl=1])
```

and have in your `my_io.F` the lines

```
#include <config.h>
      SUBROUTINE MYIO(...)

* blah...

      OPEN ( UNIT = LUCON, FILE = IFCNAM, STATUS = 'OLD',
#ifdef HAVE_FC_OPEN_READONLY
     :       READONLY,
#endif
#ifdef HAVE_FC_OPEN_ACCESSSEQUENTIALRECL1
     :       ACCESS='SEQUENTIAL',RECL=1,
#endif
     :       FORM = 'UNFORMATTED', IOSTAT = ISTAT )
```

The file `my_io.F` is listed in the `Makefile.am` as one of the `_SOURCES` of whatever library or application it contributes to, alongside the other Fortran files. If the compiler discovered by the `./configure` script can support preprocessor statements itself, then the `.F` file is treated like any other; if not, and a separate preprocessing stage is required, then the Makefile handles this itself.

Note that we are using `.F` here as the file extension for preprocessable Fortran: see Sec. A.13 for discussion.

In those cases where you wish the preprocessing step alone, such as when you are generating an include file from a template, you should name the input file with a `.F` extension also. You will need to include a preprocessing rule in the makefile. The following idiom might be helpful:

```
# Run a .F file through the fpp preprocessor, to produce a file with
# no extension.
# The following deals with case-insensitive filesystems, on which
# foo.f and foo.F would be the same file.  FPP_OUTPUT is
# either "" (in which case the preprocessor writes to foo.f, and
# the filesystem is presumably case-sensitive) or ">$@".
foo: foo.F
        rm -f foo
        $(FPP) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) \
           $(FPPFLAGS) $(CPPFLAGS) foo.F $(FPP_OUTPUT)
        test -f foo || mv foo.f foo
```

or the following, for a suffix rule

```
SUFFIXES = .F
.F:
        rm -f $@
        $(FPP) $(DEFAULT_INCLUDES) $(FPPFLAGS) $(CPPFLAGS) $< $(FPP_OUTPUT)
        test -f $@ || mv $(<:.F=.f) $@
```

In this second example, in which we have elided the useful comment, we have declared the `.F` suffix (we wouldn't have to do this if there were 'real' `.F` sources in this directory), and are using the `$<` and `$@` magic variables.

In second example, we elided most of the flag variables we included in the first one, since we probably don't need them. In case of doubt, copy the `.F.f:` rule in one of the generated `Makefile.in` files.

There is a more elaborate example of this in the `prm` component.

2.6 Using the Starlink CVS repository

The Starlink source code is held in a CVS repository, which has public readonly access.

If you have an account on the repository machine, you have read and write access to the full source set. When you are making a fresh checkout, or giving other commands not within the context of a local check-out, you should refer to it as

```
% cvs -d :ext:\emph{username}@cvs.starlink.ac.uk:/cvs <cvs-command>
```

where *username* is your username on that system; you will need to set the environment variable `CVS_RSH` to `ssh` to connect to the repository.

There is also anonymous CVS access to the repository. Use

```
% cvs -d :pserver:anonymous@cvs.starlink.ac.uk:/cvs login
# password starlink
% cvs -d :pserver:anonymous@cvs.starlink.ac.uk:/cvs <cvs-command>
```

There is also anonymous web-based access to the repository at `http://cvsweb.starlink.ac.uk`. See `http://dev.starlink.ac.uk/` for news.

There are a few more details about CVS, including a link to a CVS Primer, in Sec. 2.7.

Part of the point of the Starlink CVS repository is to give users ready access to the fully up-to-date sources so that more sophisticated users can build the most recent application versions for themselves and even, if they find a bug, offer fixes. Anyone who finds a bug is invited to report it through the Project's bug-reporting system at `http://dev.starlink.ac.uk/bugzilla/`, but if this report comes with fixes, it will be *particularly* welcome.

If you do have bugfixes to offer, then you should get in touch with the 'owner' of the component to discuss how to give them to the project. You will find the owner of a component by looking at the file `component.xml` in the component's checkout directory; this should list those who have worked on the component, along with someone nominated as the component's 'owner'.

If you make quite a few fixes, then it might be best to give you committer access to the repository, by giving you an account on the repository machine. Talk to someone from the project about setting that up.

2.7 CVS

There is a compact CVS Primer on the web at `http://purl.org/nxg/note/cvsprimer`, and that includes pointers to fuller documentation. This section includes a few tips on using CVS which go beyond that document, or which are more specific to the Starlink CVS repository.

2.7.1 CVS and tagging

Tagging is very important, as it is through tagging that you create branches, and mark certain sets of files as a set, so that they can be recovered as a set in future. The current tagging policy is at `http://wiki.starlink.ac.uk/twiki/bin/view/Starlink/CvsTagging`.

You make a tag with the `cvs tag` command, while you are in a directory with a part of the repository checked out:

```
cvs tag <tag-name>
```

This applies the tag to the repository versions indicated by the files in the current directory. In the most common case, you have just finished preparing the set of files for a release, so all the files in the directory are as you want them to be, and committed. There's a slight trap here: if there are any files which are *not* committed, then it is the repository version which corresponds to the modified file which is tagged, not the modified file itself (this is never useful; it is simply a warning to use the `tag` command only in a fully-committed directory). If you tag a set of files which are on a branch, then it is (probably unsurprisingly) the branched files which are tagged.

There is also a `cvs rtag` command which is similar, but operates only on the repository. You won't need to use `rtag`; don't confuse the two.

2.7.2 CVS and recursion – checking out only a subtree

There are two traps in the way that CVS recurses into subdirectories. The first is that if a subdirectory is present in the repository but not in your checkout (most commonly because it has been recently added to the repository by someone else), then CVS will not by default add that subdirectory when you do an update, and will give no indication that it is there. It is not clear to me why this is a sensible default, but it is the case nonetheless. Only if the `-d` option is present on the `cvs update` command will the 'missing' directory appear.

The other 'trap' is that CVS does recurse by default when you do a checkout. This is almost always the right thing, but it can be inconvenient when you want just the top level of the repository. If you wanted to check out only the top level, or only the buildsupport tools, then the command `cvs -d ??? checkout .` would not be the right thing, since it would check out the top level and everything underneath it. A better set of commands would be

```
cvs -d username@cvs.starlink.ac.uk:/cvs co -l .
cvs -d username@cvs.starlink.ac.uk:/cvs co buildsupport
cvs -d username@cvs.starlink.ac.uk:/cvs co thirdparty/fsf
```

The first line checks out the top level directory but, because of the `-l` ('local') flag, does *not* recurse and check out the complete repository. The following lines check out particular directories, usefully recursing into their children.

If you want to check out just the `applications` directory, but none of its children, use

```
cvs -d username@cvs.starlink.ac.uk:/cvs co -l applications
```

while in the top-level directory. Don't do `... co -l .` in the `applications` directory – you'll get the top-level directory again.

Chapter 3
Building applications and libraries

This section applies *only* to building code which has been checked out of the CVS repository. It does *not* apply to building a distribution tarball: that process is nothing more than the usual:

```
% ./configure
% make
% make install
```

with the usual options for the `./configure` script, of which the most important is `-prefix` to control the installation location of the code. The default location is typically `/star`, though a particular distribution might have a different value (see `./configure -help` for more details).

Recall that the `make install`, above, installs a manifest listing the files actually installed, in the directory `/star/manifests` (depending on the value of `-prefix`, which defaults to the value shown by `./starconf.status -show STARCONF_DEFAULT_PREFIX`).

Most users of this document will likely be concerned only with the details of building a particular component, and thus concerned primarily with Sec. 3.3.

Only rather few users are concerned with building the entire tree, for example for the nightly build: this is dealt with in Sec. 3.2.

Note: In all cases, you should *not* have the `INSTALL` environment variable defined. In old-style Starlink makefiles this indicated the install location; in the new-style build system, this location is indicated by the 'prefix' described above abd elsewhere, and the generated Makefiles take the `INSTALL` variable to be the name of a program to use to perform the actual installation.

3.1 The top-level Makefile

The important targets in the top-level makefile are:

`make world` This target builds the whole tree, in an order which respects the dependencies between components. It builds all the targets listed in the variable `ALL_TARGETS` within the top-level `Makefile.in`. Those targets are install manifests, in the default manifest directory (`/local/star/manifests` in the example below), so that `make world` both builds and installs the entire software set.

The bootstrap that this is part of is described in more detail in the next section.

`make <manifest-file>` As an alternative to `make world`, you may specify a single manifest file. This builds and installs the corresponding component, and all of its dependencies. Note that this does nothing if the given manifest file is up-to-date with respect to its dependencies on other manifest files. If you have just updated a component, then you can rebuild and reinstall it by deleting the component's manifest file and remaking it.

`make` **or** `make all` This recurses into each of the directories listed in `AC_CONFIG_SUBDIRS` within the top-level `configure.ac`, and invokes `make all` there. This target brings the tree up-to-date after a CVS update, but it does so without respecting the dependencies between components. This target is present for the sake of consistency, and as a convenience for bringing the local tree up-to-date after a CVS update, and you should not use it as a shorthand or alternative for `make world` above.

`make clean` The targets `clean`, `distclean` and `maintainer-clean` simply recurse into the children and invoke the corresponding target there.

`make install` This targets does nothing, and is also present largely for the sake of consistency. Depending on the context, you should use `make world` or, if you have just updated and rebuild a particular component, then either delete and remake its manifest file (as mentioned above), or go to the component's directory and use `make install`.

Note that the building above presumes that all the required code is checked out of the repository – it does not handle the checkouts for you, and if you do not have a required component checked out, it will simply fail.

3.2 Bootstrapping and building the entire tree

If you wish to build the entire tree after a fresh checkout, you do so at the top level, using the `./bootstrap` script and `Makefile.in` located there. Neither of these is a 'standard' one, as installed by *starconf* or Starlink automake.

These instructions are echoed in the file `README` in the top-level directory.

The complete procedure for a full build is as shown below, for an installation into the notional directory `/local/star`; explanations follow.

```
1% cvs -d :ext:username@cvs.starlink.ac.uk:/cvs checkout .
2% unset STARLINK INSTALL
3% export STARCONF_DEFAULT_STARLINK=/local/star
4% export STARCONF_DEFAULT_PREFIX=/local/star
5% rm -Rf /local/star
6% PATH=/local/star/bin:/local/star/buildsupport/bin:...??? # see note
7% ./bootstrap
... a few harmless warnings, and miscellaneous other chatter
... takes quite a while
8% make configure-deps
... builds some programs needed before configure
9% ./configure -C
... mucho blah-blah-blah
10% make world
... etc
11% ls /local/star/manifests
adam      automake  ems     hlp      messgen  par       sae   star2html  task
ams       chr       fio     lex      messys   parsecon  sla   starconf
atimer    cnf       hds     libtool  misc     pcs       sock  string
autoconf  dtask     hdspar  mers     msp      psx       sst   subpar
```

Line 1: check out the entire collection from the repository. See Sec. 2.7.2 for hints on how to be more discriminating, which may be desirable, since not everything in the repository is controlled by this bootstrapping build, most notably the contents of the `java` directory.

Lines 2-4: unset the `STARLINK` and `INSTALL` variables, and set the two starconf variables as shown. For a whole-tree build, these two should have the same value, to ensure that the build uses only tools and libraries which have been built at an earlier point in the build. For fuller details of these variables, and more information about controlling the bootstrapping process, see Sec. 3.5. This example shows sh-style commands for manipulating the environment; you will of course use the csh-style `setenv` and `unsetenv` if you're using that shell.

Also at this point you should review any other environment variables that might affect the build. For example, the variables `F77`, `FC` and `CC` will force the build system to use the specified compilers. In particular, the default IRAF installation sets the `F77` variable to be a script which works happily in most cases, but which will *not* work with libtool, and will cause the build to fail with opaque error messages about being 'unable to infer tagged configuration'. Unless you do wish to force certain compilers to be used, it is probably better to unset these variables before building the software. Try `./configure -help` for a list of 'some influential environment variables', and Sec. 2.1.5 for more discussion of this issue.

Line 5: the directory where the components are to be installed can be empty, and need not exist at the start. It doesn't matter if this is not empty in fact, since most of the installation tree is not examined by the build. The only exception is `/local/star/manifests`, since the files in this directory control what is actually built – see the notes for line 9.

Line 6: put `/local/star/bin` and `/local/star/buildsupport/bin` in the path. Make sure that there are *no* Starlink binary directories elsewhere in the path: running `which messgen` (for example) at this point should find nothing.

Line 7: run the `./bootstrap` script. This configures, builds and installs the 'buildsupport' tools – the autotools plus the starconf application – then recursively runs the bootstrap code in a selection of the directories beneath this one. The directories the bootstrap recurses into, in this directory and its children, are those listed in a `AC_CONFIG_SUBDIRS` macro in `configure.ac`. After this, you should be able to do

```
% which autoconf
/local/star/buildsupport/bin/autoconf
% autoconf --version
autoconf (GNU Autoconf) 2.59
...
%
```

to verify that the correct tools have indeed been installed in the correct place.

Line 8: configure the entire tree (the `-C` option indicates that configure should cache its results, speeding up the process). This configures this directory and recurses into the children named in `AC_CONFIG_SUBDIRS` in this `configure.ac` and those below it.

If you will need to give `./configure` some help to find include files and libraries, you *might* try setting the `CFLAGS` or `LDFLAGS` variables in the environment, in the manner described in Sec. 2.1.5. Note, however, that this is not a recommended practice.

Line 9: build the entire tree, using the `make world` target described in Sec. 3.1. The make builds all the components in an order which ensures that each component is build after the components it depends on, as declared in `STAR_DECLARE_DEPENDENCIES` invocations in those components' `configure.ac` files. The set of dependencies can be examined (if you're interested) in the file `componentset.xml`, which is built up from the various directories' `component.xml` files, as reprocessed into `Makefile.dependencies` .

Note that the dependencies are expressed as dependencies of components' *manifest* files on each other, and the top-level makefile is written so that each component is built with the pair of commands `make; make install`, which builds the component then installs both it and its manifest. Thus, if you need to ask for a particular component, foo, to be built, you can do so from the top-level directory with `make /local/star/manifests/foo`.

Line 10: after the make, a manifest is installed for each of the built components.

3.3 Building a single component

In Sec. 3.2 we saw how you would bootstrap and build the entire tree, building each component's dependencies before building the component itself.

If, on the other hand, you wish to build only one component, because you wish to install the latest and greatest version of some application, or because you want to work on the CVS version, then you can do that without, in principle, building the entire code collection.

The component directories are organised so that you can build a component from a CVS checkout by moving to the component's directory and giving the sequence of commands:

```
% ./bootstrap
% ./configure
% make
% make install
```

Before you can do this, however, you will have to do a little preparation, and there are two cases.

One of the functions of the `./bootstrap` script is to regenerate the `./configure` script if necessary, since it is not checked in to the repository. This requires that the Starlink-specific autotools are in your path, and so you must first install these 'buildsupport' tools if they are not installed already. Instructions for that are below

There are two cases next. The first is where you currently have a built and installed Starlink system which you either built yourself earlier, or which you installed from a Starlink distribution (note: a Starlink CD distribution dating before 2005 doesn't count here – though the applications and libraries are essentially the same, the distribution and building arrangements changed radically). The second case is where you are only interested in one component, but are starting from scratch, without a pre-existing distribution (of course, once you successfully complete the second case, you are in the situation where the first case applies).

Case 1 – pre-built Starlink system: This is the easy one. You need check out only the component of interest (see Sec. 2.7.2 for notes on that).

- Change to the component's directory.
- Define the environment variable `STARCONF_DEFAULT_STARLINK` to point to the top of the existing tree, which might be `/star` or its equivalent.
- Define `STARCONF_DEFAULT_PREFIX` to point to the same place if you want to install your new versions there, or to a different location if you want to install them separately.

At this point you can run `./bootstrap` as above, and have these locations baked in to the configuration as the defaults.

Alternatively, you can control these two locations at configure time, *after* you have run `./bootstrap`. You control the installation location with the `-prefix` option and the location of the Starlink tree either with the `STARLINK` environment variable or, better, with the `-with-starlink` option. Although the use of the `STARLINK` variable is supported, it involves less (easily forgettable) defaulting if you use the `-with-starlink` option instead. See Sec. 2.1.5 for details. Having said this, setting the `STARCONF_DEFAULT...` variables is probably preferable, since you don't then have to remember to supply the options if you run `./configure` again.

Case 2 – building from scratch: This isn't actually much more complicated, but does take a little longer. You need to check out all of the components that your component depends on; since there is (currently) no automatic way to discover what this set of components is, in practice you might as well check out the whole repository. Since you don't have a pre-built Starlink directory available, then you will have to build all or most of the tree, before the component you're interested in can be built. In this case, follow the instructions in Sec. 3.2 – which include setting the `STARCONF_DEFAULT_...` variables to suitable values – but instead of finishing with `make world` in step 9, do `make /mystar/manifests/cpt`, where `/mystar` is the location you have chosen for the installation tree (as specified in `STARCONF_DEFAULT_STARLINK`), and `cpt` is the component you are interested in. This will build and install in turn each of your component's dependencies, and then the component itself.

If you are doing this because you want the latest and greatest version of a particular component, then you are finished.

If you are doing this because you want to do development work on the component, then you might well want to keep the functioning version separate from your development version. In this case, you need to go through this procedure once with STARCONF_DEFAULT_STARLINK and ..._PREFIX having the same values, as above. Next, you change to the component's directory, and either bootstrap your development component a second time, with STARCONF_DEFAULT_PREFIX set to a separate installation location (you are now in the situation where case 1 applies, above, and you can give the comments `./bootstrap; ./configure; make`); or else re-run `./configure` with its `-prefix` option set to point to that alternative location (the first mechanism simply sets the default for the second, and is probably preferable for the reasons mentioned above).

Note that part of the function of the `./configure` script is to find the location of important files and freeze them in to the Makefile. Thus the STARLINK variable is ignored after configure-time, so that you cannot, in general, change the STARLINK variable or change your path, and have new binaries picked up.

3.4 Building monoliths

As illustrated in Sec. 2.1.2, you can build monoliths in an analogous way to the way you build programs, by declaring the monoliths as the value of the bin_MONOLITHS variable.

Along with the bin_MONOLITHS variable, a Makefile must also declare a set of tasks using the TASKS primary, and this set will be empty if the monolith is an ATASK. If a monolith foo has a set of tasks a and b, then this declaration will look like this:

```
bin_MONOLITHS = foo
foo_TASKS = a b
```

That is, the tasks are listed without any .ifl extension.

The foo_TASKS primary indicates to automake that there will be .ifl files in the local directory corresponding to each of the tasks in the monolith, that these .ifl files should be distributed, and that the appropriate extra links should be made, in the binary directory, when the monolith is installed. The tasks variable might also be used in the foo_SOURCES variable to declare the dependence of the monolith on the Fortran files corresponding to the tasks (it would be possible to automake to have inferred this last step, but it seems clearer overall to have this dependency at least made explicit). Automake permits the usual Makefile variable rewritings, so that if you have a Fortran file corresponding to each task, then this set is `$(foo_TASKS:=.f)`which might be useful.

The generated Makefile.in will install each of the task .ifc files, plus the monolith .ifc file. The monolith .ifl file is constructed by appending each of the task .ifl files, wrapped in `begin monolith` and `end monolith`, and this generated .ifl file should not be included in the repository or the distribution.

If you are building an ATASK, then the task is the same name as the program. Indicate this by *omitting* the ..._TASKS variable:

```
bin_MONOLITHS = foo
# no variable foo_TASKS, since foo is an ATASK
```

(the extra comment is useful to help document the slightly non-obvious construction here). In this case, the only .ifl file is that for the ATASK. The makefile makes sure that this .ifl file is distributed, so the file should either exist, or have a rule in the Makefile.am.

The MONOLITH primary allows the prefixes bin, check and noinst, rather like PROGRAMS.

In order to use the monolith support, you must request this by including the declaration STAR_MONOLITHS in the configure.ac file, most rationally near the AC_PROG_CC-style declarations.

3.5 Bootstrapping without building: configuring starconf and the autotools

If you intend to work only on a single component, then you can configure and build it as described in the previous section. As described there, the bootstrapping procedure requires that the 'buildsupport' tools – *starconf* plus Starlink-specific autotools – be installed and in your path.

You *must* install the autotools which are part of the CVS repository, since these have been extended and customised specially for the Starlink build tree (for the details of the differences, see Sec. B). The standard autotools will not work.

The way to do this is to make sure you have the required components checked out (see Sec. 2.7.2), then go to the top of the tree and give the command

```
% ./bootstrap --buildsupport
```

This builds and installs the 'buildsupport' tools – namely the autotools plus starconf – but does not go on to bootstrap the rest of the tree. The functions of the variables `STARCONF_DEFAULT_PREFIX` and `STARCONF_DEFAULT_STARLINK` were described in Sec. 2.2: in order to configure starconf at this stage, you must define these two variables *as environment variables* before running the `./bootstrap` script. This will cause the tools to be installed in the directory `$STARCONF_DEFAULT_PREFIX/buildsupport/bin`, and the configured starconf to configure components, when it is itself invoked, so that the default value of `STARCONF_DEFAULT_PREFIX` is the value given here, as described in the previous section.

You must use this method of configuring starconf and the three autotools – autoconf, automake and libtool.

After that, you should put the `.../buildsupport/bin` directory in your path, and start work.

See also Sec. 2.2.1 for how to make a version of the buildsupport tools which has a default installation prefix different from the buildsupport installation location.

Chapter 4
Incorporating a package into the Starlink build system

The process of adding a package to the build system, and autoconfing it, is reasonably mechanical. The main differences from the traditional build system are as follows.

There is now no `./mk` file, and so no platform configuration using the `$SYSTEM` environment variable. Instead, all platform dependencies should be discovered by the configuration process. It is only in rather extreme cases that you will need to resort to platform-specific code, and that should be handled by the starconf macro `STAR_PLATFORM_SOURCES`.

You should try to avoid mentioning or referring to any specific platform when configuring. Test for *features*, working with those you find, and working around those you don't; don't test for platforms and believe you can reliably then deduce what features are available.

Traditional Starlink makefiles had two phases, 'build' and 'install' (plus the various export targets). These makefiles often did on-the-fly editing of scripts as they were being installed, to edit in version numbers, or the correct path to Perl, for example. There was also implicit configuration done in the `./mk` script, which specified platform-specific compiler flags, or versions of 'tar'.

GNU-style projects, on the other hand, have three phases, 'configure', 'build' and 'install', and source file editing happens only in the first two – installation consists only of the installation of static files. Most configuration editing happens at configure time, when `.in` files are substituted with static information, such as the absolute paths to programs, determined as part of configuration. In the case where the substitution involves installation directory variables, GNU (and thus general) conventions demand that this be done at build time, since these directories involve the `$(prefix)` makefile variable, and it is deemed legitimate for the user to specify a different value at build time (`make prefix=xxx`) from that specified or implied at configuration time. The user may then specify a different prefix again at install time (`make prefix=yyy install`) for the purposes of relocating or staging the install, but this must not invalidate the value of `$(prefix)` which may have been compiled into applications. This is discussed in some detail in section 4.7.2 Installation Directory Variables of the autoconf manual, but you generally do not have to worry about it, since it is rather rare in practice that you have to compile installation directories into the applications and libraries that you build.

In general, whereas traditional Starlink makefiles quite often performed spectacular gymnastics at install time, GNU-style makefiles generally do nothing at install time, other than occasionally adding extra material to the install via one of the installation hooks supplied by automake (see section What gets installed of the automake manual).

In the traditional build system, the master source was regarded as rather private to the developer who 'owned' the code, who was free to use whatever occult means they desired to produce the sources which were put into the three distribution objects, 'export_source' (just the source), 'export_run' (just the executable) and 'export' (both). Now, everything should be put into the CVS repository, including any code-generation tools either as separate packages or as local scripts, and it is this source set which is used for nightly builds and the like. If you need tools to generate some of the distributed sources, and they cannot be included in the package for some reason, they should be checked in, and your component should declare a 'sourceset' dependency on the required tool (see Sec. A.17).

With these remarks out of the way, the following is a description of the steps involved in bringing first an application into the new fold, and then a third-party component.

4.1 Autoconfing a library

The example here shows the autoconfing of the adam library, chosen simply because it's relatively simple.

Make a directory to hold the package, and add it to the repository

```
% pwd
<cvs-checkout>/libraries/pcs
% mkdir adam
% cvs add adam
Directory /cvs/libraries/pcs/adam added to the repository
% cd adam
```

Get the complete set of source files, and check them in to the repository. This means unpacking all the files in `adam_source.tar`, which you can find in `/star/sources/pcs/adam` (as it happens).

In this case, the `adam_source.tar` distribution tarball is a suitable set of sources. This is not always true, since some Starlink distributions – especially some of the larger libraries and applications – do quite elaborate processing of their sources in the process of creating this 'source' tarball; in these cases, you should attempt to obtain a more fundamental set of sources from the package's developer (if that is not you).

The ideal source for new code is a CVS or RCS repository. A CVS repository is easy to import – you just tar it up and unpack it into the correct place within the Starlink repository. An RCS repository is barely harder, with the only difference being that you have to create the Starlink repository directory structure by hand, and copy the RCS files into place within it. The only gotcha with this route is that you must make sure that the permissions are correct on the resulting repository: you must make sure that everyone who should have any access to the repository can read *and write* each of the directories.

CVS repository access is controlled by groups (at least when the repository is shared), and so each directory within the repository *must* have a suitable group ownership, with group-write permissions; each directory must also have the setgid bit set, so that any directories created within it inherit its gid. Make sure you check this as soon as you put the repository in place, or else everyone will have problems with the repository. Within the Starlink repository in particular, all participants are part of the `cvs` group. In short, you can set the correct permissions on an imported directory `foo` with the commands

```
% find foo -type d | xargs chgrp cvs
% find foo -type d | xargs chmod g+sw
```

This sets the group-owner of each directory to be `cvs`, and sets the group-write and set-group-id bits in the permissions mask (don't tinker with file permissions, since these affect the permissions of the checked out files). You need not worry about file or directory ownership, since this always ends up being the last person who committed a file. Note: The instructions here are based on observation of CVS repositories; the actual requirements don't seem to be formally documented anywhere.

Add *all* of the source files, including files like the `mk` script and the old `makefile`, which we are about to remove. Tag this initial import with a tag <*component*>`-initial-import`, so that it is possible to recover this old-style distribution if necessary. As mentioned above, it is not always completely clear what constitutes the old-style source set: so don't do this step mechanically, use your judgement, and above all avoid losing information. Note also that some of the infrastructure libraries were added before we settled on this particular tagging practice, and so lack such an initial tag.

In this present case, the `adam_source.tar` file includes a Fortran include file containing error codes, `adam_err`; there are two problems with this.

Firstly, the filename should be uppercase: the file is generally specified within the program in uppercase, and it should appear thus on the filesystem. The traditional makefile works around this by creating a

link from `adam_err` to `ADAM_ERR`, but this won't work (and indeed will fail messily) on a case-insensitive filesystem like HFS+, used on OS X.

The second problem is that this is a generated file, though the source is not distributed, is probably misplaced, and in any case the generation was probably done on a VAX a decade ago. All is not lost, however. The functionality of the VAX 'message' utility is duplicated in the application *messgen*, in a component of the same name, along with an application *cremsg* which constructs a source file from this message file. Thus it is neither `adam_err` nor `ADAM_ERR` which we should check in, but the source file `adam_err.msg` which we reconstruct using *cremsg*. Thus with error files, it is the (probably reconstructed) `.msg` file which should be checked in to the repository, and *not* the `_err`, `_err.h` or `fac_xxx_err` files which you may have found in the old-style distribution.

The file `adam_defns` is similar, but this is genuinely a source file, so we need do nothing more elaborate than rename it to `ADAM_DEFNS`, then add it to the repository and remove the original lowercased version. Many packages have one or two `xxx_par` files, and these should be similarly renamed to `XXX_PAR`.

```
% tar xf /star/sources/pcs/adam/adam_source.tar
% chmod +x adam_link_adam
% cremsg adam_err
% cvs add *.f mk makefile adam_link_adam adam_defns.h adam_defns adam_err.msg
...
cvs server: scheduling file ‘adam_err.msg’ for addition
cvs server: use ’cvs commit’ to add these files permanently
% cvs commit -m "Initial import"
% cvs tag adam-initial-import
% mv adam_defns ADAM_DEFNS
% cvs remove adam_defns
% cvs add ADAM_DEFNS
% rm mk makefile
% cvs remove mk makefile
```

Note that CVS preserves access modes when it stores files, so we should make sure that the script `adam_link_adam` is executable *before* checking it in, and we need not bother to make it executable as part of the build process. On the other hand, scripts which are substituted by configure do need to be made executable explicitly, which you do by a variant of the `AC_CONFIG_FILES` macro. The macro invocation

```
AC_CONFIG_FILES(foo, [chmod +x foo])
```

substitutes `foo` from source file `foo.in` and then makes it executable. Note that the sequence

```
AC_CONFIG_FILES(foo)
chmod +x foo
```

would *not* work, since this is one of the cases where autoconf macros do not simply expand inline to shell code. For further discussion, see the section Performing Configuration Actions in the autoconf manual.

Create files `configure.ac`, `Makefile.am` and `component.xml.in` by copying the templates in the starconf buildsupport directory (`starconf -show buildsupportdata`); the fields in the `component.xml.in` file are discussed in Sec. 2.2.4. If you have an editor that can use it, you might also want to create a link to the DTD used for the `component.xml` file, which is in the same directory. Edit these files as appropriate, using information in the original Starlink makefile for guidance (so it's useful to keep a copy of the original `makefile` handy, rather than simply deleting it as illustrated above). Then check the files in.

What edits should you make?

The adam `Makefile.am` file looks as follows:

```
bin_SCRIPTS = adam_link_adam
lib_LTLIBRARIES = libadam_adam.la
libadam_adam_la_SOURCES = $(PUBLIC_INCLUDES) $(F_ROUTINES)
include_HEADERS = $(PUBLIC_INCLUDES)
PUBLIC_INCLUDES = ADAM_ERR ADAM_DEFNS adam_defns.h
F_ROUTINES = \
        adm_acknow.f adm_getreply.f adm_getreplyt.f adm_path.f \
        adm_prcnam.f adm_receive.f adm_reply.f adm_send.f \
        adm_sendonly.f adm_sendt.f adm_trigger.f
```

Four out of six of these variable declarations are variables meaningful to automake (see Sec. 2.1.2), and the other two are simply copied from the original `makefile`.

The adam `configure.ac` looks like this:

```
AC_INIT(adam, 3.0, starlink@jiscmail.ac.uk)
AC_PREREQ(2.50)
AM_INIT_AUTOMAKE(1.8.2-starlink)
AC_CONFIG_SRCDIR([ADAM_DEFNS])
STAR_DEFAULTS

AC_PROG_FC
AC_PROG_LIBTOOL
STAR_CNF_COMPATIBLE_SYMBOLS

STAR_DECLARE_DEPENDENCIES(build, [sae messys])
STAR_DECLARE_DEPENDENCIES(link,  [chr psx ems messys])

dnl    There is no .msg file in this directory.  The ADAM_ERR file
dnl    contains only a single definition, of the parameter ADAM__OK.

AC_CONFIG_FILES(Makefile component.xml)
AC_OUTPUT
```

The first five lines are straightforward boilerplate (see Sec. 2.1.1). The next three find a Fortran compiler, declare that we want to use libtool to build our libraries, and finally that we wish the symbols in that library to be of the sort that the CNF package is able to handle (see Sec. A.16). The 'FC' autoconf macros will search for a Fortran compiler by looking for a f95 compiler before looking for a f77 compiler; if you know or discover this is inappropriate, then you can constrain the Fortran dialect that `AC_PROG_FC` will look for by giving a value for its optional dialect argument. Macro `AC_PROG_FC` is not yet documented in the autoconf manual, but see Sec. B.

After that, we declare the dependencies. The dependencies you work out by any and all means you can. For a library, the set of 'build' dependencies is determined by the set of components which supply files which the code here includes. Grepping for all the Fortran `INCLUDE` statements and all the C `#include` directives is a good start. For link dependencies, grepping for `CALL` lines is useful for Fortran, and grepping for

```
= *\([A-Za-z][A-Za-z]*__*[A-Za-z][A-Za-z]*\) *(.*
```

will probably be handy. In fact, the script

```
#! /bin/sh -
sed -n \
        -e 's/^ *\(INCLUDE\|include\) *'\(.*\)'.*/_ \2/p' \
        -e 's/^ *\(CALL\|call\) *\([A-Za-z]*_[A-Za-z]*\).*/\2/p' \
        -e 's/.*= *\([A-Za-z][A-Za-z]*__*[A-Za-z][A-Za-z]*\) *(.*/\1/p' \
        ${1+"$@"} | \
    sort | \
    uniq
```

should give you the raw material for most of the dependencies you need. It doesn't really matter too much if you get this wrong – you might cause something to be built slightly later than it might be in the top-level bootstrap, might cause some eventual user to have to download one package more than they have to, or might create a circular dependency and break the nightly build, in which case you'll find out soon enough.

See Sec. A.17 for the description of the various types of dependencies you can specify here.

By the way, remember (because everyone forgets) that there are no components `err` and `msg`: all of the `err_` and `msg_` functions are in the `mers` component.

The next couple of lines tells you that we lied outrageously, above, when we were talking about `.msg` files. Though the remarks there are true enough in general, the `ADAM_ERR` file is special, and doesn't come from any `.msg` file. This is surprising enough that it's worth making a remark to this effect in the `configure.ac` file.

Finally, we list the files that should be configured. Essentially all starconf-style configure files should have at least these two files mentioned here.

Now run starconf. As described in Sec. 2.2, this adds some required files, and checks that the directory looks right. It will look something like this:

```
% starconf
starconf-validate: the following files are required but do not exist:
        bootstrap
starconf-validate: the following files should be checked in, but aren't
        Makefile.am configure.ac bootstrap component.xml.in component.xml
Configuring with STARCONF_DEFAULT_STARLINK=/export3/sun
Configuring with STARCONF_DEFAULT_PREFIX=/export3/sun
Creating bootstrap
```

That complained that the file bootstrap wasn't present, and then went on to install one for you; it listed a number of files which should be checked in; and it added a bootstrap script. The starconf application actually does the checking by running the *starconf-validate* script, which you can run yourself independently if you wish.

Now you have a bootstrap file, so run it:

```
% ./bootstrap
starconf-validate: the following files should be checked in, but aren't
        Makefile.am configure.ac bootstrap component.xml.in component.xml
Configuring with STARCONF_DEFAULT_STARLINK=/export3/sun
Configuring with STARCONF_DEFAULT_PREFIX=/export3/sun
File bootstrap already exists, not overwriting
autoreconf --install --symlink
configure.ac: installing `./install-sh'
configure.ac: installing `./missing'
```

The bootstrap script always re-runs starconf if it can, so this reminds you that you still haven't checked those files in. It also runs autoreconf (Sec. 2.1.4) for you, installing the helper files that requires, and constructing `configure` from `configure.ac` and `Makefile.in` from `configure.ac` and `Makefile.am`.

Now, finally, you can try `./configure` and `make`. That might just work.

Iterate until success.

When the code is working, you will might want to add it to the set of components which are explicitly built. To do this, add it to the `ALL_TARGETS` variable in the top-level `Makefile.in`. Next, go to the parent directory of the directory you have just added: the `configure.ac` file there will almost certainly have a skeleton `configure.ac` which includes a `AC_CONFIG_SUBDIRS` line, to which you should add your newly working directory.

4.2 The build system 'interface'

The previous section describes the steps required to bring a component into the build system. This section makes reasonably explicit what was implicit in that previous section, by describing the 'interface' between the build system and a particular component. The word 'interface' is in scare-quotes there because this interface isn't formal and isn't enforced, but it should be useful as a check-list and as an overview.

1. `component.xml`: every component has to have an XML-valid instance of this, as defined by `componentinfo.dtd` with element `<component>` at the root level (see Sec. 2.2.4).

2. A Makefile which implements most/all of the GNU standard targets. I think we actually use 'all' (default), 'install', 'check' and 'dist' as part of the build system, and I've mentioned 'clean', 'distclean', and 'maintainer-clean' elsewhere in this document. The GNU coding standards add 'uninstall', 'install-strip', 'mostlyclean', 'TAGS', 'info', 'dvi', none of which we probably care much about (uninstalling should probably be handled by whatever package-management tool we settle on, rather than a makefile). The build system does `cd xxx; make; make install` for component `cpt`, where `xxx` is the contents of `/componentset/component[@id='cpt']/path` in the `componentset.xml` file. This requirement comes for free, given that you use a `Makefile.am` file and automake, and this requirement is mentioned only for completeness.

3. Doing 'make install' additionally installs a `$prefix/manifests/cpt` manifest, which is a valid instance of the `componentinfo.dtd` DTD with the `<manifest>` element at the root level. Again, this step comes for free when you use the automake system.

4. Bootstrapping: the top-level bootstrap script includes a call to 'autoreconf'. This configures the whole tree, using autoconf's built-in support for recursing, based around the macro `AC_CONFIG_SUBDIRS`, and you should make sure that any new components are pulled in to this mechanism. Apart from that, the behaviour of the `./configure` scripts isn't really part of this 'interface', except that components are linked in to the tree-wide configure by virtue of being mentioned in the `configure.ac` scripts in their parent (see the 'bridging' scripts in `applications/configure.ac` and `libraries/configure.ac`). If, for some reason, you wished to incorporate a large number of components in a new tree (perhaps you want to include some complicated tree of Perl modules, for example), then you would similarly hand-write some 'bridging' scripts, which preserve the property that your new tree is configured (doing whatever is required in a particular situation) when its parent is.

4.3 Distribution of components

The `Makefile.in` files which automake generates generally handles distribution pretty successfully, and the command `make dist` will usually do almost all the work of packing your files into a distribution which can be built portably. In some cases, however, you have to give it a little help.

There are two potential problems. Firstly, automake may not be able to accurately work out the set of files which ought to be distributed. Consider the following makefile fragment (this, along with the other examples in this section, is from the AST distribution, which presents a variety of distribution problems):

```
noinst_PROGRAMS = astbad
astbad_SOURCES = astbad.c pointset.h
AST_PAR: ast_par.source astbad
    sed -e 's/<AST__BAD>/'`./astbad | tr 'E' 'D'`'/' ast_par.source >$@
```

Automake packages anything mentioned in a `_SOURCES` variable, so `astbad.c` and `pointset.h` are included in the distribution automatically. However it does not attempt to work out every consequence

of the makefile rules, and so fails to spot that `ast_par.source` is going to be needed on the build host. In general, a file which is mentioned only in a makefile dependency will not be automatically distributed by automake. Files such as this should be included by the distribution by listing them in the value of the `EXTRA_DIST` variable:

```
EXTRA_DIST = ast_par.source
```

Automake also supports the `dist_` and `nodist_` prefixes to automake variables. These can be used to adjust automake's defaults for certain primaries. Automake does not distribute `_SCRIPTS` by default (this is since they are sometimes generated, but I for one find this counter-intuitive), so if you want a script to be distributed, you must use a prefix:

```
bin_SCRIPTS = ast_link
dist_bin_SCRIPTS = ast_link_adam
noinst_SCRIPTS = ast_cpp
dist_noinst_SCRIPTS = makeh
```

We see all four common possibilities here: `ast_link_adam` and `makeh` are needed in the distribution but need no configuration, and so should be distributed as they stand; `ast_link` and `ast_cpp` are configured, so these files should not be distributed, since the corresponding `.in` files are (as a result of being mentioned in `AC_CONFIG_FILES`). Also `ast_cpp` and `makeh` are used only to build the library, and are not installed. You could get the same effect by listing `ast_link_adam` and `makeh` in `EXTRA_DIST`, but it is probably a little less opaque if all the information about particular files is kept in one place.

Incidentally, the other occasionally important 'prefix-prefix' like `dist` is `nobase`. See Sec. 5.2.

On the other hand, files in `_SOURCES` variables are distributed by default, so you must turn this off if one of these files is generated at configure time:

```
libast_la_SOURCES = \
    $(GRP_C_ROUTINES) \
    $(GRP_C_INCLUDE_FILES) \
    $(GRP_F_INCLUDE_FILES) \
    ast_err.h
nodist_libast_la_SOURCES = \
    ast.h \
    AST_PAR
```

Though the set of included files is deterministic, I find it is not terribly predictable, and the best way to do this sort of tidyup is by making a distribution, trying to build it, and thus discovering which files were left out or included by accident. There is no harm in listing a file in `EXTRA_DIST` which would be included automatically.

For fuller detail on automake's distribution calculations, see section What Goes in a Distribution of the automake manual.

The second distribution problem is that some Starlink components do quite a lot of work at distribution time, building documentation or generating sources, generally using programs or scripts which are not reasonably available on the eventual build host. This is in principle out of scope for automake and autoconf, but since it is common and fairly standardised in Starlink applications, Starlink automake and autoconf provide some support for pre-distribution configuration.

All configuration tests in `configure.ac` should be done unconditionally, even if they are only meaningful prior to a distribution – they are redundant afterwards, but cause no problems. Any files which should be present *only* prior to the distribution should be listed in `configure.ac` inside macro STAR_PREDIST_SOURCES. The `./configure` script expects to find either all of these files or none of them, and if it finds some other number, it will warn you. If it finds these files, it concludes that you are

in a pre-distribution checkout, and sets the substitution variable `@PREDIST@` to be empty; if it finds none, it concludes that you are in a distributed package, and defines `@PREDIST@` to be the comment character `#`. This means that makefile fragments which are only usable prior to distribution should all be prefixed with the string `@PREDIST@`, and they will thus be enabled or disabled as appropriate. The distribution rules mentioned above mean that any configuration of such undistributed files must be done by hand in the `Makefile.am`, and not by `AC_CONFIG_FILES`, since this macro automatically distributes the files implied by its arguments.

For example, the AST `configure.ac` has:

```
STAR_LATEX_DOCUMENTATION([sun210 sun211], [sun210.htx_tar sun211.htx_tar])
STAR_PREDIST_SOURCES(sun_master.tex)
STAR_CHECK_PROGS(star2html)
STAR_CHECK_PROGS(prolat, sst)   # prolat is part of SST
```

The `sun_master.tex` file is used when the SUN/210 and SUN/211 files are being generated, and should not be distributed. Since the process of generating the documentation uses application `star2html`, we check for this and for `prolat` (and thus do this redundantly even after distribution). Most components can get away with the one-argument version of `STAR_LATEX_DOCUMENTATION` which avoids these complications, and does the equivalent of the `star2html` check internally.

This AST configure script also has

```
STAR_PREDIST_SOURCES(error.h.in version.h.in)
```

and carefully avoids calling `AC_CONFIG_FILES(error.h version.h)`. It still has to configure these files prior to distribution, so this has to be done in the `Makefile.am`:

```
@PREDIST@predist_subs = sed \
@PREDIST@    -e 's,@star_facilitycode\@,$(star_facilitycode),' \
@PREDIST@    -e 's,@PACKAGE_VERSION\@,$(PACKAGE_VERSION),' \
@PREDIST@    -e 's,@PACKAGE_VERSION_MAJOR\@,$(PACKAGE_VERSION_MAJOR),' \
@PREDIST@    -e 's,@PACKAGE_VERSION_MINOR\@,$(PACKAGE_VERSION_MINOR),' \
@PREDIST@    -e 's,@PACKAGE_VERSION_RELEASE\@,$(PACKAGE_VERSION_RELEASE),' \
@PREDIST@    -e 's,@PERL\@,$(PERL),'
@PREDIST@error.h: error.h.in
@PREDIST@        rm -f $@; $(predist_subs) $< >$@
@PREDIST@version.h: version.h.in
@PREDIST@        rm -f $@; $(predist_subs) $< >$@
```

(this is the same technique that was illustrated in passing in the discussion of 'installation locations' in Sec. 2.1.2). The leading `@PREDIST@` strings mean that this stanza causes no problems after distribution, when the `error.h.in` files are not present. See Sec. A.27 for more details.

This is admittedly a rather crude technique, but it is a lot less fragile than the more elegant alternatives.

4.3.1 Making a distribution

In general, the details of making distributions are outside the (current) scope of this document. However there is one aspect we can usefully mention, since it interacts with the buildsupport tools which configure the distribution.

Normally, when you `./bootstrap` a directory, it arranges to install it by default in a location governed by starconf (see Sec. 2.2 for details). Since this default is a location on your local machine, it might not be appropriate for a distribution tarball. In that case, you will probably want to ensure that the built distribution will install into `/star` (or `/usr/local` perhaps). There are multiple ways you can do this.

4.3.1.1 A distribution from the complete tree

If you are doing this for a complete tree, with the intention of building all the software and rolling distribution tarballs with the result, then the procedure you should use is as follows.

(1) Check out the repository as usual.

(2) Identify the directory you want to be the default installation location for the distributed components, and the directory in which the distributed tarball will find other tools. These will typically be the same, but if you are distributing a development or updated version, you might want the default prefix to be `/stardev` but the default Starlink directory to remain `/star`. Set these as the `STARCONF_*` defaults.

```
% export STARCONF_DEFAULT_PREFIX=/star
% export STARCONF_DEFAULT_STARLINK=/star
```

If this directory really is just `/star` then these settings could in principle be omitted and the variables simply left unset, since `/star` is the default for both, but declaring them explicitly avoids ambiguity. The directory mentioned here need not exist or be writable.

(3) Identify a directory tree to hold the products as you build them; I'll use `/tmp/star` as an example. Since this directory will receive the the buildsupport tools, and other tools built during the build, it will need to be included in your PATH.

```
% PATH=/tmp/star:$PATH
```

(4) Bootstrap the tree, creating as you do so a set of buildsupport tools which are put in this working tree. If you do not add this option then the bootstrap will attempt to install the buildsupport tools in `/star`.

```
% ./bootstrap --buildsupport-prefix=/tmp/star
```

(5) Configure the tree, here overriding the `STARCONF_*` defaults.

```
% ./configure -C --prefix=/tmp/star --with-starlink=/tmp/star
```

This will install the components into `/tmp/star`, with later components using tools installed there earlier, but with the `./configure` scripts within those components written so that they will still install in `/star` by default.

(6) Then build the tree as usual.

```
% make world
```

The manifests will end up in `/tmp/star/manifests`.

After this process has completed, if you go to a built component and make a tarball using `make dist`, then this tarball will install in `/star` by default, which you can check using `./configure -help`.

4.3.1.2 Distributing a single component

If, on the other hand, you wish to create a distribution tarball of a single component, then you do not have to reconfigure the entire tree to do so (fortunately).

The more robust way is to create a special set of buildsupport tools which have the desired directory as their default prefix. Say you want the distribution to install by default in `/star`, but have these special buildsupport tools installed in `/local-star/makedist` (for example) . Do that as follows:

```
% STARCONF_DEFAULT_PREFIX=/star ./bootstrap --buildsupport \
    --buildsupport-prefix=/local-star/makedist
```

(or use `env ...` on csh-type shells; you would typically specify `STARCONF_DEFAULT_STARLINK` here, too).. Setting the two `STARCONF_*` variables is actually redundant, here, since both have `/star` as their default, but it does no harm and can make things usefully explicit.

At this point, you can check out the appropriate version of your software (probably via a CVS export of a particular tag), put this `/local-star/makedist/bin` in your path (rehashing if you are using csh), and call `./bootstrap; ./configure; make dist` as usual. The command `./configure -help` will show you the default prefix for this `./configure` script.

An alternative way is to use the `acinclude.m4` method described in Sec. 2.2.1, setting `OVERRIDE_PREFIX` to `/star` (for example). When you run `./bootstrap` after that, the given prefix will be used instead of the prefix baked into the installed buildsupport tools.

That is, to create a distribution of component `foo`, located in `libraries/foo` in the repository, you should *tag* the source set with a suitable release tag, such as `foo-1-2-3`, and then, in a temporary directory, do the following:

```
% cvs -d \emph{???} export -r foo-1-2-3 libraries/foo
% cd libraries/foo
% echo 'm4_define([OVERRIDE_PREFIX],[/star])' >acinclude.m4
% ./bootstrap
% ./configure
% make dist
```

This will leave a tarball such as `foo-1.2-3.tar.gz` in the current directory.

4.4 Regression tests

Automake provides some simple support for regression tests. There is a (terse) description of these in the automake manual, in the section Support for test suites, but it lacks any example. You run the tests with `make check`, after the build, but before the component is installed.

You can set up tests as follows.

```
TESTS = test1 test2
check_PROGRAMS = test1 test2

test1_SOURCES = test1.f
test1_LDADD = libemsf.la libems.la `cnf_link`

test2_SOURCES = test2.c
```

The `TESTS` variable lists a set of programs which are run in turn. Each should be a program which returns zero on success, and if all the programs return zero, the test is reported as a success overall. If a non-portable test makes no sense on a particular platform, the program should return the magic value 77; such a program will not be counted as a failure (so it's actually no different from 'success', and the difference seems rather pointless to me). A `PROGRAMS` 'primary' (see Sec. 2.1.2 for this term) indicates that these are programs to be built, but the 'prefix' `check` indicates that they need be built only at 'make check' time, and are not to be installed.

The `SOURCES` primary is as usual, but while the `test2` program is standalone (it's not clear quite how this will test anything, but let that pass), the `test1` program needs to be linked against two libraries, presumably part of the build. We specify these with a `LDADD` primary, but note that we specify the two libraries which are actually under test as two *libtool libraries*, with the extension `.la`, rather than using the `-lemsf -lems` `` `cnf_link` `` which `` `ems_link` `` uses as its starting point (this example comes from

the `ems` component). That tells libtool to use the libraries in this directory, rather than any which have been installed.

The fact that test programs must return non-zero on error is problematic, since Fortran has no standardised way of controlling the exit code. Many Fortran compilers will let you use the `exit` intrinsic:

```
rval = 1
call exit(rval)
```

to return a status. Since this is test code, it doesn't really matter that this *might* fail on some platforms, but if this worries you, then write the test code as a function which returns a non-zero integer value on error, and wrap it in a dummy C program:

```
test1_SOURCES = test1.f test1_wrap.c
test1_wrap.c:
        echo "int main() { exit (test1_()); }" >test1_wrap.c
```

If the tests you add use components other than those declared or implied as component dependencies (see Sec. A.17), then you should declare the full set of test dependencies using `STAR_DECLARE_DEPENDENCIES([test], [...])`.

4.5 Importing third-party sources

NOTE: this section is subject to a little change. The guidance in this section is to some extent subject to change as we get more experience including third-party sources into the tree.

If an application needs to rely on a non-Starlink application, and especially if it relies on a modified version, then the sources for that application should be checked in to the `thirdparty/` part of the tree.

There are two stages to this. Firstly we have to import a distribution version of the software, and secondly we have to bring it in to the Starlink build system.

The example here is the GNU m4 distribution, which is one of the buildsupport tools necessary on Solaris, which has a non-GNU m4 (autoconf relies on language extensions in GNU m4, which is therefore required). The first step is rather mechanical, the second better introduced by example.

First, we get and import the sources (see the fuller details in the CVS manual):

```
% cd /tmp
% wget http://ftp.gnu.org/pub/gnu/m4/m4-1.4.tar.gz
% tar xzf m4-1.4.tar.gz
% cd m4-1.4
% cvs -d :ext:username@cvs.starlink.ac.uk:/cvs import \
    -ko -m "Import of GNU m4 1.4" \
    thirdparty/fsf/m4 FSF m4-1-4
```

The `-ko` option turns off any keyword expansion for the newly-imported files. Thus they will retain the values they had when they were imported. This appears to be the practice recommended by the CVS manual, though it is not absolutely clear that it is best, and this should not be taken as a firm recommendation.

The `thirdparty/fsf/m4` argument is the location of the new component within *our* repository, the path to which will be created for you if necessary. FSF – the 'Free Software Foundation' – is the 'vendor' in this case, and this is used for the location within the `thirdparty/` tree as well as the next CVS argument. CVS uses this vendor argument as the name of the branch this new import is nominally located on.

Finally, this import command tags the imported files with the tag you give as the last argument. This should use the same convention as other tags within the Starlink repository, namely the component name and version number.

Note that we import a *distribution tarball* of the source, namely one including the `configure` script and any other generated files. This is because we cannot reliably bootstrap the original sources if they, for example, and also so that we can reliably track any future releases of the tarball. Even if the component has a public CVS archive, resist the temptation to import a snapshot of that.

Now that the source set is in the repository, you can go back to your checkout tree and check the new component out:

```
% cd <checkout-tree>
% cvs -d :ext:username@cvs.starlink.ac.uk:/cvs co thirdparty/fsf/m4
[blah...]
% cd thirdparty/fsf/m4
% cvs status -v configure.in
===================================================================
File: configure.in      Status: Up-to-date

   Working revision:    1.1.1.1
   Repository revision: 1.1.1.1 /cvs/thirdparty/fsf/m4/configure.in,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      -ko

   Existing Tags:
        m4-1-4                          (revision: 1.1.1.1)
        FSF                             (branch: 1.1.1)
```

We see that the newly-imported files have been put on the 1.1.1 branch, named `FSF`. Any files we add to this component, and any files we modify, will go on the trunk.

What you do next depends on how easy it is to configure the new component. We'll look at three examples, the `m4` component, containing the GNU version of m4, the `cfitsio` component, containing the HEASARC FITS library, and the `tclsys` component, containing a distribution of Tcl.

(1) The `m4` configuration is quite regular (as befits a core GNU component), and so admits of reasonable adaptation in place, by simply editing the configuration files included in the distribution (and then, despite what we said above, regenerating the distributed `./configure` script.

(2) The `cfitsio` distribution comes with its own configure scripts, but they are generated using a rather old version of autoconf, so we want to avoid touching them. However the build is basically rather simple, and only a few built files need to be installed. This component shows how to wrap a distribution's own configuration in a straightforward way.

(3) The `tclsys` component, on the other hand comes with its own fearsomely intricate configuration and installation mechanism, which we want to disturb (or indeed know about) as little as possible. This section shows how to wrap such an installation in the most general way.

4.5.1 Configuring m4

The first thing to do is to add the `component.xml.in` file:

```
% cp ‘starconf --show buildsupportdata‘/template-component.xml.in \
      component.xml.in
```

after which we edit `component.xml.in` appropriately. This editing turns out to be slightly more intricate than usual: the `configure.in` file is old-fashioned enough that it does not define the substitution variables that `component.xml.in` is expecting, and it requires a slightly closer inspection of `configure.in` to determine what these should be (`@PRODUCT@` and `@VERSION@` in this case). Although this step seems redundant, it is best to have this file configured rather than completely static, so that the generated `component.xml` will remain correct when and if a new version of the 'm4' component were imported. Having said that, ensure that the `<bugreports>` element in the `component.xml.in` file points to a Starlink address – we don't want bugs in *our* modifications to be reported to the original maintainers.

Note that it is the now-deprecated `configure.in` file that is the autoconf source in this component, and that it has now-deprecated syntax; there is no need to update this. When we run autoreconf, we discover that this is not the only obsolete feature, and we have to do some futher mild editing of `configure.in` before it is acceptable to the repository version of autoconf, though it still produces a good number of warnings. These don't matter for our present purposes: the `./configure` which autoreconf produces (and which we commit) still works, and the component builds and runs successfully. If the `configure.in` were old enough that our current autoconf could not process it at all, then we might consider retrieving and temporarily installing an older version of autoconf – the `./configure` script includes at the top a note of the autoconf version which produced it.

In order for the new component to be a good citizen in the Starlink build tree, it needs to install a file manifest as part of the `install` target. Add to the `configure.in` file the lines

```
: ${STAR_MANIFEST_DIR='$(prefix)/manifests'}
AC_SUBST(STAR_MANIFEST_DIR)
```

If we were using Starlink automake, this would be enough to prompt it to include support for installing a manifest in the `Makefile.in` it generates. The GNU m4 distribution does not, however, use automake, so we need to add this by hand. The following additions to `Makefile.in` do the right thing, though they are rather clumsier than the support that Starlink automake adds:

```
STAR_MANIFEST_DIR = @STAR_MANIFEST_DIR@

[...]

# Change the install target from the default
install: config.h install-manifest.xml
         for subdir in $(SUBDIRS); do \
             echo making $@ in $$subdir; \
             (cd $$subdir && $(MAKE) $(MDEFINES) $@) || exit 1; \
         done
         $(srcdir)/mkinstalldirs $(STAR_MANIFEST_DIR)
         $(INSTALL_DATA) install-manifest.xml $(STAR_MANIFEST_DIR)/$(PRODUCT)

# Add the install-manifest.xml target
install-manifest.xml: all
        rm -Rf STAGING
        mkdir STAGING
        $(MAKE) prefix=`pwd`/STAGING install
        rm -f $@
        echo "<?xml version='1.0'?>"                          >$@
        echo "<!DOCTYPE manifest SYSTEM 'componentinfo.dtd'>" >>$@
        echo "<manifest component='$(PACKAGE_NAME)'>"         >>$@
        echo "<version>$(PACKAGE_VERSION)</version>"          >>$@
        echo "<files>"                                        >>$@
        find STAGING -type f | sed 's,^.*STAGING,$(prefix),'   >>$@
        echo "</files>"                                       >>$@
        echo "</manifest>"                                    >>$@
        rm -Rf STAGING
```

(if you look at `thirdparty/fsf/m4/Makefile.in` you will see that we actually need to have `PRODUCT` and `VERSION` there instead of the more general `PACKAGE_NAME` and `PACKAGE_VERSION` above). Note that the actual installation happens within the rule for `install-manifest.xml` – this guarantees that the `$(prefix)` stored in the manifest is the same as the `$(prefix)` actually used for the installation.

Since we are creating this manifest entirely by hand, we are responsible for ensuring that the resulting file conforms to the DTD in `componentinfo.dtd` (in the starconf `buildsupportdata` directory).

We add the `component.xml.in` file to the repository, and look at what we have.

```
% cvs add component.xml.in
% cvs -n update
```

This tells us that `configure`, `configure.in` and `config.h.in` have been modified, and `component.xml.in` added, but we also discover that files `stamp-h.in`, `doc/stamp-vti` and `doc/version.texi` have also been modified, as part of the regeneration of the `./configure` script. The modification to the stamp file `stamp-h.in` we should probably commit, to avoid dependency niggles in the future, but the meaningless changes to the two `doc/` files we can just discard:

```
% cvs commit -m "Add component.xml template" component.xml.in
% cvs commit -m "...blah..." configure configure.in config.h.in
% cvs commit -m "Stamp file regenerated by autoreconf" stamp-h.in
% rm doc/stamp-vti doc/version.texi
% cvs update doc
```

We can see that the changes to `configure.in` are now on the trunk for this component, rather than the FSF branch.

```
% cvs status configure.in
===================================================================
File: configure.in      Status: Up-to-date

Working revision:    1.2
Repository revision: 1.2     /cvs/thirdparty/fsf/m4/configure.in,v
Sticky Tag:          (none)
Sticky Date:         (none)
Sticky Options:      -ko
```

4.5.2 Configuring CFITSIO

The `cfitsio` component wraps the HEASARC FITS library of the same name, and installs from it the library itself, plus a few header files. Although we probably could adapt the library's distributed autoconf script, that script is sufficiently old, and there are sufficiently few components installed, that it is less trouble to simply wrap this configuration in another one.

The `cfitsio` component contains a `cfitsio/` directory containing the distributed library source. Along with it are the usual `component.xml.in` file, copied from the template in ‘`starconf -show buildsupportdata`‘, and `Makefile.am` and `configure.ac` files, which are too different from the template ones for them to be helpful.

The `configure.ac` looks like this:

```
dnl    Process this file with autoconf to produce a configure script
AC_REVISION($Revision$)
AC_INIT(cfitsio, 2.4-90, starlink@jiscmail.ac.uk)

AC_PREREQ(2.50)
```

```
AM_INIT_AUTOMAKE(1.8.2-starlink)

AC_CONFIG_SRCDIR([cfitsio.news])

STAR_DEFAULTS

dnl    To configure CFITSIO proper we just run ./configure in the
dnl    cfitsio sub-directory. Do not invoke AC_CONFIG_SUBDIRS,
dnl    since that prompts autoreconf to try to reconfigure that directory, and
dnl    automake to assume it's allowed to play there, too.
(
    cd cfitsio
    ./configure --prefix=$prefix
)

dnl    Find required versions of the programs we need for configuration
AC_PROG_LIBTOOL
AC_PROG_MAKE_SET

STAR_LATEX_DOCUMENTATION(sun227)

AC_CONFIG_FILES([Makefile component.xml])
AC_OUTPUT
```

Note that we must use `STAR_DEFAULTS`, and we must not use `AC_CONFIG_SUBDIRS`, for the reason described in the comments above, even though configuring the subdirectories is exactly what we want to do. This particular combination of commands is sufficient to configure the `cfitsio` distribution – you might need to do different things for other third-party packages. That's all the configuration we need; how about the Makefile?

The `Makefile.am` looks like this:

```
## Process this file with automake to produce Makefile.in

dist_bin_SCRIPTS = cfitsio_link

lib_LIBRARIES = cfitsio/libcfitsio.a
# There are no sources for libcfits in this directory.
cfitsio_libcfitsio_a_SOURCES =

include_HEADERS = cfitsio/fitsio.h cfitsio/longnam.h \
    cfitsio/fitsio2.h cfitsio/drvrsmem.h

stardocs_DATA = @STAR_LATEX_DOCUMENTATION@

EXAMPLE_SOURCES = cfitsio/cookbook.c cfitsio/cookbook.f

# Put examples in dist_pkgdata
dist_pkgdata_DATA = $(EXAMPLE_SOURCES) \
    cfitsio_changes.txt CFITSIO_CONDITIONS Licence.txt cfitsio.news

# Headers and example sources are made within cfitsio
$(include_HEADERS) $(EXAMPLE_SOURCES):
        (cd cfitsio; unset CFLAGS; $(MAKE))

# Must be a separate target from above, otherwise automake doesn't notice,
# and adds its own, conflicting, target.
cfitsio/libcfitsio.a:
        (cd cfitsio; unset CFLAGS; $(MAKE))
```

```
...
```

(if you look in the `cfitsio` component, you will see that the actual `Makefile.am` does a little more than this, and is rather more copiously commented, but these are the important parts).

This uses a number of useful automake features. First of all, the assignments which control what is installed – the contents of the `_LIBRARIES`, `_HEADERS` and (via `EXAMPLE_SOURCES`) `_DATA` primaries – refer to files within the `cfitsio/` subdirectory. When these are installed, the path to them is stripped, so that `libcfitsio.a`, for example, ends up installed in `.../lib/` and not in `.../lib/cfitsio/`, as you might possibly intuit. This is the useful behaviour which is turned off by the use of the `nobase_` prefix, as described in Sec. 5.2.

Second, the presence of the `lib_LIBRARIES` line means that automake expects to find some library sources somewhere, and if it is not told where they are with a `xxx_SOURCES` variable, then it will assume a default based on the library name. In this case, of course, the sources are not in this directory, and we do not wish automake to generate rules to build them, so we have to pacify it by giving a value for the `cfitsio_libcfitsio_a_SOURCES` variable (note the canonicalisation of the library name).

Each of the `cfitsio/` targets is built in the same way, by switching to the `cfitsio/` directory and making 'all' using that directory's own Makefile. The only remaining wrinkle is that it turns out we have to specify the `cfitsio/libcfitsio.a` target by itself, rather than as part of the previous compendium target. Due, probably, to a bug, automake does not appear to 'notice' the library if it's mentioned in the compendium target, and instead generates its own conflicting target.

4.5.3 Configuring tcl

The `tcl` component (like the parallel `tk` component) is paradoxically easy to configure, partly because its distributed `./configure` script is generated by a version of autoconf too old for us to handle directly, which means that we want to avoid touching it as much as possible, but also because the Tcl distribution has an installation mechanism which is too complicated to be handled by the simple scheme in the previous `cfitsio` section. The method we use instead, as described below, actually looks simpler, but you should probably not resort to it unless necessary, since the starconf macro is uses – `STAR_SPECIAL_INSTALL_COMMAND` – is a dangerously blunt instrument.

After being unpacked and imported as described above, the top-level of the checked out Tcl distribution looks like this:

```
ptolemy:tcl> ls
CVS/       README   compat/  generic/  license.terms  tests/  unix/
ChangeLog  changes  doc/     library/  mac/           tools/  win/
```

The way that the distributed `README` tells us to compile Tcl is to change to the `unix/` subdirectory, and type `./configure; make`. We want to create files `Makefile.am` and `configure.ac` in this directory, which handle this for us.

We add a `component.xml.in` as before, plus non-standard `configure.ac` and `Makefile.am` files. The `configure.ac` should look like this:

```
dnl    Process this file with autoconf to produce a configure script
AC_REVISION($Revision$)
AC_INIT(tclsys, 8.2.3, starlink@jiscmail.ac.uk)

AC_PREREQ(2.50)
AM_INIT_AUTOMAKE(1.8.2-starlink)

AC_CONFIG_SRCDIR([license.terms])
```

```
STAR_DEFAULTS

AC_PROG_MAKE_SET

dnl    To configure Tcl, run ./configure in the 'unix' directory.
dnl    Do not invoke AC_CONFIG_SUBDIRS, since that prompts autoreconf
dnl    to try to reconfigure that directory, and automake to assume
dnl    it's allowed to play there, too.
(
    cd unix
    ./configure --prefix=$prefix
)

STAR_SPECIAL_INSTALL_COMMAND([cd unix;
        $(MAKE) INSTALL_ROOT=$$DESTDIR install])

AC_CONFIG_FILES([Makefile component.xml])
AC_OUTPUT
```

This looks rather like the `cfitsio` example, except for the addition of the new macro, which uses a parameterised version of the distribution's own install command to make the installation into the Starlink tree.

As described in Sec. A.29, this adapts the standard Makefile `install` target so that it uses the given command to make an installation. Note that the version of autoconf which generated the Tcl `Makefile.in` template was one which used the `INSTALL_ROOT` variable instead of the `DESTDIR` variable used by more modern versions, and so we have to adjust this in this command line. This `DESTDIR` variable is important, as it is used during the installation of the component, to do a staged installation, from which a manifest can be derived.

This staging step is important, and is rather easy to get wrong. If you do get it wrong, then the installed manifest will be wrong, probably causing problems later when it comes to deinstalling or packaging this component. The role of the `INSTALL_ROOT` variable above was discovered only by inspecting the `Makefile.in` in the Tcl distribution: if there were no such facility, we could (probably) fake much of it by using something like `$(MAKE) prefix=$$DESTDIR$$prefix install` in the installation macro. The distributed Tcl Makefile appears to respect the variable, but it is important to check for errors such as making absolute links, or using the current directory incautiously (search for `LN_S` or `` `pwd` `` in the `Makefile.in`).

The `Makefile.am` file is even simpler:

```
## Process this file with automake to produce Makefile.in

@SET_MAKE@
RECURSIVE_TARGETS = all clean

default: all

$(RECURSIVE_TARGETS):
        cd unix; $(MAKE) $@
```

After writing the `Makefile.am` and `configure.ac` files, all you need to do is run starconf to create a `./bootstrap` file, and check in the files which starconf suggests.

That's it!

4.5.4 Time order of third-party sources

When importing third-party code that includes pre-created `configure` scripts, `Makefile.in` etc., that you do not want to re-generate automatically, because they have inconsistent timestamps – that is say a Makefile.in file is older than an associated Makefile.am, because the initial import was in the wrong order or too quick (file systems typically resolve at one second intervals), then you may need to re-establish the time ordering of some files, but only if the imported build system is sensitive to these changes. The way to do this is by creating a file `bootstrap_timeorder` that contains a list of relative file names (to the directory containing the `bootstrap` script), in the order from oldest to newest. Look for an example of one of these files in the source tree.

4.6 Documentation

There is fairly complete support for documentation within the build system. Most Starlink documentation is in the form of LaTeX files respecting certain conventions (see SGP/28: Writing Starlink documents and SUN/199: Star2HTML for some further details), but a small amount is in a custom XML DTD, converted using the kit contained in the `sgmlkit` component, documented in SSN/70.

The simplest case is where you have one or more `.tex` files in the component directory (that is, the directory which contains the `component.xml` file). In this case, you simply declare the document numbers, and the files named after them, in a `STAR_LATEX_DOCUMENTATION` macro, as in:

```
STAR_LATEX_DOCUMENTATION([sun123 sc99])
```

This will put these document codes into the substitution variable `@STAR_DOCUMENTATION@`, which you can use to substitute in `component.xml.in`, and it will arrange to compile the LaTeX files `sun123.tex` and `sc99.tex`. This macro is documented fully in Sec. A.21.

The final step in this case is to declare that the documentation is to be built and installed in the Starlink documentation directory. You do that by including the following line in the component's `Makefile.am`:

```
stardocs_DATA = @STAR_LATEX_DOCUMENTATION@
```

(see Sec. A.32). This substitution variable expands to the default list of documentation targets (at present comprising `.tex`, `.ps` and `.htx_tar`, but subject to change in principle).

The next simplest case is where the source files are in a different directory from the `component.xml`. Indicate this by appending a slash to the end of each of the document codes located elsewhere. For example:

```
STAR_LATEX_DOCUMENTATION([sun123/ sc99])
```

This will behave as above for the `sc99.tex` file, but differently for the `sun123` document. That code, `sun123` is still included in `@STAR_DOCUMENTATION@`, but it is the variable `@STAR_LATEX_DOCUMENTATION_SUN123@` which is set to the default target list, and not `@STAR_LATEX_DOCUMENTATION@`. Documentation specified in this way does *not* – contrary to appearances – have to live in a similarly named subdirectory of the component directory; instead you will have to separately make sure that the documentation is built at build time.

Suppose, for example, you are managing the documentation for the package which contains SUN/123, and suppose the source for this is in a subdirectory `docs/sun-cxxiii` (whimsy is a terrible thing), then you would still refer to the documentation via `STAR_LATEX_DOCUMENTATION(sun123/)` as above, but in the file `docs/sun-cxxiii/Makefile.am` you would include at least the line

```
stardocs_DATA = @STAR_LATEX_DOCUMENTATION_SUN123@
```

and make sure that directory `docs/sun-cxxiii` is built, and its contents installed when appropriate, by the usual method of mentioning the directory in a `SUBDIRS` declaration in the component `Makefile.am`:

```
SUBDIRS = docs/sun-cxxiii
```

The `datacube` component uses this mechanism to build its documentation.

If you need to add extra files to the `.htx` tarball, you can do so via the `.extras` file described in Sec. A.21.

Sometimes you need to do more elaborate things to build your documentation – the `ast` component is a good example, here. In this case, you can give a non-null second argument to the `STAR_LATEX_DOCUMENTATION` macro, giving a list of makefile targets which should be built. The document codes in the first argument are still included in `@STAR_DOCUMENTATION@`, but the second argument is included verbatim in `@STAR_LATEX_DOCUMENTATION@` without any defaulting. You are responsible for adding the given targets to the Makefile and, as before, you should add `@STAR_LATEX_DOCUMENTATION@` to the `stardocs_DATA` Makefile variable.

If you have produced XML documentation, use the `STAR_XML_DOCUMENTATION` macro, which is closely analogous to the LaTeX one.

4.6.1 Components containing only documentation

Though most applications and libraries have some documentation associated with them, some documentation is not associated with any particular software, and so lives in a component by itself. These components are principally the various cookbooks and system notes, though there are a few SUNs in this category as well. Though the correct way to handle such components should be fairly clear from the discussion above, it seems worth while to make a few remarks here to clear up some ambiguities.

Like any other component, a documentation-only component must have a bootstrap file, and `Makefile.am`, `configure.ac` and `component.xml.in` files, as described in Sec. 4.1; however these will typically be rather simple.

We can look at the SC/3 configuration files for an example.

The `configure.ac` file, shorn of most of its comments, looks like this:

```
dnl    Process this file with autoconf to produce a configure script
AC_INIT(sc3, 2, starlink@jiscmail.ac.uk)
AC_PREREQ(2.50)
AM_INIT_AUTOMAKE(1.8.2-starlink)
AC_CONFIG_SRCDIR(sc3.tex)

STAR_DEFAULTS(docs-only)

STAR_LATEX_DOCUMENTATION(sc3)

AC_CONFIG_FILES(Makefile component.xml)

AC_OUTPUT
```

Note the `docs-only` option to the `STAR_DEFAULTS` macro (see Sec. A.18 for discussion). This particular `configure.ac` file is to handle document SC/3.2 – this second edition of the document is represented by the `AC_INIT` macro having a 'version' number of '2'.

The `Makefile.am` file has some mild complications:

```
## Process this file with automake to produce Makefile.in

stardocs_DATA = @STAR_LATEX_DOCUMENTATION@
starexamples_DATA = $(distdir)-scripts.tar

sc3.htx_tar: sc3-scripts.tex

# sc3-scripts.tex is an appendix to the HTML version of the document,
# comprising the scripts contained in the scripts/ directory.  These are
# also bundled in the htx tarball by virtue of being listed in the
# sc3.htx_tar.extras file.
sc3-scripts.tex: scripts
        cd scripts; \
        for f in *; do if test -f $$f; then \
            echo $$f | sed 's/\([a-z0-9_]*\).*/\\subsection{\\label{se_\1_source}\\xlabel{\1}\1}/'; \
            echo "\htmladdnormallink{Download $$f}{$$f}"; \
            echo ''; \
            echo '\begin{verbatim}'; \
            cat $$f; \
            echo '\end{verbatim}'; \
        fi; done > ../$@

$(distdir)-scripts.tar: scripts
        DN=$(distdir)-scripts; \
            mkdir $$DN; \
            for f in scripts/*; do if test -f $$f; then cp $$f $$DN; fi; done;\
            tar cf $@ $$DN; \
            rm -Rf $$DN
```

This is the complete SC/3 file. The only thing that is always required in these docs-only `Makefile.am` files is the `stardocs_DATA = @STAR_LATEX_DOCUMENTATION@` line, which has exactly the same function described in Sec. 4.6 above. However cookbooks like SC/3 often have both sets of examples (in this case, example scripts) and some automatically generated documentation, and this example illustrates how to describe both of these. The `starexamples_DATA` line specifies a file which is to be installed in the examples directory (typically `/star/examples`; see Sec. A.32), and the `Makefile.am` therefore provides the rule for generating that tarball of scripts.

In addition, the `sc3.tex` file includes (in the TEX sense) a generated file `sc3-scripts.tex`, which documents these various scripts, and so we include in this `Makefile.am` the rule for generating this file from the contents of the `scripts/` directory. We must also indicate that the `sc3.htx_tar` file depends on this generated file, and we do this by including that dependency (without the build rule, which is generated automatically) in this `Makefile.am` file.

Finally, note that this SC/3 directory uses the `.htx_tar.extras` mechanism which is described rather in passing in Sec. A.21.

4.7 Adding components: the final step

Whichever type of component you have added, the final step, after committing your changes, is to ensure that your new component and its newly-declared dependencies are integrated into the network of dependencies contained within the `Makefile.dependencies` file at the top level. To do this, you must go to the top level of a *full checkout* of the repository, make sure your new component is checked out there, then delete `Makefile.dependencies` and remake it.

```
% rm Makefile.dependencies
% make Makefile.dependencies
# ... remakes componentset.xml, and from that Makefile.dependencies
% cvs -n update
M componentset.xml
M Makefile.dependencies
```

The `Makefile.dependencies` file is generated using a Java program, so you must have a JDK in your path before starting this procedure (you might possibly need to run `./configure -no-recursion` to bring the `Makefile` there up to date with respect to `Makefile.in`). Alternatively, you can invoke make with `make JAVA=/path/to/jdk Makefile.dependencies`.

Double-check both `Makefile.dependencies` and `componentset.xml`, using `cvs diff <file>`: ensure that the right material has been added before re-committing these two files. If this diffing appears to indicate that material has been *removed* from either file, this probably means that you don't have a full or up-to-date checkout, so investigate that and fix things up before committing.

If the component you have added should be included in the 'make world' build, then you should add it to the list of targets listed in the `ALL_TARGETS` variable at the top of the top-level `Makefile.in`. You should do this only if both this component *and anything it depends upon* build successfully from scratch.

Chapter 5
FAQs

5.1 General FAQs

For FAQs specifically about installation, see the next section, and for quite detailed FAQs about the various ways that state is held in the starconf system, see Sec. 5.3.

In the examples below, the installation location (that is, `STARCONF_DEFAULT_PREFIX`) is written as `/local-star`, and the example component is usually `cpt`.

How do I control which compiler and compiler options I use? You can control the compiler at configure time by specifying the variable `F77`, `FC`, `CC` or `CXX`, as appropriate; and you add compiler options with the `FFLAGS`, `FCFLAGS` or `CFLAGS` variables. For details and warnings, see Sec. 2.1.5.

I've just been told "`make: *** No rule to make target '/local-star/manifests/cpt', needed by 'world'. St`
The component `cpt` has been referred to in a dependency, but the `Makefile.dependencies` file has no knowledge of it. Whoever added the `cpt` component forgot to do the final step mentioned in Sec. 4.7, so you should do that now.

Why does `make world` not do anything? The `make world` build works by comparing timestamps of the *installed* files in the manifests directory – that is, in something like `/local-star/manifests`, outside of the build tree. It only checks whether each component was *installed* after each of the components it depends on. If you wish to use this to (rebuild and) reinstall a given component, then simply delete the manifest file for the component in question and re-run `make world`.

Alternatively, `make` in the top-level checkout directory will recurse, and invoke `make all` in each of its children, and thus bring the whole tree up to date.

You can check the default installation location – shown as `/local-star` above – with `./starconf-status -show STARCONF_DEFAULT_PREFIX`

Also, `make world` only makes those targets listed in the variable `ALL_TARGETS` within the top-level Makefile. Thus if the component you're interested in isn't there, nor required by something that is there and that will be rebuilt, nothing will happen. See also Sec. 4.7.

See Sec. 5.3 for a few more remarks about state.

I've updated my checkout of a component. How do I best update the installation of it? The most straightforward way is to change to the directory in question, and do `./configure; make; make install` there.

Alternatively, you can stay in the top-level directory, delete the component's manifest file, `rm /local-star/manifests/cpt`, and remake it: `make /local-star/manifests/cpt`.

Actually, the component is one of the buildsupport tools. Is that different? Yes. The best way is update your checkout of the component in question, then run `./bootstrap -buildsupport` in the top level of the checkout. When the bootstrap script is run in this mode, it always deletes any buildsupport manifests, to ensure that the buildsupport tools are *always* rebuilt if updated.

It's best to avoid the `./configure; make; make install` route described in the last item. Some of the buildsupport tools need special handling, and the `./bootstrap` script knows how to do that.

I've updated my checkout and a large number of components were updated (or I'm paranoid). What's the best way to
Just doing `make` would remake everything that needs remaking, but wouldn't install it. The command `make world` wouldn't work either, because that only works with the dependencies between manifest files – this is the way that the `Makefile.dependencies` file expresses the dependencies of components on each other.

The best way to do this is as follows. After your update, and while in the top-level directory, do

```
% rm /local-star/manifests/*
% make buildsupport
% make world
```

That is, you delete *all* the manifests, and remake them. This does a `make; make install` in each component directory, and that step handles any files that were updated.

What's the best way to generate files at build time? Look at the `Makefile.am` for the `prm` component. It's a good example of using `noinst_` tools, the Fortran preprocessor, and `./configure` variables in a reasonably tidy way.

I want to do odd things at clean/distribution time. How? Odder than what is described in Sec. 4.3?

Automake also has support for extending what is cleaned and what is included in the distribution, though this does not have to be as elaborate as the support for extended installation above. See the documentation on `CLEANFILES` and `EXTRA_DIST` in the Automake manual.

I want to incorporate an external package which already has a configuration system. How? This is 'third-party code': see Sec. 4.5 for several examples.

How do I write portable Makefiles and shell scripts? The autoconf manual has a very useful collection of portability advice in its section manual *Portable Shell Programming*. Also, the Single Unix Specification, version 3 provides standardised specifications of the behaviour of Unix library functions and system utilities. The Single Unix core specification is identical to POSIX.3 (IEEE Std 1003.1) and ISO/IEC 9945:2003.

I've just been told that libtool is "unable to infer tagged configuration". What on earth is that supposed to mean?
I'm not sure what this is supposed to mean *exactly*, but what it seems to mean in practice is that libtool has become terribly confused about your compiler and hasn't been configured to recognise it. The most common reason for this is that libtool uses a different set of default compilers to those of autoconf, and in particular is not as bright about Fortran 95 compilers. The simplest way to resolve this is to define `F77` to match the compiler determined by autoconf (or indeed define `FC` and `F77` to be your preferred compiler).

Other reasons this may happen are when you force `./configure` to use a particular compiler by specifying one of the `F77`, `FC` or `CC` environment variables at configure time. You may have forgotten that you have that variable set.

Alternatively, you may be unaware that that variable is set: a known manifestation of this problem is that the IRAF startup script sets `F77` to be an IRAF-specific script, which works OK from the command-line (given that you want to use IRAF), but which leaves libtool in a confused heap.

This problem has also been spotted in the form `libtool: compile: specify a tag with '-tag'`, with apparently the same cause.

See Sec. 2.1.5 for more discussion of these variables.

I've just modified an include file, ran make in a component, but nothing has compiled Automake provides the `depcomp` command to automatically generate dependencies on include files (these are added to `Makefile.in`). Unfortunately this support doesn't extend to Fortran, only C, so modifying a file included in Fortran will not result in the files being recompiled. Also not all C compilers and platforms are supported. To avoid this problem either keep the list of dependencies up to date in the `Makefile.am`, or be prepared to clean directories during development.

It's all going horribly wrong. Help! Things to think about:

(1) Are there some stray environment variables? Environment variables such as `F77` or `CC` influence the compilers chosen at configure time, as discussed in Sec. 2.1.5. There is a particular problem with the IRAF startup script often sourced from the `.login` file: see Sec. 5.1.

(2) Are the buildsupport tools up to date? This is fairly unlikely (once the build system becomes a production system), but might be worth checking. See Sec. 3.5.

(3) Think how happy you are that life has been good to you so far.

5.2 Installation FAQs

I have an executable which creates some files during the build – how do I prevent it from being installed? Use the `noinst` prefix when you declare the program. See Sec. 2.1.2.

How do I install things in `/local-star/docs,etc,examples,help`? In the `Makefile.am` use the `DATA` prefixes `stardocs`, `staretc`, and so on. See Sec. 2.1.2 for an example.

Note that by `/local-star`, here, we refer to the directory indicated by `STARCONF_DEFAULT_STARLINK` – if you are interested in changing this when making distribution tarballs, see the next question.

I want to make distribution tarballs (for example) install into `/star`. How? This is a general question about forcing the `$prefix` for a directory. See the description of `./bootstrap` in Sec. 4.3.1, which uses a general mechanism described in Sec. 2.2.1.

Automake flattens paths when it installs things. Why? This is a feature, which you can turn off if necessary. A `Makefile.am` line like `include_HEADERS = a.h subdir/b.h` will cause both `a.h` and `b.h` to be installed in the same place, as indicated by the `_HEADERS` primary. It means that if you had some good reason for creating or distributing the header `b.h` in its subdirectory, you don't have to copy it into the top directory just prior to installation. If you specify `nobase_include_HEADERS`, on the other hand, this will cause the hierarchy to be preserved. This is documented rather in passing in the automake manual, in the section An Alternative Approach to Subdirectories (and no, I don't know why 'nobase' is supposed to be intuitive, either).

There's an example of using this in Sec. 4.5.2.

I want to install symbolic links at install time. How? That counts as 'weird' for the purposes of this discussion. See the next item.

I want to do weird things at install time. How? Automake provides a couple of escape hatches for installing extra material. These are the `install-exec-local` and `install-data-local` targets for installing platform-dependent and -independent files. The associated rules are run at install time, and can do anything, the similar `install-exec-hook` and `install-data-hook` targets are run after all other install targets, and can do post-installation cleanup and the like. See section What Gets Installed in the automake manual for details.

For example, the `dtask` component installs a special object file in the library directory. This isn't a program, or script, or library, and so isn't naturally handled by any other automake mechanism. The `Makefile.am` there includes the following:

```
dtask_main.o: dtask_main.f
install-exec-local: dtask_main.o
        $(mkdir_p) $(DESTDIR)$(libdir)
        $(INSTALL_PROGRAM) dtask_main.o $(DESTDIR)$(libdir)
        $(MANIFEST) && echo "MANIFEST:$(DESTDIR)$(libdir)/dtask_main.o" || :
```

This first creates the library directory if necessary. The rules then use the `$(INSTALL_PROGRAM)` command to install the object file in the library directory (you'd use `$(INSTALL_DATA)` command for a data target) . The installation location is `$(libdir)`, which is a location which includes the `$(prefix)` variable, so that this install rule behaves as other such rules do. Also, it prefixes the installation location with `$(DESTDIR)`: this is usually the empty string, but it can be set to something else to perform a staged installation. Finally, if the `$(MANIFEST)` variable evaluates to true, then we echo the echo the name of installed file to `stdout` prefixed by the string `MANIFEST:`; note that the command ends with '`|| :`', which ensures that this composite command succeeds even if `$(MANIFEST)` evaluates to `false`. This is a Starlink extension, and is how the manifest file is built up.

Yes, this is a fuss, but it's the consequence of using the `install-xxx-local` back door. Each of these features is important if this hand-made install rule is to behave the same way as the automatically-generated rules. This technique is also inherently precarious, since if there were ever some reason to modify the installation conventions (for example to modify the way the manifests were built up), then each such back-door would have to be adjusted by hand. That is, don't use this mechanism unless you really have to.

In particular, you often need to create links in this situation. To do that portably (symbolic links are not required by POSIX, and so the autoconf macros are able to emulate them if necessary), put the `AC_PROG_LN_S` macro in your `configure.ac`, and use the `$(LN_S)` macro in your `Makefile.am`. Note that the autoconf manual that this macro does not in principle support a path in the second argument; thus do not write `$(LN_S) /target/TARGET /link/LINK`, but instead

```
cd $(DESTDIR)/link; rm -f LINK; $(LN_S) $(DESTDIR)/target/TARGET LINK
```

The option `rm -f` is portable. Note that we have included the important `$(DESTDIR)` variable here, and that the `/link` and `/target` locations here should probably be one or other of the `$(???dir)` variables instead.

5.3 State FAQs

This is only a summary of the various ways that state is held within the Starlink build system. The details are in Sec. 2.4.

How do I get rid of state in a checked-out directory? One of `make clean`, `make distclean` or `make maintainer-clean` should do what you want. The first gets rid of compiled files, the second everything but distributed files, and the third returns the directory to an almost pristine state, though without deleting `./configure` or the files it needs to run.

The nuclear option for clearing state from a directory `ping` is to delete it and check it out afresh:

```
% cd ..
% cvs release -d ping      # or: mv ping ping-old
% cvs update -d ping
```

The CVS 'release' command tells CVS you have finished with a particular checkout. With the `-d` option, it is basically `rm -rf` *except* that it checks you have no modified but uncommited files left within `ping`.

How do I start again? As mentioned above, `make maintainer-clean` in the top level should recurse and make `maintainer-clean` in each of the children. That unmakes and unconfigures everything, but doesn't undo the bootstrapping. The bootstrapping done by `./bootstrap` doesn't depend on any state except the contents of the manifest files, and to undo that, you need only delete the manifest.

For maintainer-clean to work, one sometimes needs to make extra declarations in `Makefile.am`. There could well be components where maintainer-clean doesn't work as fully as it ought. Do mention those if you spot them.

How do I control where a component is installed? In a variety of ways. In the list below, each mechanism effectively supplies the default for the mechanism before it.

(1) You can specify an installation prefix at install time:

```
% make prefix=/odd/place install
```

This might break your installation if there was a prefix baked in to your component at build time. Do this only if you have a particular reason for doing so, and know what you're doing.

(2) At make time:

```
% make prefix=/odd/place
```

This builds your component with the Makefile variable `$prefix` set to the given value. The component will be installed into this location at install time, unless you override the location as described above. See the discussion of 'installation directory variables' in Sec. 2.1.2.

(3) At configure time:

```
% ./configure --prefix=/odd/place
```

This is usually the most appropriate way to do this.

(4) By setting a 'sticky variable':

```
% ./starconf.status STARCONF_DEFAULT_PREFIX=/odd/place
```

This sets the default for the `./configure` script's `-prefix` option.

(5) By configuring *starconf*:

```
% pwd
.../buildsupport/starconf
% ./configure STARCONF_DEFAULT_PREFIX=/odd/place
...
% make; make install
```

This sets the default for all directories configured using this *starconf* application. This is effectively what you do when you set the environment variable `STARCONF_DEFAULT_PREFIX` before you run the top-level `./bootstrap` script, as described in Sec. 3.2.

OK, now I've changed my mind about where I want components to be installed – the 'prefix'. How do I tell the build s

Easy. Set the `STARCONF_...` variables as you want them, as described in Sec. 3.2, and in the top level of the checkout, do

```
% ./bootstrap --buildsupport
% rm config.cache
```

The `-buildsupport` option forces the bootstrap to remake and reinstall the buildsupport components, taking the current values of the `STARCONF_...` variables into account.

You should also remove any `config.cache` files, both in the top-level directory and elsewhere, since these often store paths to applications and libraries which are probably not appropriate for your new prefix.

For more details, see the notes on state in Sec. 2.4.

What does this mean?: `required file './install-sh' not found` It means that you need to run *autoreconf* with different options.

You have just run *autoreconf* by hand (haven't you?) and it is complaining that it cannot find the `./install-sh` file which *automake* requires (in fact *autoreconf* has spotted a `Makefile.am` in your directory, and so has invoked *automake*, and it is this which is reporting the error). Give *autoreconf* the `-install -symlink` options, which installs required files, and does so by symlinking rather than copying them.

What does this mean?: `configure: error: '<env-var>' has changed since the previous run`
The message might say:

```
% ./configure -C
configure: loading cache config.cache
configure: error: 'STARLINK' has changed since the previous run:
configure:   former value:  /somewhere/else
configure:   current value: /export3/sun
configure: error: changes in the environment can compromise the build
configure: error: run 'make distclean' and/or 'rm config.cache' and start over
```

You have changed an 'influential environment variable'. See Sec. 2.1.5.

Chapter 6
Miscellaneous hints and tips

This section contains assorted techniques and patterns which don't fit neatly in anywhere else. Please do add to this section.

6.1 Forcing automake's choice of linking language

XXX no: there's an better mechanism, as yet undocumented. Watch this space, but don't use this technique.

When building a library or application, automake needs to make a decision about which language to use to do the final link. For example, it must use the Fortran linker to make a library which consists of or includes Fortran modules, and the C linker if it consists of pure C. This matters more on some platforms than others – the OSX linker has some very fixed ideas on this point (it heartily disapproves of common blocks, and so needs special Fortran juju to make a library from Fortran which uses them).

Automake makes this decision based on the source code which is listed in the library's `_SOURCES` variable. When you are assembling a library purely from convenience libraries, however (see section 'Linking static libraries' in the Libtool manual), the `_SOURCES` variable is empty, and automake ends up using the C linker, for want of any other information.

You can avoid this problem, and indicate to automake which langauge it should use, by giving a dummy value for the `_SOURCES` variable as follows:

```
libsubpar_adam_la_SOURCES = dummy_routine.f
libsubpar_adam_la_LIBADD = \
        subpar/libsubpar_convenience.la \
        parsecon/libparsecon_convenience.la
```

where the `dummy_fortran.f` module is a trivial Fortran routine:

```
C A dummy Fortran routine.  See comments in Makefile.am
      SUBROUTINE DUMMY
      END
```

As the comment indicates, it is wise to include an explanation of this ... let's not call it a hack ... in the `Makefile.am`.

This is currently used in the 'pcs', 'gks', 'astrom' and 'pgplot' components.

6.2 Conditionally building components, I

It is occasionally useful to build code conditionally, depending on features of the environment. About the only place where this is necessary, however, is when building a target of a configure dependency (see Sec. A.17), where a library (for example) must be built only if it is not available on the platform already.

There is no way to decide not to build a component, so instead we arrange the configuration of the component in such a way that if the component's code is not required, then the component trivially builds and installs nothing, or nothing more than a stamp file.

This is true for the `jpeg` component, which is a configure dependency of `startcl`, and located in the repository in `thirdparty/ijg/jpeg`. The important parts of `jpeg`'s `configure.ac` are as follows:

```
AC_CHECK_HEADER([jpeglib.h],
  [MAIN_TARGET=jpeglib-stamp],
  [
  (
    cd src
    echo ./configure --prefix=$prefix --cache-file=config.cache
    ./configure --prefix=$prefix --cache-file=config.cache
  )
  MAIN_TARGET=jpeglib
])
AC_SUBST(MAIN_TARGET)

STAR_SPECIAL_INSTALL_COMMAND([
    if test $(MAIN_TARGET) = jpeglib; then
      cd src;
      for d in bin lib include man/man1;
        do $(mkdir_p) $$DESTDIR$(prefix)/$$d;
      done;
      $(MAKE) DESTDIR=$$DESTDIR install install-lib;
    fi])
```

The first part wraps the actual jpeglib configuration code (see Sec. 4.5.3 for the rationale for that code) inside an autoconf test for the `jpeglib.h` include file. If it finds it, there is nothing to do; if it doesn't then it does actually configure the package. The decision it makes is recorded in the substituted variable `@MAIN_TARGET@` and its corresponding Makefile variable `$(MAIN_TARGET)`.

Note that the `STAR_SPECIAL_INSTALL_COMMAND` is expanded at a different time from the header test, with the effect that it is effective in both conditions, when the header file is there and when it isn't. Thus the command inside it must be written with this in mind, and explicitly test the value of `$(MAIN_TARGET)`.

The `Makefile.am` has corresponding support, the important parts of which are as follows.

```
all-local: $(MAIN_TARGET)

jpeglib:
        cd src; $(MAKE) all

jpeglib-stamp:
        rm -f $@
        { date; \
          echo "jpeglib.h found in system -- no need to build our own"; } >$@
```

We can't override the 'all' target, but instead use automake's 'all-local' hook which, if it is present, is built at the same time as 'all'. Thus depending on the value of the variable `$(MAIN_TARGET)`, as determined and substituted by `configure.ac`, we build either the jpeglib software, or a stamp file. The install command in the `configure.ac` file above then avoids installing anything if `$(MAIN_TARGET)` is not the string `jpeglib`.

6.3 Conditionally Building Components, II

The other way in which you might conditionally build a component is to suppress building it at the end of the configuration stage, if it turns out that the important features are missing from the environment.

For a description of this, see Sec. A.30.

6.4 Manipulating compiler and linker flags

Sometimes it is necessary to add compiler or linker flags for particular make targets, or add libraries to a link. This is described in the automake manual, in the section Program and Library Variables. This is a powerful mechanism, but there is a gotcha.

For each of the variables `CFLAGS`, `LDFLAGS` and so on, automake maintains a parallel variable `AM_CFLAGS` or `AM_LDFLAGS`, the contents of which are managed by automake and autoconf, so that the unprefixed variables are free to be overridden by the user (see section Variables reserved for the user of the automake manual; to be clear, 'user' in this context means the person building the distributed software, as opposed to the maintainer, or whoever is creating the distribution). If automake sees one of the above per-target variables, however, it carefully uses this *instead* of the `AM_` variable, not in addition to it. This replacement is sensible, since if this were not the case then there would be no way to remove a problematic flag from an `AM_` variable, but it is not the behaviour you might guess at first, and so can come as rather a surprise. Given that you want to add to these flags rather than replace them, you do so with the idiom:

```
libxxx_la_FCFLAGS = $(AM_FCFLAGS) -my-magic-flag
```

An alternative to setting per-target flags like this is to set one or more of the `AM_*FLAGS` variables directly. These variables are 'yours' as the author of the `Makefile.am` and `configure.ac` files – the automake system ensures that the variables are included in the correct places in compile and link commands, but neither it nor autoconf set these flags. See the FAQ on 'Flag Variables Ordering' in the automake manual for some further discussion.

There is another set of similar variables, named `STAR_*FLAGS`, which are maintained by `starconf` and the `STAR_*` macros in `configure.ac`, and which are the means by which those macros pass their discovered magic on to the Makefile. You shouldn't need to know that, and you should not need to adjust these variables at all, but if for some occult reason you need to refer to the complete set of flags used to compile and link programs, you will need to include these flags also. In general, Makefile variables such as `$(COMPILE)` and `$(FCCOMPILE)` are most suitable for any by-hand compilations you need to do, and they already have all the extra flags you need, in the right order..

Appendix A
The Starconf macros and variables

The following macros may be used within the `configure.ac` file. Of these, only the `STAR_DEFAULTS` macro is required to be in the file.

In the descriptions below, optional arguments are shown in square brackets, along with their default value. If the sole argument is optional (for example in the case of `STAR_DEFAULTS`), you do not need to supply the brackets.

The macros below ultimately do their work by expanding into shell script, and these results are exposed in the `./configure` file. Although it can occasionally be useful to examine these implementation details, resist the temptation to write your `configure.ac` scripts based on what you find there. If macro behaviour is not documented here, you should not rely on it. If you *need* to rely on it, then consult with the starconf maintainer to have the documentation changed, and the behaviour thus fixed.

A.1 AC_FC_CHECK_HEADERS

`AC_FC_CHECK_HEADERS(include-file...)`

Fortran analogue of `AC_CHECK_HEADERS`, though it only takes the first argument, giving the list of include files to check. For each include file, defines `HAVE_`*include-file* (in all capitals) if the include file is found. Respects the current value of `FCFLAGS`.

A.2 AC_FC_CHECK_INTRINSICS

AC_FC_CHECK_INTRINSICS (function...)

Like `AC_CHECK_FUNCS`, but instead determine the intrinsics available to the Fortran compiler. For each intrinsic in the (whitespace-separated and case-insensitive) argument list, define `HAVE_INTRINSIC_intrinsic` if it is available. For example, `AC_FC_CHECK_INTRINSICS(sin)` would define `HAVE_INTRINSIC_SIN` if the 'sin' intrinsic function were available (there are probably rather few Fortrans which don't have this function).

There are some functions, such as `getcwd`, `getenv` and `chdir`, which are provided as intrinsics by some Fortran compilers, but which others supply as external routines in the Fortran runtime. To deal with this (exasperating) situation, you need a construction like this in your `configure.ac`:

```
AC_FC_CHECK_INTRINSICS([getcwd])
AC_LANG_PUSH([Fortran])
AC_CHECK_FUNCS([getcwd])
AC_LANG_POP([Fortran])
```

and then conditionalise on the defines `HAVE_GETCWD` and `HAVE_INTRINSIC_GETCWD` in your preprocessable code. That is, you use the standard macro `AC_CHECK_FUNCS`, but wrapped in declarations which force the check to use Fortran (this doesn't work in standard autoconf, since the required support is missing).

A.3 AC_FC_HAVE_BOZ

`AC_FC_HAVE_BOZ`

Test whether the `FC` compiler supports BOZ constants in the Fortran 95 style. These are integer constants written in the format `B'xxx'`, `O'xxx'` and `Z'xxx'`. If so set the preprocessor variable `HAVE_BOZ` to be 1. This should be true of all Fortran 95, and later, compilers.

See also `AC_FC_HAVE_TYPELESS_BOZ` and `AC_FC_HAVE_OLD_TYPELESS_BOZ`.

A.4 AC_FC_HAVE_OLD_TYPELESS_BOZ

`AC_FC_HAVE_OLD_TYPELESS_BOZ`

Test whether the Fortran compiler (`FC`) supports typeless BOZ constants in the OLD (VMS and g77) Fortran style. These are constants written in the format `'xxx'X`, which allow the initialisation of any type to a specific bit pattern. If so set the preprocessor variable `HAVE_OLD_TYPELESS_BOZ` to be 1.

See also `AC_FC_HAVE_BOZ` and `AC_FC_HAVE_TYPELESS_BOZ`

A.5 AC_FC_HAVE_PERCENTLOC

`AC_FC_HAVE_PERCENTLOC`

Test whether the `FC` compiler has the `%LOC` extension. If so, define the preprocessor variable `HAVE_PERCENTLOC` to be 1.

A.6 AC_FC_HAVE_PERCENTVAL

`AC_FC_HAVE_PERCENTVAL`

Test whether the `FC` compiler has the `%VAL` extension. If so, define the preprocessor variable `HAVE_PERCENTVAL` to be 1.

A.7 AC_FC_HAVE_TYPELESS_BOZ

`AC_FC_HAVE_TYPELESS_BOZ`

Test whether the Fortran compiler (`FC`) supports typeless BOZ constants in the Fortran 95 style. These are constants written in the format `X'xxx'` that allow the initialisation of any type of variable to a specific bit pattern. If so set the preprocessor variable `HAVE_TYPELESS_BOZ` to be 1.

See also `AC_FC_HAVE_BOZ` and `AC_FC_HAVE_OLD_TYPELESS_BOZ`

A.8 AC_FC_HAVE_VOLATILE

`AC_FC_HAVE_VOLATILE`

Test whether the Fortran compiler (`FC`) supports the VOLATILE statement. VOLATILE is used to stop the optimisation of a variable, so that it can be modified outside of the program itself. If supported the preprocessor variable `HAVE_VOLATILE` is set to be 1.

A.9 AC_FC_LITERAL_BACKSLASH

AC_FC_LITERAL_BACKSLASH

Check whether the compiler regards the backslash character as an escape character. The backslash is not in the Fortran character set, so ISO-1539 does not specify how this is to be interpreted, but many Unix Fortran compilers interpret `'\n'`, for example, as a newline, and `'
'` as a single backslash. Many Unix compilers have switches which allow you to choose between these two modes.

This macro tests the behaviour of the currently selected compiler, and defines `FC_LITERAL_BACKSLASH` to 1 if backslashes are treated literally – that is if `'\\'` is interpreted as a *pair* of backslashes and thus that `'\n'` is interpreted as a pair of characters rather than a newline.

Note that, if you simply want to have a character constant which contains a single backslash, then the following initialisation will do that for you without any need for this macro (thanks to Peter Draper).

```
*  Printable backslash: some compilers need '\\' to get '\', which
*  isn't a problem as Fortran will truncate the string '\\' to '\'
*  on the occasions when that isn't needed.
      CHARACTER * ( 1 ) CCD1__BKSLH
      PARAMETER ( CCD1__BKSLH = '\\' )
```

Fortran is required to truncate the initialising string without error, as noted in ISO-1539 sections 5.2.9 and 7.5.1.

A.10 AC_FC_OPEN_SPECIFIERS

AC_FC_OPEN_SPECIFIERS(specifier ...)

The Fortran OPEN statement is a rich source of portability problems, since there are numerous common extensions which consiste of extra specifiers, several of which are useful when they are available. For each of the specifiers in the (whitespace-separated) argument list, define `HAVE_FC_OPEN_`*mungedspecifier* if the specifier may be given as argument to the OPEN statement. The *mungedspecifier* is the *specifier* converted to uppercase and with all characters outside `[a-zA-Z0-9_]` deleted. Note that this may include 'specifiers' such as `access='append'` and `[access='sequential',recl=1]` (note quoting of comma) to check combinations of specifiers. You may not include a space in the 'specifier', even quoted. Each argument must be a maximum of 65 characters in length (to abide by Fortran 77 line-length limits).

See Sec. 2.5 for an example of using this preprocessor definition in a Fortran program.

A.11 AC_FC_RECL_UNIT

AC_FC_RECL_UNIT

When opening a file for direct access, you must specify the record length with the `OPEN` specifier `RECL`; however in the case of unformatted direct access files, the *units* of this specifier are processor dependent, and may be words or bytes. This macro determines the units and defines `FC_RECL_UNIT` to contain the number of bytes (1, 2, 4, 8, ...) in the processor's unit of measurement.

Note that unformatted files are not themselves portable, and should only be used as either temporary files, or as data files which will be read by a program or library compiled with the same Fortran processor. With this macro, however, you can read and write such files in a portable way.

A.12 AC_PROG_FC

`AC_PROG_FC([COMPILERS...], [DIALECT])`

`[COMPILERS...]` is a list of specific compilers you want it to try, so you could give it `[f77 g77]` as an argument. That's useful if you have rather specific demands or need to more-or-less customize this.

More useful is the `DIALECT`, which can be 77, 90, 95 or 2000. Thus if you give this as `AC_PROG_FC([], 77)` or `AC_PROG_FC([], 1977)` then this will search only its internal list of Fortran 77 compilers (which happens to be [g77 f77 xlf frt pgf77 fort77 fl32 af77] currently).

A.13 AC_PROG_FPP

`AC_PROG_FPP(required-features,[FPP-SRC-EXT=F])`

`required-features` is a space-separated list of features that the Fortran preprocessor must have for the code to compile. It is up to the package maintainer to properly set these requirements.

`FPP-SRC-EXT` is an optional specification of the file extension for preprocessable Fortran source files (without a leading dot). It defaults to `F`: There would appear to be a problem, here, with case-insensitive filesystems, such as the default HFS+ on MacOS X, and Windows filesystems, since the extension `.F` is indistinguishable from plain `.f`. However there is no problem in practice, since the Fortran compilers you are likely to encounter on these platforms are all capable of compiling preprocessable Fortran directly to object code, without requiring the intermediate `.f` file (and there is a check in autoconf to double-check that this is true).

The macro works out if the Fortran compiler discovered by macro `AC_PROG_FC` has the requested set of features. If so, it arranges for the compilation to be done 'directly'; if not, it arranges for 'indirect' compilation, where the preprocessable Fortran code is converted to pure Fortran code and only subsequently passed to the Fortran compiler.

The configure variables which this macro sets, and which are available for substitution, are as follows.

`FPP` The Fortran preprocessor which the macro decides is suitable.

`FPPFLAGS` The flags which will be given to the compiler (in direct mode) or preprocessor (in indirect mode).

`FPP_SRC_EXT` This is the file extension (without a dot) of preprocessable Fortran files; by default this is 'F'.

`FPP_COMPILE_EXT` This contains the file extension which the Fortran compiler will accept. It is the same as `FPP_SRC_EXT` if the compiler itself can do preprocessing ('direct' compilation), or a dummy value (not .f, since that would conflict with the pre-existing rule for compiling Fortran) if it cannot, and the file must be preprocessed separately ('indirect').

`FPP_PREPROCESS_EXT` The partner of FPP_COMPILE_EXT. This is `FPP_SRC_EXT` for indirect compilation, or a dummy value for direct compilation, when the corresponding separate-processing generated rule should be ignored.

`FPP_MAKE_FLAGS` This is used to pass flags to the `cpp` or `fpp` command if we compile directly, and leave them out otherwise.

`FPP_OUTPUT` This is used to redirect FPP output to the .f file in those cases where FPP writes to stdout rather than to a file: it is either blank or something like `>$@`

You do not generally need to know or care about the above variables, since if you have any preprocessable source files in your source set, then automake will insert appropriate build rules in the generated `Makefile.in` on your behalf. If, however, you need to run the preprocessor 'by hand' for some reason (perhaps because you are using the preprocessor to do something other than compiling `.F` to `.o`, so that the `.F` file is not being mentioned in a `_SOURCES` primary), then a suitable stanza in a `Makefile.am` or `Makefile.in` would be:

```
file.f: file.$(FPP_SRC_EXT)
        $(FPP) $(FPPFLAGS) $(CPPFLAGS) file.$(FPP_SRC_EXT) $(FPP_OUTPUT)
```

where we have ensured that the input file has the extension `$(FPP_SRC_EXT)`. You can add extra options such as `-I.` if necessary (automake puts these in the variable `$(DEFAULT_INCLUDES)`).

NOTE: There are some features of this macro support which are arguably bad autoconf style, and it is *very* likely that this macro will change. The documentation below is only for completeness, and you are advised not to pay any attention to it. See Sec. C.

Supported features are:

include the preprocessor must correctly process `#include` directives and `-I` options

define correctly process `-D` options

substitute substitute macros in Fortran code (some preprocessors touch only lines starting with #)

wrap wrap lines that become too long through macro substitution. `fpp` is probably the only preprocessor that does this.

cstyle Do not suppress C style comments (the `-C` option in cpp)

CSTYLE *Do* suppress C style comments (e.g. code contains C-style comments, and compiler may not know how to handle them)

Features can be abbreviated: i, in, inc etc. are equivalent to include. Features can be deselected (feature not needed) by prepending "no", e.g. nodef (=nodefine), now (=nowrap).

Default for the feature list is [include define substitute nowrap nocstyle noCSTYLE]

Feature requirements corresponding to the defaults may be omitted

Note that "wrap" implies "substitute", and CSTYLE and cstyle cannot be requested at the same time. The macro adjusts this automatically.

A.14 STAR_CHECK_PROGS

`STAR_CHECK_PROGS(progs-to-check-for, [package=''])`

Note: The functionality of this macro is very likely to change.

For each of the programs in `PROGS-TO-CHECK-FOR`, define a variable whose name is the upcased version of the program name, and whose value is the full path to that program, or the expected installation location of that program if no absolute path can be found. Because of this default behaviour, this macro should *only* be used for locating Starlink programs such as messgen or alink, and not as a general replacement for `AC_CHECK_PROG`. Any characters in the program outside of the set of alphanumerics and underscores are normalised to underscores.

The optional second argument gives the name of the package containing the program in question. Some packages install their binaries in package-specific directories, and this argument allows this macro to look there as well.

For example: `STAR_CHECK_PROGS(messgen)` would define the variable `MESSGEN` to have the full path to the messgen application.

Calls AC_SUBST and AC_ARG_VAR on the generated variable name.

This is the analogue of `AC_CHECK_PROG`, except that: (1) the variable name defaults to the program name, (2) if the program is not found, the variable value is the program's name without any path at all.

The current value of the `PATH` variable is augmented by the location of the binary installation directory, using the current default value of the prefix (not ideal, since this may in principle change when the component being configured is installed, but it's the best we can do at configure time); and by the `$STARLINK/bin`, `$STARLINK/Perl/bin` and `$STARLINK/starjava/bin` directories (so this is a suitable macro to use to locate the distributed Perl binary).

If the program in question cannot be found, the the substitution variable is set to be the name of the program without any path, in the expectation or hope that the program will eventually end up in the path. This is not ideal, but it is the best that can be done in the important case where the program is being configured as part of bootstrapping build, and thus before any packages have been built and installed, so that there is in fact no installed program to be found. This heuristic will obviously fail when the program to be checked for is not in the path, and this will typically happen when the second 'package' argument is non-null. The pattern to use in this case is as follows.

Some components build their documentation using the (now deprecated) `sst` component. This uses the `per-package-dirs` option, and so installs its applications in a package-specific directory. You would therefore check for the `prolat` tool (for example) in `configure.ac` using `STAR_CHECK_PROGS(prolat,sst)`. If the `sst` component is in fact installed, this would substitute `@PROLAT@` with the full path to the application; but in a bootstrap build could do no more than subsitute it with the bare program name `prolat`, as described above. In order to use this within a `Makefile.am`, you would have to use a rule like

```
target: dependency
        PATH=$$PATH:$(STARLINK)/bin/sst @PROLAT@ args...
```

where the `PATH` is temporarily augmented with the installation location of the `sst` component. This is pretty safe, because the `STARLINK` Makefile variable is always present and always substituted. Despite that, it is not ideal, since it depends on undocumented (but unlikely to change in practice) knowledge of how the `sst` component is installed, and it is possible to think of convoluted installation schemes which might trip this up. Given that, the only time this is likely to matter is during a bootstrap build (at other times, `prolat` would be installed already and so `@PROLAT@` would be substituted with a full path), so it is unlikely to be a problem in practice.

A.15 STAR_CNF_BLANK_COMMON

`STAR_CNF_BLANK_COMMON`

Determine the global symbol used for the Fortran blank common storage space. This is a specialist macro probably only of use to the `CNF` package. Its only side-effect is to define the C macro `BLANK_COMMON_SYMBOL`.

A.16 STAR_CNF_COMPATIBLE_SYMBOLS

`STAR_CNF_COMPATIBLE_SYMBOLS`

Work out what is required to have the Fortran compiler produce library symbols which are compatible with those expected by the CNF package. If you are building a library which includes Fortran code, then you should call this macro, which possibly adjusts the `FCFLAGS` variable.

At present, all this macro has to do is simply work out whether it needs to stop g77 adding a second underscore to generated symbol names (it adds a single underscore to most Fortran symbols, but by default adds two when the symbol name already contains an underscore); the other Fortran compilers we use don't need any extra options, as it happens. However this could potentially be much more complicated. The autoconf `AC_F77_WRAPPERS` macro detects more possibilities, but probably not a completely exhaustive set. In future it might be necessary to extend the CNF macros, by somehow merging the results of `AC_F77_WRAPPERS` into it, and at that point it might be necessary to extend this macro.

This macro is designed to work with CNF, however it does *not* require the `cnf.h` headers to be installed, because it should remain callable at configuration time before *anything* has been installed. Instead we fake the functionality of the definition `F77_EXTERNAL_NAME` in `cnf.h`, which appends an underscore (just one) to the end of C symbols.

A.17 STAR_DECLARE_DEPENDENCIES

`STAR_DECLARE_DEPENDENCIES(type, deplist, [option=''])`

Declare dependencies of this component. The `TYPE` is one of 'sourceset', 'build', 'link', 'configure', 'use' or 'test', and the `DEPLIST` is a space separated list of component names, which this component depends on in the specified way.

This is less complicated than it seems. The only types you're likely to have to specify are 'build' and 'link' dependencies.

- If a particular component *D* needs to be installed in order for a given component *C* to be built (perhaps because it uses include files installed by the component *D*), then that second component has a *build* dependency on *D*: *C*'s `configure.ac` should contain `STAR_DECLARE_DEPENDENCIES(build,D)`.
- If *C* is a library which depends on functions from another library *D*, then *D* does not have to be present in order to build *C*, but it does have to be present, at some later stage, in order to link against it: `STAR_DECLARE_DEPENDENCIES(link,D)`
- If *D* is a library which we need to link against in order to build our component *C* (which is almost certainly an application), then this is still a build dependency on *D*, but we must indicate that we depend on all of *D*'s link dependencies also: `STAR_DECLARE_DEPENDENCIES(build,D,link)`.

In full, the types of dependencies are as follows:

Sourceset dependencies These are the components which must be installed in order to build the complete set of sources, either for building or for distribution. This includes building documentation, so it would include `star2html` as well as `messgen`. These dependencies are generally added for you, by the `STAR_LATEX_DOCUMENTATION` or `STAR_MESSGEN` macros.

Build dependencies These are the dependencies which are required in order to build this component. This is typically because you require include files installed by the other component, but it might also be because you require an executable built by it (such as the `compifl` tool within the `parsecon` component). See also the discussion of 'option', below. You may not have two components which have a build dependency on each other, since that would mean that each would have to be built before the other, which is impossible.

Link dependencies These are the dependencies required to link against the libraries in a component. That means all the libraries that this component's libraries use. These are not necessarily build dependencies, since if you are building a library *X*, which uses functions in library *Y*, the library *Y* does not have to be present in order to *build* library *X*, only when you subsequently want to actually link against it. Thus you can have two components which have mutual link dependencies. If you are building an application, however, all its link dependencies will actually be build dependencies and should be declared as such. In other words, the distinction between build and link dependencies is important only for library components.

Configure dependencies There are some components which must be installed *before* the configure step, since they must be present in order for other packages to configure against them. Components which are listed as a configure dependency of some other component are made during the `make configure-deps` stage of the bootstrapping build process.

You should try hard to avoid declaring dependencies of this type, since they necessarily subvert the normal build order. Note that transitivity implies that any dependencies of a configure dependency are also configure dependencies, so that declaring a configure dependency on sst for example (an obvious but bad plan), results in half the software set being built before configuration. Which is crazy.

You can see the set of configure dependencies by looking at the `configure-deps` target in `Makefile.dependencies`.

Use dependencies These are the dependencies required in order for the component to be used by something else, after it has been built and installed. For example a library which called another application as part of its functionality would have only a use dependency on the component which contained that application. If no use dependencies are declared, we take the use dependencies to be the same as the link dependencies. We expect that such use dependencies will become more important when we start using this dependency information to build distribution packages, or to assemble `.rpm` or `.deb` packages. You should add any use dependencies you are aware of, even though we cannot really debug them until a distribution system is in place.

Test dependencies These are required in order to run any regression tests which come with the component. It's generally a good idea to avoid making this a larger set than the use dependencies, but sometimes this is unavoidable. If no test dependencies are declared, we take the test dependencies to be the same as the use dependencies. Note that the system does not currently make any use of this information.

The point of all this is that different dependencies are required at different times. The set of dependencies in the master makefile is composed of all the 'sourceset' and 'build' dependencies, but not 'link' or 'use' dependencies, and since the core Starlink libraries are closely interdependent, the set of 'build' dependencies needs to be kept as small as possible in order to avoid circularities (that is, A depending on B, which depends, possibly indirectly, on A).

It is because of this intricacy that the dependencies scheme is so complicated. *In general, it is safe to declare most things to have build dependencies*, and only refine these into link or use dependencies when things won't work otherwise.

All these relationships are transitive: if A has a build dependency on B, and B has one on C, then A has a build dependency on C. You can augment this by using the final 'option' argument: if, in component A's `configure.ac`, you say `STAR_DECLARE_DEPENDENCIES(build, B, link)`, then you declare that A has a build-time dependency on B, but that (presumably because B is a component consisting entirely or mostly of libraries) you need to link against B, so component A has a dependency on all of B's *link* dependencies, not just its build dependencies. An application component should generally declare such a dependency for each of the libraries it depends on. This is (I believe) the only case where this 'option' attribute is useful, though it is legal for each of the dependency types.

Because of the transitivity, you need only declare direct dependencies. If package A depends on package B, which depends in turn on package C, then package A need not declare a dependency on C.

The macro may be called more than once. The results of this macro are expressed in the file `component.xml` in the component directory.

A.18 STAR_DEFAULTS

`STAR_DEFAULTS([options=''])`

Sets up the defaults for Starlink `configure.ac` files. The optional `OPTIONS` argument holds a space-separated list of option keywords, of which the only ones defined at present are `per-package-dirs` and `docs-only`.

As part of its initialisation, this macro does the following:

- Makes sure that the variable `STARLINK` is defined, so that you can use it elsewhere in your `configure.ac` file. The value of this variable depends on the `STARCONF_DEFAULT_STARLINK` variable which is defined by the `./starconf.status` script (see Sec. 2.2), and can be overridden by a setting of `STARLINK` in the environment, though you are discouraged from doing that.
- Sets appropriate vales for variables `AM_FCFLAGS` and `AM_LDFLAGS` (you shouldn't have to care, as the point of these prefixed variables is that they should be orthogonal to the unprefixed variables with the similar names, but see Sec. 6.4 for a gotcha).
- Defines and substitutes variables `PACKAGE_VERSION_MAJOR`, `..._MINOR`, `..._RELEASE` and `..._INTEGER`. These are respectively the major, minor and release numbers extracted from `PACKAGE_VERSION`, and an integer representation of these consisting of `major * 1000000 + minor * 1000 + release`.

This macro also sets the `prefix` shell variable to its default value. Standard autoconf `./configure` scripts also do this, but at a slightly later stage, which prevents the variable being used safely within `configure.ac` scripts. This is useful if you have to configure a thirdparty package in a subdirectory, by calling its `./configure` script with a suitable `-prefix` option.

The option `per-package-dirs`, has three consequences:

(1) applications, help and etc files are installed in a package-specific directory (however the `fac_*_err` files are regarded as being in a separate category, and still install in `/star/help`, without any package-specific prefix);

(2) the otherwise-forbidden variable `bin_DATA` is permitted: any files assigned to this variable are installed in the (package-specific) binary directory as data, as opposed to executables;

(3) a file `version.dat` is created (at make rather than install time). This holds the bare `PACKAGE_VERSION` number, and is installed in the binary directory.

See also Sec. C.

The `docs-dir` option declares that this component contains only documentation (or at least, no code). This triggers mild changes in the behaviour of the `STAR_DEFAULTS` macro which should make configuration somewhat faster in this case. Certain other macros, such as `STAR_MESSGEN`, are disabled.

A.19 STAR_INITIALISE_FORTRAN_RTL

`STAR_INITIALISE_FORTRAN_RTL`

Define a macro which can be used in a C or C++ main program to initialise the Fortran RTL, including, for example, doing whatever work is required so that the Fortran `getarg()` function works. This defines the macro `STAR_INITIALISE_FORTRAN(argc,argv)`. If nothing is required or possible on a particular platform, then this expands, harmlessly, to nothing.

You might use this as follows:

```
#include <config.h>

int main(int argc, char** argv)
{
    STAR_INITIALISE_FORTRAN(argc, argv);

    /* etc */
}
```

A.20 STAR_LARGEFILE_SUPPORT

`STAR_LARGEFILE_SUPPORT`

Set macros for compiling C routines that want to make use of large file support. This is a joining of `AC_SYS_LARGEFILE` and `AC_FUNC_FSEEKO` so defines the macros `_FILE_OFFSET_BITS`, `_LARGEFILE_SOURCE` and `_LARGE_FILES` as needed , along with `HAVE_FSEEKO` when `fseeko` and `ftello` are available.

To use large file support you need to use fseeko and ftello, instead of fseek and ftell, when `HAVE_FSEEKO` is defined (and use type `off_t` for offsets) and compile all C code with the other defines (many of the usual system calls are replaced with large file supporting ones, when this is done).

Currently the `AC_FUNC_FSEEKO` macro does not define `_LARGEFILE_SOURCE` under Solaris, so this function makes sure that this is done. It also gathers the values of `_FILE_OFFSET_BITS`, `_LARGEFILE_SOURCE` and `_LARGE_FILES` and sets the `STAR_LARGEFILE_CFLAGS` variable so that it can be used as part of CFLAGS when compiling. This in useful for passing to packages which are not directly part of the starconf system. Note that normally the necessary defines are written to `config.h` or are already made part of the local `CFLAGS`, so this variable is not needed.

A.21 STAR_LATEX_DOCUMENTATION

`STAR_LATEX_DOCUMENTATION(documentcode, [targets=''])`

Generate the standard makefile targets to handle LaTeX documentation source. The parameter `documentcode` should be a space-separated list of codes like 'sun123' – they must not include any `.tex` extension. Each of the documentcodes may optionally be followed by a slash, to indicate that the source file is not in the current directory; see below.

The second, optional, argument gives an explicit list of the targets which are build. If this is *not* specified, then a standard list is used (`.tex`, `.ps` and `.tar_htx` at present, though this is in principle subject to change) and corresponding rules added to the generated makefile. If it is specified, it must be non-null, and its value is a list of files which are to be added to the distribution, and no extra Makefile rules are added. Thus if users need anything complicated done, they should use this second argument and provide rules for satisfying the given targets.

In the latter case, the `.tex -> htx_tar` rule is still emitted, so you can use it, but it requires the substitution variable `@STAR2HTML@`, and so if you *do* use it, you will have to make that available, either through `STAR_CHECK_PROGS(star2html)` or otherwise (see Sec. A.14). If you need to tune the parameters used when running the `star2html` command, then assign these to the `STAR2HTML_FLAGS` macro in `Makefile.am`.

The default rule for compiling LaTeX files simply runs LaTeX twice. If you need to do more than this – for example to run BibTeX or make an index – then you can override this by setting the variable `LATEX2DVI` in the `configure.ac` file; it is already substituted with a default value, so you don't have to call `AC_SUBST` on it. This variable should refer to LaTeX as `$$LATEX`, and use `$$1` to refer to the root of the LaTeX file to be compiled (that is, the filename with any `.tex` extension removed). The doubled dollar signs are because this is evaluated within the context of a 'make' rule. Thus if you needed to run BibTeX between the LaTeX runs, you would define:

```
LATEX2DVI='$$LATEX $$1; bibtex $$1; $$LATEX $$1'
```

This macro defines the substitution variable `@STAR_LATEX_DOCUMENTATION@` to be the second macro argument if that is present, or the default target list if not. This variable must therefore be used as a value for the `stardocs_DATA` Makefile variable (see Sec. A.32).

If any `documentcode` is followed by a slash, then the default target list is assigned to `@STAR_LATEX_DOCUMENTATION_<DOCUMEN` instead; you may not use this latter feature if the second argument is non-null.

See Sec. 4.6 for more discussion of this feature, plus an example.

The configure option `-without-stardocs` suppresses building documentation.

It is occasionally necessary to add extra files to the `.htx` tarball – for example `.gif` images, or other files referenced by the HTML version of the documentation. To request that extra files should be added to the `starnnn.htx_tar` archive, you should create a file `starnnn.htx_tar.extras` listing the filenames which should be included, separated by whitespace (spaces, tabs or newlines).

A.22 Variable STAR_MANIFEST_DIR

The `STAR_MANIFEST_DIR` substitution variable is slightly special. If it is defined in your `configure.ac` file, then Starlink automake will include in the generated `Makefile.in` code to create and install a manifest of all the files installed.

If you set this variable at installation time:

```
make STAR_MANIFEST_DIR=/my/special/manifest install
```

then this overrides the manifest location defaulted at configure time. If the variable is set to be null (`make STAR_MANIFEST_DIR= install`) then you will suppress the creation and installation of the manifest.

You define it as follows:

```
: ${STAR_MANIFEST_DIR='$(prefix)/manifests'}
AC_SUBST(STAR_MANIFEST_DIR)
```

This sets the variable to be `$(prefix)/manifests`, but allows it to be overridden by a setting of the variable in the environment. Note that is is set to be relative to the value of the `$(prefix)` makefile variable.

You do *not* have to set this in general, as this is one of the things that the `STAR_DEFAULTS` macro (Sec. A.18) takes care of. The only time you need to use this feature is when you are importing a third-party code set into the tree (as described in Sec. 4.5). In that case, you do not want to specify `STAR_DEFAULTS` in the `configure.ac`, but you do need to install a manifest. If there is more than one `configure.ac` file in the tree, you will have to add this for each one.

See also Sec. A.29.

A.23 STAR_MESSGEN

`STAR_MESSGEN([msgfile-list])`

Handle generating message, error, and facility files.

Declare that we will need to use the messgen utility. This macro does not by itself cause the messgen rules to be included in the makefile – that is done by automake, when it sees a `include_MESSAGES` variable.

In the `Makefile.am` you declare the `include_MESSAGES` or `noinst_MESSAGES` variable to have as value each of the error files you require from the `.msg` file. For example, you might write:

```
include_MESSAGES = ast_err.h AST_ERR
```

to generate, install and distribute the C and Fortran error message files. You may include a `_SOURCES` declaration to indicate the name of the `.msg` file, but the default value (`ast_err.msg` in this case) is usually correct. The files thus listed are installed in the same include directory as other header files. You should not list a file in both `include_MESSAGES` and `include_HEADER`.

If you need to work with an error file which is not public, you can declare it with

```
noinst_MESSAGES = dat_err.h
```

Such a file is generated but not installed. In order that it be distributed, you should include `$(noinst_MESSAGES)` in the `_SOURCES` declaration of the program which includes it.

Although it's difficult to see why you'd want to, you can include the `nodist_` prefix with this primary.

You do not need to mention the `fac_xxx_err` file which messgen also generates – this is generated and installed automatically.

The optional argument is a space-separated list of files, each of which has a set of message declarations in it, in the format prescribed by the messgen utility. If this is present, then the named files are declared as pre-distribution files (the macro calls STAR_PREDIST_SOURCES on them), and so the resulting configure script should expect not to find them in an unpacked distribution. This is useful as documentation or as a shortcut for calling the latter macro, but recall that it is the presence of the automake `include_MESSAGES` variable which does the work.

The macro may be called more than once if you have more than one `.msg` file in the directory.

The files which this generates should be declared as sources in the Makefile.am if they are to be used in code, or are to be installed. For example:

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c $(include_MESSAGES)

include_MESSAGES = foo_err.h

BUILT_SOURCES = $(include_MESSAGES)
```

If they are used in building the package, you will probably need to declare them additionally as `BUILT_SOURCES`.

The macro also implicitly declares a 'sourceset' dependency on the messgen package.

A.24 STAR_MONOLITHS

`STAR_MONOLITHS`

Declare that we will be creating monoliths. This is conceptually similar to the various `AC_PROG_FC` declarations, and does whatever configuration is necessary to handle these.

Note that the declarations done in the `Makefile.am`, declaring the name of the monolith and the names and source files of the tasks, are slightly redundant inasmuch as some of that information could be implied. However, this is required to be explicit for clarity and consistency, and so accomodate the (currently unexploited) possibility that the tasks and `.ifl` files longer have the one-task-per-file relationship they have now.

See Sec. 3.4 for discussion.

A.25 STAR_PATH_TCLTK

`STAR_PATH_TCLTK([minversion=0], [options=''])`

Finds a tclsh, and the associated libraries, optionally searching for Tk as well.

Sets substitution variable `TCL_CFLAGS` to the C compiler flags necessary to compile with Tcl, `TCL_LIBS` to the required library flags, and variable `TCLSH` to the full path of the matching `tclsh` executable and `TCL_VERSION` to the `major.minor` version number; if Tk is requested, it similarly sets `TK_CFLAGS`, `TK_LIBS` and `WISH`. Define the cpp variable `TCL_MISSING` to 1 if the required programs and libraries are not available (that is, it is also set if Tcl is available but a suitable version of Tk isn't).

If argument `minversion` is present, it specifies the minimum Tcl/Tk version number required.

If the argument `options` is present, it is a space-separated list of the words `tk` and `itcl`. If one or both of these is present, then the macro will find a Tcl location which also has Tk or itcl installed (note that the itcl test doesn't do anything at present).

The macro searches first in the path, and then in a selection of platform-specific standard locations. The configure option `-with-tcl` allows you to provide a path to a `tclsh` binary, which is put at the head of the list of locations to search. Option `-without-tcl` suppresses the search, and results in no variables being substituted.

This is intended to be similar to macro `AC_PATH_XTRA`.

A.26 STAR_PLATFORM_SOURCES

`STAR_PLATFORM_SOURCES(target-file-list, platform-list)`

Generate the given target-file for each of the files in the list `TARGET-FILE-LIST`, by selecting the appropriate element of the `PLATFORM-LIST` based on the value of `AC_CANONICAL_BUILD`. Both lists are space-separated lists.

For each of the platforms, `<p>`, in platform-list, there should be a file `<target-file><p>`. There should always be a file `<target-file>default`, and if none of the platform-list strings matches, this is the file which is used. If the 'default' file is listed in the 'platform-list', then it is matched in the normal run of things; if it is not listed, it still matches, but a warning is issued.

If you wish no match *not* to be an error – perhaps because there is a platform-dependent file which is redundant on unlisted platforms – then end the platform-list with `NONE`. In this case, if no file matches, then no link is made, with no error or warning.

This macro uses the results of ./config.guess to determine the current platform. That returns a triple consisting of cpu-vendor-os, such as '`i686-pc-linux-gnu`' (OS=linux-gnu), '`sparc-sun-solaris2.9`', or '`alphaev6-dec-osf5.1`'.

The extensions `<p>` in platform-list should all have the form '`cpu_vendor[_os]`', where each of the components 'cpu', 'vendor' and 'os' may be blank. If not blank, they are matched as a prefix of the corresponding part of the config.guess value. Thus '`_sun_solaris`' would match '`sparc-sun-solaris2.9`' but not '`sparc-sun-sunos`', and '`_sun`' would match both. For a `<target-file>` file foo.c, this would result in '`ln -s foo.c_sun foo.c`'.

Calls `AC_LIBSOURCE` for each of the implied platform-specific files.

A.27 STAR_PREDIST_SOURCES

`STAR_PREDIST_SOURCES(source-files)`

Give a (space-separated) list of files which should exist only in the pre-distribution (ie, repository checkout) state. If one of these is found, then the substitution variable `PREDIST` is set to a blank rather than the comment character `#`. This means that Makefile rules which are intended to work only in the pre-distribution state – for example to generate distributed sources – should appear in `Makefile.am` with each line prefixed by `@PREDIST@`. After configuration, these rules will be enabled or disabled depending on the presence or absence of the marker files listed in this macro.

We should find either all of the marker files or none of them; if some but not all are found, this is probably an error of some type, so warn about it. This means, by the way, that it is the presence or absence of the first marker file which determines whether we are in the pre-distribution or post-distribution state, with the rest providing consistency checks.

The macro may be called more than once. Multiple calls are equivalent to a single call with all the marker files in the list. Automake checks that the files listed here are not in the list of distributed files, and issues a warning if they are.

If you use this `@PREDIST@` variable, then it is important to refer to all the files thus protected in a `STAR_PREDIST_SOURCES` call. The build will possibly still work if you forget to do this, but if so you are discarding a potentially useful consistency check.

For examples of use, see Sec. 4.3

A.28 STAR_PRM_COMPATIBLE_SYMBOLS

`STAR_PRM_COMPATIBLE_SYMBOLS`

This macro supports the use of the `VAL__` constants that are defined in the include file `PRM_PAR`. If checks if any special compiler flags are required to support PRM's use of typeless BOZ (binary octal, hexadecimal) constants, or if there's no typeless BOZ support any special flags that are required for that.

In fact this macro is only currently of use with the gfortran compiler which has no typeless BOZ support, so requires that the `-fno-range-check` flag is set so that assigments to integers can silently overflow (BOZ constants are replaced with their integer and floating point equivalents). In general this macro should be used by all packages that include PRM_PAR, all monoliths are assumed to use this by default.

A.29 STAR_SPECIAL_INSTALL_COMMAND

Third-party sources will not come with a `make install` target which installs a manifest. It is reasonably easy to add such support, however, as described in Sec. 4.5.3. The crucial feature is the *one line* installation command declared in the macro `STAR_SPECIAL_INSTALL_COMMAND`. If this macro variable is present in a `configure.ac` file, then Starlink automake will generate an `install` target which uses this command to do the installation, in such a way that a manifest file can be generated automatically. The argument to this macro is executed as the last command (or commands) in a subshell, the exit value of which is tested for a success status. For this to work, the command must have the following features:

(1) It must include the exact string `$(MAKE)` (as opposed to plain '`make`'), as part of some `$(MAKE) ... install` command.

(2) It must be prepared to install its files in a location which is prefixed by the value of the environment variable `$DESTDIR`. Recent versions of automake generate support for this variable automatically, older versions appear to have had the same functionality, but used a variable `$INSTALL_ROOT` for it.

(3) The subshell within which the given command is executed must exit with a zero status if the command is successful, and a non-zero status otherwise. This will happen if, for example, the last command in the macro argument is a 'make' command, since this exits with a non-zero status if the make fails, and this is passed on to the containing shell.

The example shown in Sec. 4.5.3 shows a use of this macro which satisfies these constraints.

The command may be as complicated as you like, but note that *it is collapsed onto a single line* by autoconf, so although you may spread the macro argument over several lines for readability, each command *must* be separated by a semicolon (or double-ampersand or double-bar).

See Sec. 4.5 for general discussion of importing third-party sources.

A.30 STAR_SUPPRESS_BUILD_IF

`STAR_SUPPRESS_BUILD_IF(test, message)`

Some components should be built only conditionally, if certain conditions apply. For example, the `sgmlkit` component can be built only if the host system has `jade` installed on it, but the lack of this is not an *error*, in the sense that it should cause a tree-wide `./configure` to fail. This macro helps with this situation.

If this macro is triggered, by the given condition being true, then `./configure`, `make` and `make install` *will all succeed* without compiling anything, with the latter installing a correctly formed manifest file which does not list any installed files.

The `test` in the first argument is a shell expression which should evaluate to true (ie, return a zero status) if the build should be suppressed. In the case of the `sgmlkit` component, the `configure.ac` includes the lines:

```
AC_PATH_PROGS(JADE, [openjade jade], NOJADE)
AC_PATH_PROGS(SGMLNORM, [osgmlnorm sgmlnorm], NOSGMLNORM)
...
STAR_SUPPRESS_BUILD_IF([test $JADE = NOJADE -o $SGMLNORM = NOSGMLNORM],
    [JADE=$JADE, SGMLNORM=$SGMLNORM: both needed for sgmlkit])
```

The `test` expression evaluates to true if either of the `$JADE` or `$SGMLNORM` variables has the special values given as defaults in the `AC_PATH_PROGS` macros earlier.

If the test is true, the message is displayed during the `./configure` run, but the configuration does *not* fail. Instead, a stamp file `STAR_SUPPRESS_BUILD` is left in the build directory, with the consequence that if you subsequently type `make`, that build will do nothing, emit the same message, and also not fail. If you subsequently invoke `make install`, that will also succeed, but install no files, except a manifest without a `<files>` element.

If you do any special installation work in your `Makefile.am`, remember to check for the existence of this stamp file, and make sure that you do not actually install anything if the file is present.

The difference between this and the standard `AC_FATAL` macro is that the latter is for an error in the configuration script – either a logic error there, or a serious problem like being unable to compile anything at all! Alternatively, if a dependency should be present because it is listed in the `configure.ac` as such, but it isn't, that's a case for a call to `AC_FATAL`. The `STAR_SUPPRESS_BUILD_IF` macro is for rather more well-mannered interactions with the user, indicating that there is other work they might have to do to get this component building.

You should use this macro rather sparingly. If a component depends on a component thus suppressed, there will likely be trouble.

A.31 STAR_XML_DOCUMENTATION

`STAR_XML_DOCUMENTATION(documentcode, [targets=''])`

Generate the standard makefile targets to handle XML documentation source. The parameter `documentcode` should be something like 'sun123' – it should not include any `.xml` extension.

For details, see the macro STAR_LATEX_DOCUMENTATION, which this macro closely matches, and which is more often used. The differences are as follows:

- The default list of targets is `.texml_tar .htx_tar .ps .pdf`, though using the second argument (discouraged) requires you to specify values for the `JADE`, `SGMLNORM` and `SGMLKIT_HOME` substitution variables (the `.texml_tar` tarball contains a `.tex` file plus any required figures).
- This macro defines the substitution variables `@STAR_XML_DOCUMENTATION@` and `@STAR_XML_DOCUMENTATION_<DOCUMENTCODE>@`.
- This macro declares a dependency on the sgmlkit component.
- The interface to `LATEX2DVI` is the same, and configure option `-without-stardocs` suppresses this documenation, too.

A.32 starxxx_DATA – special installation directories

There are several makefile variables which help you install files in special directories. These are the variables `stardocs_DATA`, `staretc_DATA`, `starexamples_DATA`, `starhelp_DATA` and `starnews_DATA` (there is also `starfacs_DATA`, for facility files, but this is handled for you, and you should not need to refer to this explicitly). In the `Makefile.am` the variable has as value a list of files to be installed as data in the corresponding place.

The list of documentation files is not necessarily known in advance, since the macro STAR_LATEX_DOCUMENTATION has a default list of targets it will generate. You can give the value of the `stardocs_DATA` as a substitution variable:

```
stardocs_DATA = @STAR_LATEX_DOCUMENTATION@
```

(plus any other documentation files you require). See Sec. 4.6 for more discussion.

A.33 Obsolete macros

A.33.1 AC_F77_HAVE_OPEN_READONLY

AC_F77_HAVE_OPEN_READONLY

Obsolete: use `AC_FC_OPEN_SPECIFIERS` instead

A.33.2 STAR_DOCS_FILES and friends

Obsolete: The `STAR_X_FILES` macros (for X in `DOCS`, `ETC`, `EXAMPLES` and `HELP`) are now obsolete, and you should use the corresponding makefile variables instead – see Sec. A.32.

A.33.3 STAR_FC_LIBRARY_LDFLAGS

`STAR_FC_LIBRARY_LDFLAGS`

Obsolete: use standard `AC_FC_LIBRARY_LDFLAGS` instead – the extra functionality which this macro added to the standard one is now handled automatically.

A.33.4 STAR_HAVE_FC_OPEN_READONLY

`STAR_HAVE_FC_OPEN_READONLY`

Obsolete: use `AC_FC_OPEN_SPECIFIERS(readonly)` instead.

Appendix B
The relationship with the GNU autotools

The build system is based heavily on the GNU autotools, autoconf, automake and libtool. Each of these is checked in to the CVS repository, and built and installed during the top-level `./bootstrap` process. Run the commands with the `-version` option to see the actual installed version.

The checked-in libtool is an unmodified libtool distribution.

The Starlink autoconf is based on version 2.59, with additions to support preprocessable Fortran; these modifications have been submitted to the autoconf maintainers, so it is hoped that these will become part of the autoconf distribution, at which point the Starlink autoconf will revert to being an unmodified one.

The additions are the macros which start `AC_...` described in Sec. A.

The Starlink automake is a more heavily modified version of automake. There are a few generic changes to support the added Fortran autoconf macros, but the majority of the changes are in the service of Starlink's specific requirements. They are summarised here.

- Addition of the `MONOLITHS` and `TASKS` primaries: These function rather like the `PROGRAMS` primary, in that the the `MONOLITHS` primary specifies one or more monoliths to install, and the `TASKS` one or more IFL tasks to associate with it. Like the `PROGRAMS` primary, each `MONOLITHS` variable should have a `xxx_SOURCES` variable associated with it. Starlink `automake` also emits the makefile rules to compile `.ifl` files to `.ifc` files.
- Addition of the `stardocs`, `staretc`, `starhelp` (and so on) prefixes to the `DATA` primary. Setting these variables has the same effect as giving an argument to the `STAR_DOCS_FILES` macro (and friends). See Sec. A.33.2.
- Extra `LDADD` options: In standard `automake`, the content of the `LDADD` variables must be a list of extra objects to add to a program, or extra libraries, specified through `-l` and `-L` flags. In Starlink `automake`, you can additionally specify extra libraries using the standard Starlink `*_link` and `*_link_adam` commands. You can add to the `LDADD` variable tokens matching `‘[a-z_]*_link‘` or `‘[a-z_]*_link_adam‘` (that is, a sequence of lowercase letters and underscore), or either of these preceded by *exactly* `$(srcdir)/`, which you would need to do only in the case of test code or the case below.

  You cannot include any options or arguments. If you want, for example, `ast_link -pgplot`, then you should create a local script named, for example, `my_ast_pgplot_link` (that is, matching the pattern above), which simply invokes the correct link script. In this case, you will have to include the `$(srcdir)/` element in the line, so that the script will be picked up from the current directory.
- Starlink `automake` spots use of the `STAR_MESSGEN` and `STAR_LATEX_DOCUMENTATION` macros in the associated `configure.ac`, and produces appropriate extra build rules in the generated `Makefile.in`. If the `STAR_MANIFEST_DIR` is to be substituted by autoconf, then Starlink automake adds support for installing manifests to the generated makefile.
- Default strictness: Starlink automake adds the `-startree` strictness level to the default `-foreign`, `-gnu` and `-gnits`. At present, it is almost indistinguishable from `-foreign`, and is the default (the default for GNU automake is `-gnu` strictness – see also Sec. C).
- All of the installation rules have been modified to recognise the presence of the `MANIFEST` makefile variable, and if it is set true, to include the path to the installed file in the output of the command. The modified `install` target manages this variable. The makefile 'install' target has been modified to create and install such manifests by default.

- The `missing` script which is installed as part of Starlink automake respects the `MISSING_SUPPRESS_RUN` environment variable.
- The package version is reported as `1.8.2-starlink` rather than plain 1.8.2.

Appendix C
Possible future changes

The starconf system is still quite new, and there are certain features of it which might change in the light of experience. These are summarised here, as a warning of what is not finalised, as reassurance if some of the documentation here seems out of date (in which case tell the document maintainer), and as a note of what aspects might need further discussion.

- The starconf application currently only generates two files – you should not assume this will always be so.
- There are only two options to `STAR_DEFAULTS` – the set will likely expand.
- The description of the tree-wide bootstrapping process in Sec. 3.2 is likely to change as more of the Starlink legacy and new code is brought under this build system umbrella. The interface to the 'make world' build may well change (see Sec. 2.4).
- The default automake strictness is `-starlink`, which is currently almost identical to the standard automake `-foreign` strictness. We may change this, and add other files.
- The `AC_PROG_FPP` macro described in Sec. B will *very* likely change its interface.