

SSN/73.2 (draft)

Starlink Project
Starlink System Note 73.2 (draft)

Mark Taylor
13 August 2001

EXTREME — Handling extreme data sets

Abstract

This package provides some utilities, background documentation, and associated files for adapting the Starlink Software Collection, and software which uses it, to handle very large data sets. The principal focus of this is to move to use of 64 bits of address space on 64-bit operating systems.

This document (SSN/73) is squarely aimed at the problem of adapting the Starlink Software Collection, and consequently focuses on the three operating systems (Solaris, Linux and Tru64) supported by Starlink, the compiled languages Fortran 77 and ANSI C, and Starlink's somewhat idiosyncratic build mechanisms. However, some of the discussion here may be of interest or use to people who are considering the change from 32 to 64 bits for software in other contexts.

Contents

1	Introduction	1
2	Changeover to 64-bit systems	2
3	Modifications to code	2
3.1	Pointer references in Fortran 77	3
3.2	Enlarging integer type in Fortran 77	4
3.3	Enlarging integer type in C	6
3.4	Changes to makefile/mk	11
4	Overview of tools for making the modifications	15
5	EXTREME cribsheet: Converting for extreme data sets in 99 easy steps	18
A	Description of tools	22
	crepint	23
	frepint	25
	inscnf	27
	extmk	29
	extmakefile	30
	cmp-xxx	31
	do-xxx	32

1 Introduction

The Extreme Data Set project is intended to allow processing of “unusually” large data sets by Starlink software, although the sizes for which special measures are required will become less and less unusual as time goes on. The principal underlying problem is that as images get larger, 32 bits are no longer enough to index into an image. The largest integer that can be stored in 32 bits is approximately 4×10^9 (unsigned) or 2×10^9 (signed). If the operating system itself uses unsigned 32 bit pointers to address bytes in memory, this means that it is impossible to map into memory an image of more than 4Gbyte or, say, two images of half that size. This could correspond to, for instance, an input and an output image simultaneously mapped each with an HDS type of `_REAL` and size of 23k pixels square.

For this sort of work therefore an operating system with 64-bit pointers is required.

For the systems supported by Starlink this currently means that Compaq Tru64 Unix can be used, as can Solaris running in 64-bit mode. On appropriate hardware the Solaris kernel may be compiled for 32 bit or 64 bit mode; but almost¹ all binaries which run on the 32-bit version will run equally well on the 64-bit version, so that reconfiguring a system from 32-bit to 64-bit should be fairly painless from a software point of view. You can tell if your Solaris kernel is 64-bit by using the `isainfo -v` command; on a 64-bit system the following response will be given

```
% isainfo -v
64-bit sparcv9 applications
32-bit sparc applications
```

User code will run up against similar problems to those faced by the operating system when coping with large images. It is often necessary to count the pixels, or the bytes, in an image, and this is typically done using a Fortran `INTEGER` or a C `int`. These are normally signed 32-bit values, with a maximum value of about 2×10^9 ; the pixel count of a 47k \times 47k image, or the byte count of a 16k \times 16k `_DOUBLE` image, will overflow this limit.

Another common requirement is holding a pointer to allocated memory, which has ultimately been acquired from a C routine such as `malloc`, in a variable. In C this will be taken care of automatically because the compiler ensures that pointer types are long enough to hold memory addresses. In Fortran 77 however there is no pointer type so that `INTEGERs`, which are normally 32 bits, have to be used. The solution to this, explained in SUN/209, is to use the `CNF_PVAL` function.

The issues addressed in this document apply to user programs which link against Starlink libraries as well as to the code which forms the USSC; if the USSC has been built for a 64-bit system, then user code which uses its libraries will need to be modified at the source level in order to work. Depending on the complexity of the code, it may be easier to do this with a few manual adjustments than by using the automatic tools supplied with the EXTREME package. The discussion here should be of use in any case.

This package provides some tools and instructions for software maintainers to use in modifying their source code to take advantage of a 64-bit environment. The rest of this document is organised as follows:

Section 2 discusses the general strategy for moving to provide 64-bit support.

¹ There can be trouble with applications which use `libkvm` or access `/proc`.

Section 3 describes the changes which have to be made to each package at source level to achieve this.

Section 4 gives an overview of the tools provided by this package to help automate the source code modification.

Section 5 gives a detailed step-by-step set of instructions which can be followed to carry out the changes.

Appendix A gives a detailed listing of the behaviour of each of the tools provided.

It is suggested that you read the first four sections for an understanding of why and how the various changes need to be made, and then follow the steps in section 5 if you have a package which you need to modify. Appendix A will be useful for reference or details of the exact behaviour of the tools.

2 Changeover to 64-bit systems

In general, it is not possible to modify packages to deal with large images independently of each other; before changing a package to use 64 bits, it is necessary that all the packages on which it depends are 64-bit compatible. Certain packages, which are used only at a command-line level, may not need changes to the code, but in general a package which provides a library against which other code is to link must be modified, since the public interfaces of the routines, specifically the types of integer arguments, will change. Additionally, in the case of Solaris, 32-bit and 64-bit object types cannot be mixed by the link loader.

At some point in the future, it may be that support for 32-bit systems is dropped, but at the moment this is undesirable, since Solaris systems running 32-bit kernels would be incapable of running code compiled for 64 bits. Older Suns (before SPARC v9) cannot be made to run the 64-bit kernels.

For these reasons, the transfer to 64 bits will be approached effectively as ports to new systems, which will be supported alongside the existing ones. All Starlink packages should therefore come to include support for two new values of the `makefile/mk SYSTEM` variable: “`alpha_OSF1_64`” and “`sun4_Solaris_64`”.

3 Modifications to code

Making Starlink packages able to run in a 64-bit environment and handle appropriately large data sets requires changes to the source code of each package. For packages like HDS, which get their hands dirty with data structures on disk, extensive changes will be required. Most packages however can hopefully be fixed by making some more-or-less automatic changes to the source code. These changes boil down to making sure that Fortran pointers can address any allocated memory, and making sure that non-pointer integers in Fortran and C have enough bits to hold sufficiently large numbers.

Because of the difficulty of distinguishing, in any automatic fashion, variables which may need to hold big numbers from those which will not, the recommended approach is to change the type of *all* integer variables to a 64-bit type in 64-bit environments. There are undesirable consequences of this, the main one being that the HDS type `_INTEGER` becomes, as far as the user of HDS is concerned, a 64-bit type; it will still be stored on disk in 32 bits, but it is in any case a function of HDS to shield the user from the disk representation of defined types. Where images are mapped with `_INTEGER` type in the new system therefore, they will take up twice as much space as they need to in memory, and the process of mapping and unmapping them can be expected to be slower. There is nothing much to be done about this; fortunately, `_INTEGER` should not be a very common type for storing large data arrays in an NDF.

Once the increase in integer size has been made globally to the source code in a package, it would be a possibility to identify variables which do not need the extra size and change their declarations (and other parts of the code which depend on these) back to the normal integer types. In most cases such retrochanges will not be worthwhile, but they could be appropriate in either of the following situations:

- The change has broken something which is easier to fix by returning to a normal integer size than by other means
- The change results in significant overuse of resources, and returning to a normal integer size will not cause too many problems.

The rest of this section describes the four main sets of changes which need to be made to each package.

3.1 Pointer references in Fortran 77

As explained in SUN/209, whenever the `%VAL` Fortran compiler directive is used to pass the value of registered allocated memory to a subroutine, its argument should be wrapped in a call to `CNF_PVAL`. This effectively allows a 32-bit integer to address memory in a 64-bit address space. This call could in principle be avoided when long integers are being used in Fortran but it is still necessary where Fortran is written using normal integers in a 64-bit environment, so these calls must be incorporated. Since this change is less problematic than the move to large integers and is mostly independent of it, it is possible and advisable to make this change to all code and test it before converting the normal integer types as described in section 3.2.

As well as the calls to `CNF_PVAL` themselves, the header file `CNF_PAR` needs to be included in affected source files to provide the declaration of the `CNF_PVAL` function.

The main pitfall of making these modifications is wrapping the argument of a `%VAL` in a `CNF_PVAL` call when it ought not to be so wrapped. `CNF_PVAL` should *only* be applied to registered pointers. If your Fortran code uses pointers which are got from C without being registered with CNF, you'll need to address this in the C code as described in SUN/209. More commonly, you may have `%VAL` invocations which are not applied to pointers at all — for instance when passing the lengths of character variables as trailing arguments which sometimes needs to be done when the compiler doesn't know that it's passing a character variable. In this case the `%VAL` invocation should be left alone and no call to `CNF_PVAL` inserted. In general, it is impossible to spot all such non-pointer uses of `%VAL` automatically. However, if `%VAL` is applied to a constant, or a small integer, then it should be left alone.

The program `inscnf`, and the corresponding driver script `do-inscnf`, is supplied for inserting CNF_PVALs in the right places. It wraps *all* arguments of `%VAL` in calls to `CNF_PVAL`, but emits warnings for those which look like they may not be pointers. It is described further in appendix A.

3.2 Enlarging integer type in Fortran 77

In Fortran 77 there is no (even slightly) portable way of declaring an integer to be of a configurable length. Therefore all variables currently declared as `'INTEGER'` should be changed instead to type `'INTEGER*8'`. This is not standard Fortran 77, but it is widely understood by compilers. This should not be interpreted (by a human) as an indication that exactly eight bytes are required, but as an indication that the variable in question needs to be long enough to hold a large number. At compile time, typically on 64-bit systems the code will be compiled as it stands, while on 32-bit systems a simple `sed` command or similar can preprocess the source code before it is fed to the compiler, under `makefile` control (see section 3.4).

Having done this, the following issues may need to be addressed.

External libraries

In Fortran, the actual arguments used in a subroutine/function call must be of the same type as the formal arguments declared in the routine itself, since arguments are passed by address. Thus if you change the type of integers passed to a subroutine, you have to change the type of the integers declared in the subroutine too. It is therefore necessary either to convert any called subroutines to using the new integer type (at least as far as their arguments go), or to ensure that any calls to these routines use variables of the old integer type. Thus a package can only be successfully converted to large integers and built once all the libraries against which it links have been similarly converted. The Starlink libraries will therefore have to be done before applications packages. If a program uses an external library which is not part of the USSC, then either the library will have to be converted in the same way, or the affected calls will have to be written using variables specially declared as the normal `INTEGER` type.

Literal integer constants as arguments

If there are any calls to functions or subroutines in the converted code which have literal integer constants as actual arguments, these will need to be replaced with expressions of the right type. There is no way of specifying the length of a literal integer constant in Fortran, so you must ensure that the expression has the correct type in some other way. The recommended method for doing this is to use a variable for the purpose, which may be assigned using a `PARAMETER` statement. Thus

```
CALL SUB( 5, STATUS )
```

must be changed to something like

```
INTEGER * 8 INT__5
PARAMETER ( INT__5 = 5 )
...
CALL SUB( INT__5, STATUS )
```

Other methods of getting around this are possible, for instance using statement functions. The include files `EXT_PAR`, which contains a set of predefined integer constants, and `EXT_DEC` and `EXT_DEF`, which define statement functions in the same way as the `PRIMDAT`

package, are provided with the EXTREME package by way of example of alternative strategies.

Intrinsic functions

Calls to standard Fortran intrinsic functions should take care of themselves, as long as the generic name is used rather than the specific one (e.g. you should use ABS instead of IABS), since they adapt themselves automatically to the type of their argument. Use of generic names is generally good practice anyway; the only time it is necessary to supply the specific name is when passing an intrinsic function itself as an actual argument to a procedure, which is pretty rare. The only intrinsic functions in standard Fortran 77 which have INTEGER-specific names are IABS, ISIGN, IDIM, MAX0, AMAX0, MIN0 and AMIN0.

There are some non-standard intrinsic functions which require INTEGER arguments, and will cause trouble if they have INTEGER*8 arguments. These are mostly system-specific ones, such as GETARG, and are likely only to be used in fairly low level code.

A related problem is using a call to an intrinsic function with an integer return type as an actual argument to a function. Since for most functions the return type is the same as the argument, this will, again, take care of itself. However, for the few intrinsic functions which return integer regardless of the type of the argument (LEN, INDEX), the return value will have to be converted to the correct type as if it were an integer literal, by assigning to an intermediate variable or using a statement function.

Storage association

The Fortran 77 standard does not state how much storage is used by an INTEGER, but does state that it should be the same amount as that used by REALs and LOGICALs. Converting all INTEGERS to INTEGER*8s will normally have the effect that this is no longer the case. Thus code which relies on this, for instance by using COMMON blocks or EQUIVALENCE statements to address the same memory as both INTEGER and REAL, will break. Such practices are, as you might expect, deprecated by the Starlink Application Programming Standard, but there may be instances of them lurking about.

Similarly, code which makes the assumption that four elements of a CHARACTER array correspond to each INTEGER will need changing.

Storage alignment

The compiler documentation indicates that 64-bit data types on Sun and Alpha systems should be aligned on 64-bit boundaries for performance reasons. Normally this is taken care of automatically by the compiler, but in COMMON blocks (or EQUIVALENCE?) the address of a variable is determined by the source code, so that the compiler is not free to move it around (the start of a COMMON block is always aligned to the largest boundary, 64 bits). Thus the following code:

```
REAL      XX, YY
INTEGER * 8 IX, IY
COMMON /BLOCK/ XX, IX, YY, IY
```

which would align everything (happily) on 32-bit boundaries if IX, IY were INTEGER*4s, will be forced to align the 64-bit IX on a 32-bit boundary. The compilers will warn about this behaviour. I believe that the result will be slower, rather than incorrect, executables, but such occurrences should ideally be fixed.

IMPLICIT variable declarations

It is highly desirable in Fortran, and usual in Starlink code, to use the “IMPLICIT NONE” declaration, so that all variables have to be explicitly declared before use. If this is not done, and integer variables are used without explicit declaration, they will not be converted to INTEGER*8. It would be possible to fix this by redeclaring IMPLICIT INTEGER statements as IMPLICIT INTEGER*8 and/or adding an IMPLICIT INTEGER*8 (I-N), but to make the job easier for the makefile in editing the source code, it is better to place an “IMPLICIT NONE” statement in every source code module, and make explicit INTEGER*8 declarations for all integers.

I/O return values

A few Fortran I/O statements have specifiers which require the name of an INTEGER variable. In standard Fortran 77 these are IOSTAT in all I/O statements, and NUMBER, RECL and NEXTREC in the INQUIRE statement. Particular compilers often provide a whole lot more, but these should hopefully not be much used in Starlink code. So there may be a problem with a statement like

```
CLOSE( UNIT = 99, IOSTAT = STVAL )
```

if the variable STVAL is now declared INTEGER*8 rather than INTEGER. Of the supported systems, Tru64 and Solaris seem to handle INTEGER*8 variables used for this purpose without complaint. Linux with g77 0.5.24 may fail to compile although (a) it seems to be OK if you compile with -O, and (b) this is reportedly fixed at g77 0.5.25. So it’s probably all right to leave these cases.

The program `frepint`, and the corresponding driver script `do-frepint`, is provided for changing INTEGER declarations to INTEGER*8. Details of which of the above constructs it fixes, and which it warns about, are given in appendix A.

3.3 Enlarging integer type in C

All references to the type `int` in C code should be changed to `INT_BIG`. This is a reference to a macro which can be defined in a system-dependent header file or on the compiler command line (and hence typically in the `mk` environment variable `CFLAGS`). For a 32-bit build it would normally be defined at build time as `int` and for a 64-bit build as `long`. As well as declarations of variables and functions, this applies to almost any other syntactically significant occurrence of the identifier `int`, for instance casts and arguments of `sizeof`. References to the types `short int` and `long int`, which are just synonyms for `short` and `long` respectively, should not be changed. Type `unsigned INT_BIG` (or, redundantly, `signed INT_BIG`) may be used. There are some places where an `int` declaration is implicit, and here an `INT_BIG` must be inserted.

As regards calling functions from external libraries, the approach described in section 3.2 of recoding all the functions in the library will of course work in C as well as in Fortran. However, this is not always necessary. If the function’s ANSI C-style declaration (prototype) is in scope when the function is called, the compiler will normally arrange for conversion of each actual argument to the declared type of the corresponding formal argument in the declaration before passing it by value. Hence the following code

```
int add( int a, int b );
INT_BIG i, j;
add( i, j );
```

is correct whatever integral type `INT_BIG` is defined as, since `i` and `j` are converted to type `int` before `add()` sees them. If the prototype of `add()` were not in scope however, the conversion would not take place and the code would be in error.² The lesson is to make sure that header files are included. Of course this relies on having ANSI C-style function prototypes; if old-style function declarations are used then no argument type conversions are made. For code which uses old-style function declarations the best thing is to convert it, or at least a corresponding header file, to ANSI C style.

However this does not solve all problems. Where a function has a variable argument list (as declared in the prototype using ellipsis `'...'` and handled in the function using the `stdarg.h` macros), the function prototype is not able to specify the types of all arguments, and so the type of the actual argument must match the type of the formal argument for correctness. If it is impractical to recode the function (as in the case of `printf`), the best solution is to cast the variable arguments to the type which is expected where the function is called.

A more difficult problem is when the address of an argument is passed so that the contents of that address can be changed by the function. In this case if the called function has a different idea of the length of the object being pointed to it will write to the wrong amount of memory, possibly overwriting other data. Consider this function:

```
void zero( int *a ) {
    *a = 0;
}
```

and this code:

```
INT_BIG x;
zero( &x );
```

If `INT_BIG` is a 64 bit type and `int` is 32 bits then only half the bits in the variable `x` will be zeroed by this call. If the function declaration is in scope when the call is made, the compiler will issue a pointer mismatch warning about this sort of thing; again, make sure that the appropriate header files are included. Again, in the case of variable argument lists, the compiler can't spot it.

To summarise, external functions should be declared before use by including the appropriate header files. If this is done, then the only problems associated with calling functions which have not been converted to use `INT_BIG` instead of `int` should be:

INT_BIG in variable part of argument list: A modified caller of an unmodified function should explicitly cast an `INT_BIG` argument in the variable `(...)` part of the argument list. Normally the cast should be to `int`, but it may be possible, as with `printf`, to cast to `long` and indicate to the called function that this has been done. See the `printf` example below. A modified function which may get called by unmodified code should expect arguments of type `int` (i.e. should call the `va_arg` macro with a second argument of `int` instead of `INT_BIG`).

Pointer to INT_BIG variable passed: A modified caller of an unmodified function will have to declare a local variable of type `int` and exchange values between it and the `INT_BIG` before, and possibly after, the call. See the `scanf` example below. A modified function which may

² In fact, the standard says simply that the effect of the call is undefined in this case. Furthermore, it seems that the three currently supported systems handle these cases without any undesirable behaviour; presumably the compilers are written such that all arguments are passed in 64 bits or in registers. But such code is still not correct.

get called by unmodified code will have to declare pointer arguments as pointers to a given fixed type (presumably `int`), not to `INT_BIG`.

Overflow: If an `INT_BIG` value which is too large to be an `int` is passed to a variable which is an `int`, arithmetic overflow will occur when C tries to do the type conversion according to the function declaration. No warning is issued by the Solaris or Tru64 C runtime systems about such overflows.

External libraries which code may have to link against can be split into a few categories:

Starlink libraries

As with Fortran, the plan is for Starlink libraries to get converted to use `INT_BIG` before code which uses them (although for C code calling C libraries this is not so necessary as with Fortran when functions are pre-declared using header files).

Blocks of source code not to be converted

It may not be a good idea to do `INT_BIG` conversion to large bodies of non-Starlink code used by the USSC; Perl and Tcl spring to mind. If there is function-level access to these packages, some recoding may be required as above.

The C standard library

The functions of the C standard library are no different from any other unconverted external library, but since their use is common, it is discussed in detail here. Most of the functions in the standard library will be handled adequately as described above by including the appropriate header files, since they do not have `int *` arguments or variable argument lists. The only exceptions in the standard library are as follows:

`frexp`

The second argument of the mathematical function `frexp` is declared `int *`, so a pointer to `int` and not to `INT_BIG` must be passed.

`bsearch, qsort`

Both these functions take as arguments a comparison function declared `int (*)(void *, void *)`, i.e. a function returning `int`. Although the calls to `bsearch` and `qsort` do not need to be modified therefore, the function passed to them must be of the right type, and not modified to be of type `INT_BIG (*)(void *, void *)`.

`signal`

The second argument of `signal` is declared as `void (*)(int)`. Calls to `signal` do not need to be modified, but functions passed to it ought to be declared functions of `int` and not of `INT_BIG`.

`printf, scanf`

The functions which use format strings and variable argument lists, `printf`, `fprintf`, `sprintf` and `scanf`, `fscanf`, `sscanf`, require careful scrutiny. For `printf` and friends any of the format specifiers `cdiouxX`³ indicate that the corresponding argument should be an `int`, and the `n` specifier indicates a pointer to `int`. For `scanf` and friends, any of the specifiers `diouxXn` indicate pointer to `int`. If any of the actual arguments

³ Note the inclusion of the `c` format specifier in this list. Although the corresponding argument will typically be of type `char`, it is promoted to `int` by the usual mechanism before being passed to `printf`. If the type of the actual argument is of type `char` and not type `INT_BIG` of course, no change will be required here.

in the call is of type `INT_BIG` (or `INT_BIG *`) when it should be of type `int` (or `int *`), then the calling code needs to be changed.

In the case of `int` arguments, if the actual argument might be too large to be represented in an `int`, then an `l` should be inserted after the `'%` sign to indicate that a long argument is being supplied and the corresponding argument should be cast to `long`. If it will definitely be possible to store the value in an `int` then the format specifier may be left alone and the argument cast to `int`. Arguments passed using the `c` or `*` specifiers should be cast to `int`, and not modified with an `l` character. In the case of `int *` arguments, intermediate variables have to be used.

Here is an example. If after simple substitution of `INT_BIG` for `int` a piece of code reads:

```
extern INT_BIG triple( INT_BIG x );
INT_BIG i, j;
char c;
scanf( "%i %c", &i, &c );
j = triple( i );
printf( "Integer tripled is %i; Character is '%c'\n", j, c );
```

then the `scanf` call must be replaced by something like this:

```
{
    long tmp;
    scanf( "%li %c", &tmp, &c );
    i = tmp;
}
```

and the `printf` call by something like this:

```
printf( "Integer tripled is %li; Character is '%c'\n", (long) j, c );
```

Other external libraries

If your code links to any other external libraries which cannot, or will not, be converted to use `INT_BIG`s, some recoding of the calls may be required as above. The most common case of this is use of the various system libraries whose functions are declared in header files in or under `/usr/include`. `waitpid()`, `signal()` and `pipe()` are a few of the culprits.

There are a few other issues which arise from replacing `int` type with `INT_BIG`:

Integer constants from `limits.h`

Where an `int` is compared against one of the values `INT_MAX`, `INT_MIN` and `UINT_MAX` defined in the system header file `limits.h`, an `INT_BIG` should be compared against one of the corresponding macros `INT_BIG_MAX` etc. These macros are defined in the header file `extreme.h`.

Implicit `int` declarations

There are several places in C (macros and typedefs apart) in which an identifier can be declared as an `int` without the `int` reserved word appearing. For example, in the code:

```
sub( x ) {
    static y;
    signed z;
}
```

the symbols `sub`, `x`, `y` and `z` all have type `int` so that the converted code should read

```

INT_BIG sub( INT_BIG x ) {
    static INT_BIG y;
    signed INT_BIG z;
}

```

int used for Fortran LOGICAL

Where a C `int` is used to represent a LOGICAL variable in Fortran, it should not be replaced by `INT_BIG`. This is only likely to arise in certain low-level code (e.g. HDS and CNF) which does direct interfacing with Fortran. Under normal circumstances code should use the macros `F77_LOGICAL_TYPE` and `F77_INTEGER_TYPE` defined in the CNF header file `cnf.h`.

The program `crepint`, and the corresponding driver script `do-crepint`, is provided for making some of these changes. It replaces all references to `int` type, with a few exceptions, by `INT_BIG` type, modifies explicit declarations which are implicitly of type `int`, and warns about constructs which might need further attention. Full details of which of the above constructs it fixes, and which it warns about, are given in appendix A.

A construction which `crepint` misses altogether is finding implicit declarations in function prototypes, which are implicitly of type `int`. These can be spotted by suitable use of the C compilers. Given a file `sub.c` which reads:

```

sub( x ) {
    return x;
}

```

then running `gcc` with the `-Wimplicit` flag generates warnings for such declarations:

```

% gcc -fsyntax-only -Wimplicit-function-declaration -Wstrict-prototypes sub.c
sub.c:1: warning: return-type defaults to 'int'
sub.c:1: warning: function declaration isn't a prototype

```

The `-proto` flag of Tru64 Unix's C compiler is a little more effort to use, but produces more concise output. Running

```

% cc -protois -noobject sub.c

```

will produce a file called `sub.H` which reads:

```

extern int sub(int x);

```

Occurrences of `int` in the output file `sub.H` should be changed to read `INT_BIG` wherever the function is declared or defined in the source code (typically in a source file and maybe a header file), so that `sub.c` should end up reading:

```

INT_BIG sub( INT_BIG x ) {
    return x;
}

```

By running

```

% cc -protois -noobject -DINT_BIG=long *.c

```

and attending to any `int` declarations in the resulting `.H` files, it should be possible to find any offending implicit declarations. Note however that this only writes function prototypes from function definitions, it does not normalise existing prototypes, so it cannot usefully be applied to header files.

3.4 Changes to makefile/mk

Since not all build environments require the enlarged integers, the build process must be modified so that 64-bit integers are used for some build environments and 32-bit integers for others. Identifying build environments is done by using different values of the SYSTEM macro. As well as the existing supported values

- ix86_Linux
- sun4_Solaris
- alpha_OSF1

which indicate that 32-bit integers should be used as before, the following new ones are added

- sun4_Solaris_64
- alpha_OSF1_64

which indicate that 64-bit integers should be used. Thus two new stanzas are required in the mk file for each package setting the values of the other makefile macros for the newly supported systems.

Additionally, if the package contains Fortran source, a new macro INTEGER8 should be defined by the mk file. This should contain text which is to replace 'INTEGER * 8' declarations in fortran source code. If blank (defined as spaces or the empty string) it means that INTEGER * 8 declarations should be left unchanged. Therefore for 64-bit systems this should be defined as the empty string, and for 32-bit systems it should be defined as 'INTEGER ' (defining it with four trailing spaces is not required, but improves the aesthetics of the modified source files). The makefile should then apply the INTEGER*8 → \$(INTEGER8) substitution to the fortran source files as it extracts them from the tar archive where they are stored. The recommended code for this is:

```
sed "s/^[ \t]*[Ii][Nn][Tt][Ee][Gg][Ee][Rr] *\* *8/ $(INTEGER8)/"
```

which should be applied only when test \$(INTEGER8) is true. Note that this changes nothing except variable declarations; in particular it will not modify references to INTEGER*8 type in IMPLICIT statements. It also makes no allowance for the more bizarre spacing possibilities permissible in Fortran 77. It would be possible to use a more comprehensive editing script, but in the interests of clarity, and of simplicity at build time, it is recommended that the above is used, and source code required to match the given pattern where edits are needed.

The intention is that Fortran source code is stored in the source archive with INTEGER*8 declarations, and is edited if necessary as it is extracted from the tar file, prior to being written in the build directory. In this way, the set of Fortran files in the build directory will be consistent and compilable, though it may not have the same content as the set of files in the source archive.

For C files, the source code itself does not need to be modified at build time, but the INT_BIG preprocessor macro must be defined. This is most easily done by adding a -DINT_BIG=int or -DINT_BIG=long flag as appropriate to the CFLAGS macro in each mk stanza. Alternatively, a SOURCE_VARIANT-controlled header file could be written and included into all C source files; the header file extreme.h could be used for this purpose.

The following gives the relevant parts of a basic mk file by way of example:

```

...
# Supported Systems:
#   The following systems are currently supported and may be
#   identified by defining the SYSTEM environment variable
#   appropriately before invoking this script:
#
#   alpha_OSF1
#       DEC Alpha machines running OSF1
#
#   alpha_OSF1_64
#       DEC Alpha machines running OSF1, long integers
#
#   ix86_Linux
#       Intel PC running Linux
#
#   sun4_Solaris
#       SUN Sparcstations running SunOS 5.x (Solaris)
#
#   sun4_Solaris_64
#       SUN Sparcstations running 64-bit SunOS 5.x (Solaris), long integers
...
#   CFLAGS (-O -DINT_BIG=int)
#       The C compiler options to use.
#
#   INTEGERS8 (INTEGER    )
#       Replacement text for 'INTEGER * 8' declarations in the original
#       original Fortran source code.  If set to the empty string,
#       INTEGER * 8 declarations will not be modified.  For 64-bit
#       systems which must be able to deal with very large images
#       this should be set to the empty string (or 'INTEGER*8').
#       Otherwise, more efficient code may be generated by setting
#       it to 'INTEGER' or 'INTEGER*4'.  The trailing whitespace is
#       optional but may make source code more readable.
...

export INTEGERS8
...

case "$SYSTEM" in

# DEC Alpha machines running OSF1.
# -----
    alpha_OSF1)
        CFLAGS='-DINT_BIG=int'
        INTEGERS8='INTEGER    '
        ...

# DEC Alpha machines running OSF1, long integers.
# -----
    alpha_OSF1_64)
        CFLAGS='-DINT_BIG=long'
        INTEGERS8=''
        ...

```

```

# SUN Sparcstations running SunOS 5.x (Solaris).
# -----
    sun4_Solaris)
        CFLAGS='-DINT_BIG=int'
        INTEGER8='INTEGER      '
        ...

# SUN Sparcstations running 64-bit SunOS 5.x (Solaris), long integers.
# -----
    sun4_Solaris_64)
        CFLAGS='-DINT_BIG=long -xarch=v9'
        FFLAGS='-xarch=v9'
        INTEGER8=' '
        ...

# Intel PC running Linux.
# -----
    ix86_Linux)
        CFLAGS='-DINT_BIG=int'
        INTEGER8='INTEGER      '
        ...

```

Note the `'-xarch=v9'` addition to the `sun4_Solaris_64` `CFLAGS` and `FFLAGS` variables, which instructs the compiler to compile for a 64-bit system. Tru64 Unix C compiles for 64 bit executables by default, so needs no extra flags.

It may also be desirable to add the lines:

```
SOURCE_VARIANT='alpha_OSF1'
```

and

```
SOURCE_VARIANT='sun4_Solaris'
```

to the `alpha_OSF1_64` and `sun4_Solaris_64` stanzas respectively, if the same machine-specific source files can be used for 32-bit and 64-bit builds on the same platforms; if this is not the case then `SOURCE_VARIANT` should be left to default to the value of `SYSTEM` as usual, and additional copies of the system-dependent files must be supplied.

The relevant parts of a suitable matching makefile would look something like this:

```

# Default values for macros for compiling C and Fortran source code.

CFLAGS = -O -DINT_BIG=int
...

# Default replacement text for Fortran INTEGER * 8 type.

INTEGER8 = INTEGER
...

# Macro for filter to replace INTEGER * 8 declarations with INTEGER8.

REPLACE_INTEGER8 = sed \
    "s/^[ \t]*[Ii][Nn][Tt][Ee][Gg][Ee][Rr] *\* *8/      $(INTEGER8)/"

```

```

# Rules for extracting non-Fortran source files from the source archive.

$(C_ROUTINES) $(PUBLIC_C_INCLUDES) $(PRIVATE_C_INCLUDES):
    $(TAR_OUT) $(PKG_NAME)_source.tar $@
    @ if test -f $@; then :;\
        else echo $@ is not in the tar file; exit 1; fi

# Rules for extracting Fortran source files from the source archive.

$(F_ROUTINES) $(PUBLIC_F_INCLUDES) $(PRIVATE_F_INCLUDES):
    $(TAR_OUT) $(PKG_NAME)_source.tar $@
    @ if test -f $@; then :;\
        else echo $@ is not in the tar file; exit 1; fi
    if test $(INTEGER8); then \
        $(REPLACE_INTEGER8) < $@ > $@_tmp; \
        mv $@_tmp $@; \
    else ;; fi

# Rules for extracting platform specific Fortran files from the archive.

$(PLATFORM_F) dummy_target2:
    $(TAR_OUT) $(PKG_NAME)_source.tar $@_$(SOURCE_VARIANT)
    @ if test -f $@_$(SOURCE_VARIANT); then :;\
        else echo $@_$(SOURCE_VARIANT) is not in the tar file; exit 1; fi
    if test $(INTEGER8); then \
        $(REPLACE_INTEGER8) < $@_$(SOURCE_VARIANT) > $@; \
        rm -f $@_$(SOURCE_VARIANT); \
        else mv $@_$(SOURCE_VARIANT) $@; fi
    ...

# Enter information about the current machine and build environment
# into the date stamp file.
...
    @ echo '    INTEGER8: $(INTEGER8)'    >>$(DATE_STAMP)

```

Lists of files for extraction from the source archive may have to be split into Fortran and non-Fortran sublists since the code for extracting them will now differ. For instance it might be convenient to replace assignments like this:

```
PUBLIC_INCLUDES = $(PKG_NAME)_par $(PKG_NAME)_err $(PKG_NAME).h
```

with something like the following:

```

PUBLIC_F_INCLUDES = $(PKG_NAME)_par $(PKG_NAME)_err
PUBLIC_C_INCLUDES = $(PKG_NAME).h
PUBLIC_INCLUDES = $(PUBLIC_F_INCLUDES) $(PUBLIC_C_INCLUDES)

```

so that the lists of Fortran and C files can appear as targets of separate extraction rules, as in the example above.

It may also be necessary to add dependencies for INCLUDE files inserted by the conversion tools; `inscnf` may include `CNF_PAR` and `crepint` may include `extreme.h` in some modified source files. Rules will need to be made for building links to the referenced include files, and dependencies added for the object files corresponding to the source files so modified.

Files may be extracted from the source archive(s) in more places than one in the makefile - in particular, for some reason lost in the mists of time the `do_test` target usually extracts required files from the source archives explicitly rather than relying on a dependency. It will be necessary to identify all such places (try grepping for `TAR_OUT`), and ensure that the `INTEGER8` macro is substituted in to the source code before it gets compiled.

Note that because of the way this approach edits Fortran source files on their way out of the source archive, the copies of the files in the build directory may not be the same as the files in the source archive, and so files cannot in general be transferred between the build directory and source archive simply using `tar`. This means that in general the source archive cannot be rebuilt from the files in the build directory. This may well break private, non-standard targets inserted into the makefile by the package developer (commonly called `archive`), and effectively means that it is not possible to tweak package source code in the build directory prior to re-exporting it.

The scripts `extmk` and `extmakefile` are provided to make the necessary modifications to `mk` and `makefile` files respectively, and are described fully in Appendix A. `extmk` can normally make all the required changes to `mk`, but `extmakefile` can only add some of the required parts; other edits such as deciding which files need to be extracted as Fortran and which as non-Fortran files must be made by hand.

4 Overview of tools for making the modifications

This section explains how to use the utilities distributed with this package to modify source code for use in 64-bit environments. The details of their operation are given in A.

To start up the EXTREME package, type

```
% extreme
```

```
EXTREME commands are now available -- Version 0.1-0
```

(if the `extreme` alias is not set up, try `source /star/bin/extreme/extreme.csh`). This makes the tools provided by this package available.

One tool is provided for each of the source code conversion tasks described in the previous section:

`inscnf`: Inserts `CNF_PVAL` calls where required in Fortran source code

`frepint`: Converts `INTEGER` to `INTEGER*8` in Fortran source code

`crepint`: Converts `int` to `INT_BIG` in C source code

`extmk`: Makes the necessary edits to a package `mk` file

`extmakefile`: Makes some of the necessary edits to a package `makefile`

Each of these is a normal Unix filter command with usage like `cat`, so may be given zero, one or two arguments to specify its input and output. Where no change needs to be made, the input is written with no changes to the output. This means that the `diff` command can be used to see what changes the filters make, by doing something like:

```
% frepint file.f | diff file.f -
```

which might give a result like:

```
94c94
<     INTEGER STATUS           ! Global status
---
>     INTEGER * 8 STATUS       ! Global status
97,98c97,98
<     INTEGER I                ! Loop variable
<     INTEGER INDF             ! NDF identifier
---
>     INTEGER * 8 I           ! Loop variable
>     INTEGER * 8 INDF       ! NDF identifier
```

Each of the filters, as well as making changes, draws attention to constructs which might need further attention by writing a message to standard error. In some cases, it will also insert a comment in the modified file where the questionable construct occurs. Such comment lines contain the name of the filter followed by a colon and explanatory text, so given the file `watcher.c`:

```
int watcher( int mins ) {
    printf( "Only %i minutes till coffee break\n", mins );
}
```

`crepint` will do the following:

```
% crepint watcher.c >fixed/watcher.c
crepint: Format string has %[cdiouX*] (comment inserted)
```

and the resulting file `fixed/watcher.c` will look like this:

```
INT_BIG watcher( INT_BIG mins ) {
/* crepint: Format string has %[cdiouX*]                               */
    printf( "Only %i minutes till coffee break\n", mins );
}
```

which can then be fixed by hand as described in section 3.3.

If the filter thinks it has lost track of the source code in a dangerous way, i.e. that it might be making changes which are likely to invalidate the code, it may exit with an error status, and print a message to standard error to that effect.

Each of the filters has a go at retaining the aesthetic qualities of the code; an attempt is made to respect case usage and spacing conventions in Fortran, padding whitespace is shuffled to keep things at the same column as before if possible, and so on. If replacement text is longer than the original then Fortran lines are broken in hopefully reasonable places, but no line breaks in existing lines are introduced in C.

In general, these filters try to be as helpful as possible, and to make any changes which can reasonably be done automatically. On the whole, they work quite well, but they are not infallible and ought to be used with human supervision. Details of the exact capabilities and behaviour of each of them is given in the command descriptions in Appendix A.

In addition, for each of the source code modifying filters `inscnf`, `frepint` and `crepint`, a driver script is provided. For converting large numbers of files, this is likely to be the most convenient

way to proceed. Each driver script runs the filter on a given set of files, summarises any warnings, and writes any files that have been changed in a new directory. Any files which did not need alteration are not rewritten. It also performs some crude safety checks that the modified source files seem to have been changed in the right way (for instance, that no changes have been made except for the ones which should have been made, and that changes to Fortran code do not result in lines longer than 72 characters). These checks do not do such careful parsing of the source code as the conversion filters themselves, so they can throw up false positives or false negatives, but they give an extra level of confidence.

The driver script for `inscnf` is called `do-inscnf` and is invoked with the files to be converted as command line arguments. For each of its command line arguments it runs `inscnf` and

- It writes a line to a log file `./inscnf.log` saying how many lines were changed. This file does not contain anything requiring attention which is not printed to standard output.
- If and only if some changes were made, it writes the modified copy of the file, under the same name, into a directory called `./inscnf.changed`.
- If the filter or the driver script thinks there might be something which needs human attention, it writes a suitable message to standard output. These warning messages can be understood in conjunction with the documentation of the filter, or of the corresponding discussion in section 3.
- It reports a brief summary to standard output of its activities, indicating how many files have changed, and whether any files have gained new dependencies on include files.

Finally it writes a short summary of the run. An excerpt of its output might look like this:

```
% do-inscnf *.f *.gen
astimp.f:          2 x Last arg %VAL in FTS1_GKEYD
import.f:          Integer %VAL arg in CCD1_IMFIT?
import.f:          Last arg %VAL in CCD1_IMFIT
...
71 new dependencies on include file 'CNF_PAR'.
71/483 modified files written in ./inscnf.changed.
Logfile is ./inscnf.log.
```

The other driver scripts, `do-crepint` and `do-frepint`, differ in the exact warnings they give, but otherwise work the same way as `do-inscnf`.

The driver script may occasionally throw out a comment like this:

```
ndf1_pshdt.f:      Possible error in modifications?
```

which indicates that there seems to be a difference between the output of the filter and the input plus the expected changes. The checking is done by a quick-and-dirty Perl script called `cmp-inscnf` (and similarly for the other filters), and if you see this warning you can run this script yourself to get an idea what the problem may have been. The above warning was emitted by `do-frepint`; investigate as follows:

```
% cmp-frepint ndf1_pshdt.f
*** Edit error near line 69?:
CALL NDF1_SPLDT( STR, 1, LEN( STR ), ' ', -10, F, L, NFIELD, STATUS )
CALL NDF1_SPLDT( STR, 1, LEN( STR ), ' ', - 10, F, L, NFIELD, STATUS )
```

from which it is clear that the culprit is some harmlessly inserted whitespace. Note that in order to make these comparisons, the `cmp-*` scripts deal in a cavalier fashion with formatting, so the source lines may not look very much like the original copies and the line number reporting may not be accurate, but the message should be sufficient to locate the problem. Such “Possible error in modifications?” warnings ought to be infrequent, and may often not be indicative of a real problem, but it is as well to investigate them.

Detailed descriptions of the behaviour of the filter programs is given in appendix A

5 EXTREME cribsheet: Converting for extreme data sets in 99 easy steps

This section gives a list of what you should do to a Starlink package to enable it to cope with large images. Many of the steps can be applied equally to modifying non-USSC code which makes use of the USSC. The steps here give an indication of how to proceed, but for more discussion of what’s going on, refer to other sections in this document. Some examples of the commands you could use are given, but obviously you can change these to personal taste.

- (1) Identify all C source files and all Fortran source files (the listing of package files given by the Source Code Browser can be useful for this).

```
% tar xf package_source.tar
% echo >ffiles *.f *.gen pkg1_par pkg1_err
% echo >cfiles *.c *.h
```

- (2) Start up the EXTREME package

```
% extreme

EXTREME commands are now available -- Version 0.1-0
```

- (3) Run `do-inscnf` on Fortran files to wrap `%VAL` arguments in `CNF_PVAL` calls, if this has not already been done.

```
% do-inscnf 'cat ffiles'
pkg1_sub.f:          Constant %VAL arg in PKG1_ADD?
5 new dependencies on include file 'CNF_PAR'.
5/25 modified files written in ./inscnf.changed.
Logfile is ./inscnf.log.
```

- (4) Check `CNF_PVAL` insertions which generated warnings, and remove them if they wrap arguments which are not registered pointers.

```
% vi inscnf.changed/pkg1_sub.f
```

- (5) Rebuild source archive, rebuild package, and test.

```
% mkdir retar
% cd retar
% tar xf ../package_source.tar
% cp ../inscnf.changed/* .
```

```

% tar cf ../package_source.tar *
% cd ..
% rm -r retar inscnf.changed
...

```

- (6) Ensure that you have 64-bit copies of all the Starlink libraries on which the package depends (these are not required for making the changes, but will be needed for building and testing the modified package).
- (7) Run `do-frepint` on Fortran files to replace `INTEGER` declarations by `INTEGER * 8`.

```

% tar xf package_source.tar 'cat ffiles'
% do-frepint 'cat ffiles'
pkg1_mult.f:          INTEGER*2 declaration not changed
pkg1_divid.f:         No IMPLICIT NONE in module PKG1_DIVID
pkg1_misc.f:         Nowhere to declare INT_'s in module PKG1_MISC

22/25 modified files written in ./frepint.changed.
Logfile is ./frepint.log

```

- (8) Check constructs which led to warnings and make any necessary edits, following the discussion in section 3.2:

Missing IMPLICIT NONE: An `IMPLICIT NONE` statement should really be inserted into the module in question and explicit declarations made for all the variables used. If this is impractical, the only essential change is to make sure that all the implicit `INTEGER` variables are explicitly declared as `INTEGER * 8`. Note it is *not* sufficient to make an `IMPLICIT INTEGER*8 (I–N)` declaration, since the resulting declarations would not be edited at build time by the makefile machinery described in section 3.4.

EQUIVALENCE statements: Ensure that none of the memory associations are between `INTEGER` and some other type. If they are, careful recoding will probably be required.

Use of INTEGER specific intrinsic functions: If possible these should be changed to the corresponding generic names of the intrinsic functions. If this is not possible some other workaround may be required.

INTEGER*n declarations left unchanged: These probably require no further modification.

Nowhere to declare INT_'s: This indicates that literal integer constants were found as (apparently) actual arguments to subroutine calls and replaced by constants, but `frepint` couldn't find a suitable place in the module to put the `PARAMETER` statements (it looks for an `INCLUDE` statement). In this case you will have to insert the lines yourself, but running `frepint` will tell you what it would have inserted, so that you can paste it in by hand.

```

% frepint pkg1_misc.f >/dev/null
frepint: Nowhere to declare INT_'s in module PKG1_MISC

* Local constants for use as actual arguments:
  INTEGER * 8 INT__0
  INTEGER * 8 INT__12
  PARAMETER ( INT__0 = 0 )
  PARAMETER ( INT__12 = 12 )

```

- (9) If there is any tricky storage association involving integers, such as use of COMMON or EQUIVALENCE, or a formal argument of different type to an actual argument, you may need to recode this. Hopefully such things should be few and far between.
- (10) If the package calls any unconverted libraries (e.g. non-standard intrinsic functions), change the types of actual arguments in affected calls back to INTEGER.
- (11) It is now possible, though not normally worth the effort, to change any private INTEGER*8 variables back to INTEGERS for efficiency or other reasons.
- (12) Prepare for C source conversion: ensure that all functions are properly declared using ANSI C-style formal argument type declarations. If you think this may not be the case, compiler warnings may be of help.

```
% gcc -fsyntax-only -Wimplicit-function-declaration -Wstrict-prototypes *.c
tmp.c:160: warning: implicit declaration of function 'pipe'
% man pipe
...

```

- (13) Make sure that communication with Fortran routines is done properly using CNF as described in SUN/209. In particular it is important to use the TRAIL macro for passing character string lengths.
- (14) Run do-crepint on C files to replace references to int type by INT_BIG.

```
% tar xf package_source.tar 'cat cfiles'
% do-crepint 'cat cfiles'
input.c:                2 x Format string implies int * (comment inserted)
output.c:                5 x Format string has %[cdiouX*] (comment inserted)
output.c:                Format string non-literal (comment inserted)
toplevel.c:              Non-stdlib system header file <unistd.h>
toplevel.c:              Type of main not changed from int
toplevel.c:              Type of argc not changed from int

8/8 modified files written in ./crepint.changed.
Logfile is ./crepint.log.

```

- (15) Check constructs which led to warnings and make any necessary edits, following the discussion in section 3.3:

main or argc declaration unchanged: Check that the items in question are supposed to be int and not just called main or argc by coincidence.

Format strings: These will mostly require edits as shown in the examples and discussion in section 3.3. Positions of the warned-about calls are marked in the source code by 'crepint:' comments.

signal, bsearch and qsort calls: These calls do not need modification, but make sure that the functions which they pass are declared and defined using int type not INT_BIG.

Inclusion of non standard-library system header files: These presumably indicate use of non standard-library system libraries; the calls to such functions should be identified, and any which take int * arguments or variable argument lists should be treated in a similar way to scanf and printf. You can see which calls are made to the functions declared in a header file by removing the include directive and checking compiler warnings, something like:

21 SSN/73.2 (draft) —EXTREME cribsheet: Converting for extreme data sets in 99 easy steps

```
% grep -v '<unistd.h>' toplevel.c >tmp.c
% gcc -fsyntax-only -Wimplicit-function-declaration tmp.c
```

- (16) Check that you've attended to all the Format string warnings and removed the flagging comments.

```
% grep 'crepint:' crepint.changed/*
```

- (17) Have a look for implicit function or argument declarations and replace them by explicit INT_BIG declarations. Certain flags of gcc can be used to pick this up:

```
% gcc -fsyntax-only -Wimplicit-int -Wstrict-prototypes *.c
```

or the -proto flag of the Tru64 Unix C compiler:

```
% cc -noobject -protois -DINT_BIG=long crepint.changed/*.c
% grep '[^a-z]int[^a-z]' *.H
```

- (18) It is now possible, though not normally worth the effort, to change any private INT_BIG variables back to ints for efficiency or other reasons.

- (19) Rebuild the source archive.

```
% mkdir retar
% cd retar
% tar xf ../package_source.tar
% cp ../frepint.changed/* .
% cp ../crepint.changed/* .
% tar cf ../package_source.tar *
% cd ..
% rm -r retar frepint.changed crepint.changed
```

- (20) Use extmk to make the necessary modifications to the mk file.

```
% mv mk mk.orig
% extmk < mk.orig > mk
% chmod 755 mk
```

- (21) If extmk generated any warnings, edit the mk file by hand accordingly.

- (22) If there are platform-dependent files in the package you must either add

```
SOURCE_VARIANT='alpha_OSF1'
```

and

```
SOURCE_VARIANT='sun4_Solaris'
```

to the new alpha_OSF1_64 and sun4_Solaris_64 stanzas of the mk file respectively, or generate new platform-dependent files for the two new platforms.

- (23) Use extmakefile to make some of the necessary modifications to the makefile, and check the changes made.

```
% mv makefile makefile.orig
% extmakefile < makefile.orig >makefile
% diff makefile.orig makefile
```

- (24) If `extmakefile` indicated that it failed to make any of the changes that it wanted to, insert the relevant bits of code by hand with reference to section 3.4.
- (25) Edit the `makefile` by hand to make the rest of the necessary changes as discussed in 3.4:
- Ensure that all, and only, extractions of Fortran source files from source archives are processed using the `REPLACE_INTEGER8` macro (replace the new dummy target `$(FORTRAN_FILES)` with macros representing the Fortran files to be extracted, and check for any other use of `TAR_OUT` to extract fortran files, e.g. in the 'do_test' target). (search for uses of `TAR_OUT`).
 - Add rules for building any new include files required by the changes (`CNF_PAR`, `EXT_PAR` or `extreme.h`).
 - Add any new dependencies of object files on these include files.
 - Fix, or remove, any private archive-type targets which have been broken by the fact that source files in the build directory are no longer original copies.
- (26) Export the source, and build the package. Look out especially for the following warnings:
- Fortran variable misaligned in `COMMON` block.
 - C pointer type mismatch.
 - Incompatible arguments.

If there are compilation errors or new warnings fix them. In general, `sun4_Solaris_64` is more sensitive to problems than `alpha_OSF1_64`, since the sizes of `long` and pointers are actually different between the 32-bit and 64-bit compilations.

- (27) Test! Bear in mind that the changes may have broken the package on the old platforms as well as the new ones.

A Description of tools

crepint

Replace int by INT_BIG in C

Description:

This program is a filter which takes C source code and replaces occurrences of the type specifier 'int' by the identifier 'INT_BIG'. This identifier can then be assigned a preprocessor value of a suitable integral type (int or long) either using an include file or with a -DINT_BIG=(type) flag on the C compiler command line.

It's not quite as simple as replacing every semantically significant occurrence of the 'int' identifier; 'short int' and 'long int' type specifiers will be left alone.

If a use of int appears to be declaring a symbol called 'main' or 'argc', then this will be left alone too, and a warning written to standard error to the effect that it is not being changed.

Additionally, references to the <limits.h> macros INT_MAX, INT_MIN and UINT_MAX are replaced by INT_BIG_MAX, INT_BIG_MIN and UINT_BIG_MAX respectively. If any of these substitutions are made, then a line '#include "extreme.h"' is added after the '#include <limits.h>' line which is presumably in the file. If <limits.h> is not included in the input file, a warning is written to standard error.

Explicit declarations which are implicitly of type int will have an INT_BIG token inserted - for instance 'static x, y;' will be changed to 'static INT_BIG x, y;'.

The program will write a warning on standard error for certain constructions in the code which are likely to cause trouble after the mass redeclaration of int as INT_BIG has occurred, since in some places the type int, and not INT_BIG, is still required. These constructions are:

- Inclusion of system header files other than those of the C standard library, since these may indicate use of functions other than those warned about above with arguments of type pointer to int.
- Use of functions from the C standard library which may require changes.

The functions from the C standard library which may require changes are the following:

- Format strings in formatted I/O which may need changes because they use variable argument lists or require arguments of type pointer to int.
- The frexp() math function whose second argument must be a pointer to int
- The signal() function whose second argument is a function which must take an int argument
- The bsearch() and qsort() functions which take a comparison function as argument, and this function must be of type int

In the case of potentially dangerous format strings, for convenience a comment is inserted in the output code on the line before the function call is made. The comment will contain the character string 'crepint: '. The warning to standard error notes that the comment line has been inserted.

The following constructions are also likely to cause trouble, but will not be warned about by the program:

- Use of functions without prototypes. If header files are omitted or old style function declarations are used then the ANSI C machinery for doing type conversion at function call time will not work. Gcc's '-Wstrict-prototypes' and '-Wimplicit-function-declaration' flags are useful for this.
- Implicit declarations, which are implicitly of type int. If a name is declared simply by mentioning it without any type or type qualifiers, it is implicitly of type int, and so should become declared as INT_BIG. This program does not find these. Such implicit declarations (only?) occur in function declarations. The Tru64 Unix C compiler's '-protois' flag or gcc's '-Wimplicit-int' flag are useful for identifying these.

The program tries to adjust padding whitespace outside comments so that the spacing of the output looks OK.

No changes are made to comment lines so that, for instance, the Synopsis stanza of function prologues will not have formal argument types changed from 'int' to 'INT_BIG' .

Source code which makes sufficiently inventive use of the C will stand a good chance of confusing this program.

Usage:

```
crepint [ in [ out ] ]
```

Notes:

Although this program behaves as a filter, it is written on the assumption that it will be run on a file of a finite length: it may buffer large amounts of input before writing output, and it may not free up memory.

frepint

Replace INTEGER by INTEGER*8 in Fortran 77

Description:

This program is a filter which takes FORTRAN 77 source code and modifies it so that INTEGER declarations are rewritten as 'INTEGER * 8' . It also attempts to warn if there are usages which might cause trouble given this change.

Additionally, if there appear to be actual arguments to subroutines or functions which are literal integers, the program will attempt to replace them with symbolic constants, and to define these constants in the declaration section of the module. Thus the line

```
CALL SUB( X, 5, STATUS )
```

will be replaced by

```
CALL SUB( X, INT__5, STATUS )
```

and the corresponding declaration statements

```
INTEGER * 8 INT__5
```

and

```
PARAMETER ( INT__5 = 5 )
```

will be inserted in the declaration section of the module. The program will attempt to insert these declarations near an INCLUDE statement, and if it cannot find one, it will write a warning to standard error, including the text of the declarations that it would have made.

Attention is paid to fortran 77 source format, so that lines more than 72 characters long are avoided (unless they were there in the first place).

Some attention is paid to the aesthetic qualities of the output: line breaks are made, where possible, following the usage in, e.g., KAPPA. An attempt is made to copy the style of case usage from the input.

No changes are made to comment lines so that, for instance, the Arguments stanza of subroutine prologues will not have argument types modified from 'INTEGER' to 'INTEGER * 8' .

No change is made to references to INTEGER type in IMPLICIT statements.

The program will write a warning on standard error for certain constructions in the code which are likely to cause trouble after the mass redeclaration of INTEGER as INTEGER*8 has occurred. These constructions are:

- INTEGER * n declarations which already exist in the code (these are not modified)
- EQUIVALENCE statements
- Use of INTEGER Specific names for standard intrinsic functions (IABS, ISIGN, MAX0, AMAX0, MIN0, IMIN0). IDIM could also go in this list, but since it is a common variable name, and an uncommon intrinsic, no warning is given for IDIM.

- Any module (SUBROUTINE, FUNCTION or BLOCK DATA) which does not include an IMPLICIT NONE statement.

Usage:

```
frepint [ in [ out ] ]
```

Notes:

The program is not infallible at identifying function calls, which it needs to do in order to replace integer literals, since they look like array references. It uses the rule of thumb that if the would-be function name contains an underscore it is a function, otherwise it is an array.

It will also not identify an INTEGER-type expression as such unless it is a single integer literal; for instance the expression '3 * 5' as actual argument of a subroutine/function ought to be retyped, but will not be spotted.

In a few cases, the line breaks are not made in very beautiful places. They should, however, always be correct.

Although this program behaves as a filter, it is written on the assumption that it will be run on a file of a finite length: it may buffer large amounts of input before writing output, and it may not free up memory.

inscnf

Wrap %VAL arguments with CNF_PVAL in Fortran 77

Description:

This is a filter which takes FORTRAN 77 source code and modifies it so that text which is the argument of a %VAL directive is wrapped in a call to CNF_PVAL; i.e. input text

```
%VAL( IPTR )
```

is changed to

```
%VAL( CNF_PVAL( IPTR ) )
```

If the call to CNF_PVAL is already present no change is made, and a warning is emitted. Lines with no references to the %VAL directive are left alone.

Additionally, for each program unit in which a call to CNF_PVAL has been made, an attempt is made to insert a line like

```
INCLUDE ' CNF_PAR' ! For CNF_PVAL function
```

This is inserted after the last INCLUDE line which already exists in the program unit. If there are no INCLUDE lines there, this line is not inserted, and a warning message is printed to standard error.

Attention is paid to Fortran 77 source format, so that lines more than 72 characters long are avoided (unless they were there in the first place).

Characters ' \r' (carriage return) and ' \t' (tab) might possibly cause erroneous line breaking - if any are encountered a warning is given (these should not be in the source really). Code using columns 73-80 of the source cards for comments is likely to be mangled (nobody does this any more do they?).

Under certain improbable circumstances it is possible for the program to get stuck trying to break a line; in this case it will exit with error status and an error message.

Some attention is paid to the aesthetic qualities of the output: line breaks are made, where possible, following the usage in, e.g., KAPPA. An attempt is made to copy the style of case usage and bracket spacing from the input.

This program wraps ALL occurrences of %VAL in a call to CNF_PVAL, unless they are already so wrapped. If it suspects that %VAL may not be a legitimate candidate for this treatment, it will output a warning message to standard error. It will do this in the following cases:

- %VAL is on the last argument in the argument list (this suggests that it might be a trailing string length)
- The argument of %VAL looks like an integer constant
- The argument of %VAL looks like a Starlink-style symbolic constant (has two adjacent underscore characters).

Usage:

```
inscnf [ in [ out ] ]
```

Notes:

Although this program behaves as a filter, it is written on the assumption that it will be run on a file of a finite length: it may buffer large amounts of input before writing output, and it may not free up memory.

extmk

Modify mk file for use with EXTREME data sets

Description:

This is a simple filter which takes a Starlink mk file as input and writes as output a file which should provide the additional functionality required for EXTREME use, i.e. it adds new SYSTEM types of `alpha_OSF1_64` and `sun4_Solaris_64`, provides for use of the `INTEGER8` macro to preprocess Fortran source code before compilation, and modifies the C compilation flags for use with the `INT_BIG` macro.

The changes it attempts to make are as follows:

- Add comment blocks describing the two new values of SYSTEM
- Modify the comment describing the default value of CFLAGS
- Add a comment describing the `INTEGER8` environment variable
- Export the `INTEGER8` environment variable
- Modify SYSTEM-specific settings for existing SYSTEMs
- Add support for the two new SYSTEMs

If it is unable to make any of the modifications it would like to, it will print an explanatory message saying what it could not do.

The only change which might be required which this script does not make is to set values for the `SOURCE_VARIANT` variable. Depending on how the system-specific compilation is going to be done, it may be desirable to add `SOURCE_VARIANT='alpha_OSF1'` in the new `alpha_OSF1_64` stanza, and `SOURCE_VARIANT='sun4_Solaris'` in the new `sun4_Solaris_64` stanza. If required, this must be done by hand.

This script is not extremely intelligent - unless the mk file follows the usual pattern quite closely, the output may be in error, or complete rubbish. It is intended as a convenience, and if it does not work, then it will be necessary to make the modifications by hand; this is not too arduous.

Usage:

```
extmk < mk > mk.new
```

extmakefile

Modify makefile for use with EXTREME data sets

Description:

This is a simple filter which takes a Starlink makefile as input and writes as output a file which provides some of the changes required for EXTREME use. Not all the requisite changes can be made however, and the resulting file will need some edits to be made by hand.

The changes it attempts to make are as follows:

- Add ' -DINT_BIG=int' to the default CFLAGS macro
- Add ' INTEGER8 = INTEGER' default macro setting
- Add definition of the REPLACE_INTEGER8 macro
- Add a rule for extracting Fortran source files from source archive
- Add code for recording the INTEGER8 macro in the datestamp file

If it is unable to make any of the modifications it would like to, it will print an explanatory message saying what it could not do.

Various changes will still need to be made by hand after generating a new makefile by running this script. The principal one is splitting files between Fortran and non-Fortran ones, so that the non-Fortran files can be extracted using the old rule, but the Fortran files can be extracted using the new rule inserted by this script (the dummy target ' \$(FORTRAN_FILES)' is written as a placeholder for such files). This may require splitting up existing macros, for instance splitting PRIVATE_INCLUDES into PRIVATE_C_INCLUDES and PRIVATE_F_INCLUDES.

This script is not extremely intelligent - unless the makefile follows the usual pattern quite closely, the output may be in error. It is strongly advised that the input and output are compared using diff. If there are problems, it may be necessary to make the modifications by hand; this is not too arduous.

Usage:

```
extmakefile < makefile > makefile.new
```

cmp-xxx

Compare filtered source with original for unwanted changes

Description:

This script checks an original source code file against a version of it run through the one of the source code filter programs in the EXTREME package. It should be invoked under the name 'cmp-xxx' to check the effects of the filter 'xxx' .

The available filters xxx are:

- *inscnf* – Wrap %VAL invocations in CNF_PVAL calls
- *crepint* – Change int type to INT_BIG in C
- *frepint* – Change INTEGER to INTEGER * 8 in Fortran

If a file containing the filtered source code already exists, it can be given as the second command-line argument, otherwise the file given as the first command-line argument will be run through the appropriate filter before the comparison is made.

The script uses its knowledge of the changes made by each of the filters to identify changes which are unexpected. The idea is to serve as a check that the the source code filter program itself has not made a mistake. The parsing done by this script is much less careful than that done by the filters, so it may make mistakes - apparent discrepancies reported by this script should be investigated, but may well not indicate genuine problems.

If an apparent discrepancy is found, the script prints the first line from each file in which a discrepancy exists. The text printed is not the text as it appears in the files, but a version mangled by this program for comparison - whitespace, formatting, and other things may have been changed. The printed text should however make it easy enough to find the relevant point in the input file.

If there is an apparent discrepancy the exit status of the program will be non-zero; if the files match then there is no output and exit status is zero.

Usage:

```
cmp-xxx file [ newfile ]
```

do-xxx

Apply source code filter xxx to a set of files

Description:

This is a driver script to run one of the filter programs in the EXTREME package which modifies source code. It should be invoked under the name 'do-xxx' to run the filter 'xxx'

The available filters xxx are:

- `inscnf` – Wrap %VAL invocations in CNF_PVAL calls
- `crepint` – Change int type to INT_BIG in C
- `frepint` – Change INTEGER to INTEGER * 8 in Fortran

For each of its command line arguments (files) it:

- Runs the filter xxx.
- Writes a line to the file `./xxx.log` saying how many lines were changed.
- If and only if changes were made, it writes the modified file under the same name into the directory `./xxx.changed`.
- If the filter wrote any warnings to standard error, these are summarised to standard output.
- Checks that the input file and output file don't look different in any unexpected ways, and writes a warning if they do.
- For Fortran files, checks that no lines have been extended beyond 72 characters (this should not be possible, it's just an extra precaution).

It then writes a short summary of the run, including a report of whether any files gained new dependencies on include files. Thus anything which the filter or this script thinks may require human attention is written to standard output while modified files are written to a new directory.

As part of its checking, it invokes a script called `cmp-xxx` on each pair of (original, changed) files; these scripts are crude checks that the changes made have not done anything to the source code to change its intention other than what should have been done. They have been written in a fairly quick and dirty way however, and may well turn up false positives or false negatives.

Usage:

`do-xxx files`