SUN/7.6

Starlink Project
Starlink User Note 7.6

S M Beard
P M Allan
Malcolm J. Currie
Peter W. Draper
David Berry

2020 Feb 12

GENERIC — A Utility for Preprocessing Generic Fortran and C Subroutines 1.4 User's Guide

Abstract

GENERIC is tool for Fortran 77 and C developers to create type-specific code from a generic source that includes tokens for type-dependent elements. This saves effort supporting multiple versions of the source code.

Contents

1 Introduction

GENERIC is a utility which preprocesses a generic Fortran or C subroutine into its different types and concatenates these routines into a file. The file can then be compiled with the appropriate compiler to produce an object module.

It also supports standardized C generic programming by providing a suitable set of include files and rules for using them

2 GENERIC Fortran Subroutines—an Example

A generic subroutine is one which has an argument (or arguments) which may be one of a number of different Fortran data types.

As an example, consider a subroutine which initialises every element in an array to zero. One might want a different version of the routine for initializing a double precision array, an integer array and a real array. The following subroutines would need to be written:

```
ZEROD( N, DARR )   Zero every element in a DOUBLE PRECISION array
ZEROI( N, IARR )   Zero every element in an INTEGER array
ZEROR( N, RARR )   Zero every element in a REAL array
```

The integer routine might contain the following code (in a file called `zeroi.f`):

```
      SUBROUTINE ZEROI( N, ARRAY )
*+
*  Name:
*     ZEROI
*  Purpose:
*     Zero all the elements of an4 INTEGER array
*  Invocation :
*     CALL ZEROI( N, ARRAY )
*  Description :
*     This sets all the elements of an INTEGER array to zero
*  Arguments:
*     N = INTEGER( Given )
*        Dimension of array
*     ARRAY( N ) = INTEGER( Given and Returned )
*        The array to be zeroed
*                   "
*     (the rest of the standard prologue)
*                   "
*  Type Definitions:
      IMPLICIT NONE
*  Arguments Given:
      INTEGER N
*  Arguments Given and Returned:
      INTEGER ARRAY( N )
```

```
*  Local Variables:
      INTEGER I                   ! Array index
*-

      DO I = 1, N
         ARRAY( I ) = 0
      END DO
      END
```

The REAL and DOUBLE PRECISION versions of the routine would have to be written in a similar way, and this obviously involves a lot of duplication of effort. The situation gets even worse if other versions of the routine (such as BYTE, INTEGER*2) are needed. Also, it is very difficult and tedious to keep all the versions in step if they have to be edited individually when the routines are changed.

The GENERIC utility is a labour-saving device which enables all the various types of routines to be constructed automatically from one master routine. If any changes become necessary, only this one master routine needs to be edited.

To use the GENERIC utility, the routine listed above is replaced by the following (in a file called `zero.gen`):

```
      SUBROUTINE ZERO<T>( N, ARRAY )
*+
*  Name:
*     ZERO<T>
*  Purpose:
*     Zero all the elements of a <COMM> array
*  Invocation :
*     CALL ZERO<T>( N, ARRAY )
*  Description :
*     This sets all the elements of a <COMM> array to zero
*  Argument:
*     N = INTEGER( Given )
*        Dimension of array
*     ARRAY( N ) = <TYPE>( Given and Returned )
*        The array to be zeroed
*                    "
*     (the rest of the standard prologue)
*                    "
*  Type Definitions:
      IMPLICIT NONE
*  Arguments Given:
      INTEGER N
*  Arguments Given and Returned:
      <TYPE> ARRAY( N )
*  Local Variables:
      INTEGER I                  ! Array index
*-

      DO I = 1, N
         ARRAY( I ) = 0<CONST>
      END DO
      END
```

This is a "generic routine." Other examples of generic routines include the DAT_ routines in the Hierarchical Data System (SUN/92) and the PAR_ routines of the ADAM parameter system (SSN/29).

The items in angle brackets, <> , are "tokens" which the GENERIC utility replaces when it converts the routine into one of a given type. In the above example, the tokens would be replaced as follows.

Token	Double precision replacement	Integer replacement	Real replacement
<T>	D	I	R
<TYPE>	DOUBLE PRECISION	INTEGER	REAL
<COMM>	DOUBLE PRECISION	INTEGER	REAL
<CONST>	.0D0	(blank)	.0E0

The types are generated by issuing the following command.

```
% generic -t dir zero.gen
```

More details on the use of the GENERIC utility, and a full list of tokens and types, are given in the rest of this document.

2.1 Using the GENERIC utilty to process generic C files

If the source files supplied to the GENERIC utility have a file type that starts with "`.c`", then they are assumed to contain C source code for functions that are designed to be called from Fortran using the macros in the `f77.h` header file provided by the CNF package. The same rules apply regarding the interpretation of the file types as for Fortran source code, except that "`.cgen`" is used in place of "`.gen`" and the initial "`.g`" used in other Fortran files type (for instance "`.gdr`") becomes "`.c`" (for instance "`.cdr`"). The output text is written to a file with type "`.c`" rather than "`.f`", and no text wrapping is performed in the output files.

An example generic C source file is shown below (`ccg8_um3.cdr` from KAPLIBS):

```
#include <stdint.h>
#include "sae_par.h"
#include "prm_par.h"
#include "f77.h"

F77_SUBROUTINE(ccg8_um3<TLC>)( INTEGER8(NPIX), INTEGER8(NLINES),
                               <CNFTYPE>_ARRAY(STACK), INTEGER8(MINPIX),
                               <CNFTYPE>_ARRAY(RESULT), DOUBLE_ARRAY(NCON),
                               INTEGER8(NBAD), INTEGER(STATUS) ){
/*
*+
*  Name:
*     CCG8_UM3<T>

*  Purpose:
```

```
*     Combines data lines using an unweighted mean.

*  Language:
*     C (designed to be called from Fortran 77)

*  Invocation:
*     CALL CCG8_UM3<T>( NPIX, NLINES, STACK, MINPIX, RESULT, NCON,
*                       NBAD, STATUS )

*  Description:
*     This routine accepts an array consisting a series of (vectorised)
*     lines of data.  The data values in the lines are then combined to
*     form an unweighted mean.  The output means are returned in the
*     array RESULT.

*  Arguments:
*     NPIX = INTEGER*8 (Given)
*        The number of pixels in a line of data.
*     NLINES = INTEGER*8 (Given)
*        The number of lines of data in the stack.
*     STACK( NPIX, NLINES ) = <TYPE> (Given)
*        The array of lines which are to be combined into a single line.
*     MINPIX = INTEGER*8 (Given)
*        The minimum number of pixels required to contribute to an
*        output pixel.
*     RESULT( NPIX ) = <TYPE> (Returned)
*        The output line of data.
*        to the output line.
*     NCON( NLINES ) = DOUBLE PRECISION (Returned)
*        The actual number of contributing pixels from each input line
*        to the output line.
*     NBAD = INTEGER*8 (Returned)
*        The number of bad values in the output array created while
*        forming the statistics.  It excludes those bad values whose
*        corresponding values along the collapse axis are all bad.
*     STATUS = INTEGER (Given and Returned)
*        The global status.

*-
*/

/* Local Variables: */
   double sumd;          /* Sum of values */
   double sumw;          /* Sum of weights */
   double value;         /* Output value */
   int64_t i;            /* Loop variable */
   int64_t j;            /* Loop variable */
   int64_t ngood;        /* Number of good pixels */
   const <CTYPE> *pstack; /* Pointer to next input value */

/* Initialise returned values. */
   *NBAD = 0;
   for( i = 0; i < *NLINES; i++ ){
      NCON[ i ] = 0.0;
   }

/* Check inherited global status. */
   if( *STATUS != SAI__OK ) return;

/* Loop over for all possible output pixels.  */
   for( i = 0; i < *NPIX; i++ ){

/* Initialise sums. */
      sumw = 0.0;
      sumd = 0.0;

/* Set good pixel count. */
      ngood = 0;

/* Loop over all possible contributing pixels forming weighted mean
   sums. */
      pstack = STACK + i;
      for( j = 0; j < *NLINES; j++ ){
         if( *pstack != VAL__BAD<T> ) {

/* Conversion increment good value counter. */
            ngood++;

/* Sum weights. */
            sumw += 1.0;

/* Form weighted mean sum. */
            sumd += (double)( *pstack );

/* Update the contribution buffer---all values contribute when forming
   mean. */
            NCON[ j ] += 1.0;

/* Move the input pointer on to the value in the next input line. */
         pstack += *NPIX;
      }

/* If there are sufficient good pixels output the result. */
      if( ngood >= *MINPIX ) {
         value = sumd/sumw;
         RESULT[ i ] = (<CTYPE>) value;

/* Trap numeric errors. */
         if( RESULT[ i ] != (<CTYPE>) value ) {
            RESULT[ i ] = VAL__BAD<T>;
            (*NBAD)++;
         }

/* Not enough contributing pixels, set output invalid unless all of them
   are bad. */
      } else {
         RESULT[ i ] = VAL__BAD<T>;
         if( ngood > 0 ) (*NBAD)++;
```

```
            }
        }
    }
```

The above file could be processed using tyhe command:

```
    % generic ccg8_um3.cdr
```

to create output file `ccg8_um3.c` containing expanded versions of the above code for HDS data types _DOUBLE and _REAL.

3 Generic C programming

Two tools are provided for generic C programming. Which one to use depends on the nature of the C source code:

- If the C source code contains functions that are designed to be called from Fortran using the macros in the `f77.h` header file provided by the CNF package, then the same GENERIC utility should be used that is used to process Fortran source file (described in the previous section).
- Otherwise the recipe described in the rest of this section should be used. Note that in this case we use the C preprocessor to create the necessary code, not the GENERIC utility. Clearly there are many ways to do this task, what is presented here is just a standardized method tuned for working with Starlink libraries.

First create your generic code routines. These actually go into an include file that is conventionally named with extension `.cgen`. There are several macros defined for use when defining generic functions.

- `CGEN_BAD`
- `CGEN_FUNCTION`
- `CGEN_HDS_TYPE`
- `CGEN_MAX`
- `CGEN_MIN`
- `CGEN_PRM_TYPE`
- `CGEN_TYPE`

You need to use `CGEN_FUNCTION` as part of the normal function declaration so that generic forms of the function name can be generated on inclusion. Each routine in a generic include file should start:

```
      return_type CGEN_FUNCTION(function_name) ( arg decs )
```

So a routine called `kpg1_bad` that returned an int and accepted a `CGEN_CTYPE` pointer, called `value`, would be defined as:

```
          int CGEN_FUNCTION(kpg1_bad) (CGEN_CTYPE *value) {
              if ( value[0] == CGEN_BAD ) {
                 return 1;
              }
              return 0;
          }
```

with the trivial job of testing the first element of an array against the BAD value constant.

The value of `CGEN_TYPE` will be set to the C type, that is `double`, `float`, `int`, `short int`, `unsigned short int`, `char` and `unsigned char`, as appropriate.

The value of `CGEN_BAD` will be set to one of the PRIMDAT constants `VAL__BADD`, `VAL__BADR`, `VAL__BADI`, `VAL__BADW`, `VAL__BADUW`, `VAL__BADB` or `VAL__BADUB` as appropriate, and the value of `CGEN_HDS_TYPE` will be set to the HDS type, one of `_DOUBLE`, `_REAL`, `_INTEGER`, `_WORD`, `_UWORD`, `_BYTE` or `_UBYTE`.

Now create a C file to includes the generic code. The include file is included once for each of the data types you want to support (this file can also contain related non-generic code). To do this define the macro `CGEN_CODE_TYPE` to be one of the values:

```
          CGEN_DOUBLE_TYPE, CGEN_FLOAT_TYPE, CGEN_INT_TYPE, CGEN_WORD_TYPE,
          CGEN_UWORD_TYPE, CGEN_BYTE_TYPE, CGEN_UBYTE_TYPE
```

which selects the data type. Then include the file `cgeneric_defs.h` followed by your generic include file, (called `mygenerics.cgen` in the following example):

```
      #include <prm_par.h>
      #include <cgeneric.h>

      #define CGEN_CODE_TYPE CGEN_DOUBLE_TYPE
      #include "cgeneric_defs.h"
      #include "mygenerics.cgen"
      #undef CGEN_CODE_TYPE

      #define CGEN_CODE_TYPE CGEN_INT_TYPE
      #include "cgeneric_defs.h"
      #include "mygenerics.cgen"
      #undef CGEN_CODE_TYPE
```

when compiled this will generate code for `double precision` and `integer` versions of the generic functions defined in `mygeneric.cgen`.

When you want to call the generic forms of a function, the name you supply to the `CGEN_FUNCTION` macro has one of character codes `D`, `F`, `I`, `W`, `UW`, `B`, and `UB` appended. So for our example `kpg1_bad`, we have the two functions `kpg1_badD` and `kpg1_badI`.

See the comments in the include files for descriptions of the other macros.

4 Features of the GENERIC Utility

4.1 The KERNEL Tokens

The following tokens are recognised by the GENERIC utility:

`<T>` To be replaced by a single character representation of the specified type.

`<TYPE>` To be replaced by the string used to declare variables of the specified type when they are passed as arguments to the subroutine.

`<LTYPE>` To be replaced by the string used to declare variables of the specified type when they are defined as local variables in the subroutine.

`<CONST>` To be replaced by the string appended to numeric constants of the specified type.

`<HTYPE>` To be replaced by the HDS name for the specified type.

`<COMM>` To be replaced by a comment describing the specified type in plain English.

A full table of the translation of these tokens for each variable type is given in Appendix A at the end of this document.

4.2 The ASTERIX Tokens

As well as the kernel tokens, there are an additional set of tokens which will be translated if the `-a` option is specified with the GENERIC command (see below).

These tokens are available to allow GENERIC to process existing ASTERIX software (SUN/98) which uses them. They are *not* recommended for use by new software. A list of the ASTERIX tokens is given in Appendix A.

4.3 The Local CHARACTER Size

The `<LTYPE>` token is used for declaring a variable which is local to the subroutine in which it is used, whereas `<TYPE>` is used to declare a variable which is passed to the routine as an argument. In practice, the translation for both of these tokens is identical for all types except CHARACTER.

In the CHARACTER type, a variable passed as an argument is declared CHARACTER*(*), but a local variable is declared CHARACTER*`<CHASIZ>`. `<CHASIZ>` is replaced by the "local character size." If not told otherwise, GENERIC will replace `<CHASIZ>` by 200, making local character variables CHARACTER*200.

It is possible to override this default character size if it is unsuitable. You can define the environment4 variable GENERIC_CHASIZ or you can use the `-c` flag on the **generic** command.

4.4 The Fortran Maximum Record Length

The GENERIC utility assumes it is dealing with Fortran 77 source code which cannot go beyond column 72. If the substitution of a token results in a line becoming longer than 72 characters, it will be broken automatically at column 72 and continued on the next line, using a ":" continuation character in column 6 of the next line.

It is recommended that source lines in generic routines be kept well below 72 characters in length to prevent this from happening.

NOTE: *Comment lines (* i.e. *those beginning 'C', 'c', 'D', 'd' or '*') which are longer than 72 characters are unaltered by* GENERIC*. So are lines of code with an in-line comment (* i.e. *those containing a "!" character).*

This algorithm is not perfect and it is possible to fool it with end of line comments on long lines or exclamation marks in character strings. However, this seems not to have proven a problem in practice.

5 Using the GENERIC Utility

The GENERIC utility is executed by typing the following command:

```
% generic [-t <types>] [-a] [-c <chasiz>] [-s] [-sc <char>] [-u] [-x]
          file1 [file2...]}
```

Arguments in square brackets are optional and items in angle brackets are descriptive terms for the actual string that would be given. The simplest form of the **generic** command is:

```
% generic prog.gen
```

which will produce a file called `prog.f` containing routines for all of the supported data types. Likewise for C

```
% generic prog.cgen
```

will create a file `prog.c`.

A softlink called **fgeneric** is also provided to avoid a clash with the IRAF generic preprocessor.

All generic input Fortran files must have a suffix of `.gen`, or `.g<types>` where `<types>` is a list of data types (see option `-t` below for a list). The latter was introduced to simplify building using GNU tools, and is the recommended naming convention. For C files the file extension should be `.cgen`.

You can specify multiple input files explicitly or with wild cards. All output Fortran files have a suffix of `.f`, and output C files have suffix `.c`.

The **generic** command accepts the following command-line options.

-t This must be followed by a list of the data types to be processed. The list should be separated from the `-t` by a space, and should consist of one or more of the letters `a, b, B, c, d, i, l, n, r, w` or `W`. The letters have the following meanings:

`a`	all data types
`b`	BYTE
`B`	unsigned BYTE
`c`	CHARACTER
`d`	DOUBLE PRECISION
`i`	INTEGER
`l`	LOGICAL
`n`	all numeric types (*i.e.* everything except CHARACTER and LOGICAL)
`r`	REAL
`w`	WORD (*i.e.* INTEGER*2)
`W`	unsigned WORD

For example, to generate routines for all of the standard Fortran numeric data types, type the command:

```
% generic -t dir prog.gen
```

(Using `n` instead of `dir` would generate byte and word versions as well.)

-c This should be followed by a space and then an integer giving the value of the local character size required. The default value is 200. This value can be specified by the environment variable `GENERIC_CHASIZ`. If `GENERIC_CHASIZ` is defined and the `-c` argument is present, the `-c` argument takes precedence.

-s Generate the output in separate files for each data type with the single character type specifier appended to the base name of the file. The default is to generate one output file for each input file. For example, if the input file is `sub1.gen` and the `-t` flag is set to `dir`, then specifying the `-s` flag will cause output files called `sub1d.f`, `sub1i.f` and `sub1r.f` to be generated. This option is provided as it is usual on Unix systems to have all Fortran routines in separate source files.

-sc This flag allows the user to specify one or more characters that will be used to separate the base file name from the appended character when generating separate output files for each data type. The `-sc` should be followed by a space and then the character(s) to be used as the separator. For example, the command:

```
% generic -t dir -sc _ func2.gen
```

will generate files called `func2_d.f`, `func2_i.f` and `func2_r.f`. If the `-sc` flag is given, then the `-s` flag can be omitted.

-a This will cause the ASTERIX tokens to be interpreted as well as the standard ones. Its use is deprecated.

-u The B and W types substitute `ub` and `uw` tokens respectively in the code and in generated filenames. This is for compability with the original VMS version when most generic routines were developed. However, note that the `-t` values of BW should still be used to obtain unsigned byte and unsigned-word instantiations.

-x Removes the last `x` from the file name before generating the names of the output files. This is provided for compatability with early generic files in the VMS era files that had a final `x` in the name to indicate where the token would be replaced in the instantiated files. In modern usage the trailing `x` is absent, and this option is not required.

It is possible to specify the list of data types to be processed and the local character size by setting environment variables `GENERIC_TYPES` and `GENERIC_CHASIZ`, *e.g.*

```
% setenv GENERIC_TYPES cdilr
% setenv GENERIC_CHASIZ 300
```

6 Data Type Conversions—the DCV Routines

6.1 Introduction

Occasionally, it may be necessary to convert data passed into a generic routine in a generic variable into another data type. A general set of conversion functions was originally provided within GENERIC to make this possible. However, *new code requiring such conversions should invoke the PRIMDAT library of functions*. PRIMDAT also provides many other instrinsic functions that are amenable to inclusion in generic code. This section remains only to interpret legacy code that still calls DCV routines.

A naming convention is used so that the function DCV_`<T1>TO<T2>` converts data type `<T1>` into `<T2>`.

The functions which can be are defined using Fortran statement functions in a file which may be included in a program by

```
INCLUDE 'DCV_FUN'
```

which should come after any data type definitions, but before any DATA initialisation or executable code. In the standard prologue of SGP/16, it should go in the “Internal References” section.

Conversions which cannot be defined as statement functions are provided as external functions in an object library (`/star/lib/dcv.a` on a Starlink system). Programs requiring these functions should be compiled

```
% f77 prog.f -L/star/lib `dcv_link`
```

on Unix.

The types of these functions can be declared in a program by including

```
      INCLUDE 'DCV_EXT'
```

(in the "External References" section of the standard prologue).

NOTE: *Statement functions are considerably faster than external functions, so programs which use the common conversions defined in 'DCV_FUN' will execute a lot faster.*

Only numeric conversions are supported at present, and there are no DCV_ functions for conversions involving CHARACTER and LOGICAL data types. A full list of the functions, the conversion performed, and their location and restrictions, is given in Appendix B.

6.2 An Example of Data Type Conversion

As an example, a generic subroutine for finding the mean of the data contained in an array, where the array is of varying type but the mean is always REAL, might look like the following routine.

```
      SUBROUTINE MEAN<T>( N, ARRAY, MEAN )
*+
*  Name:
*     MEAN<T>
*  Purpose:
*     Calculate the mean value contained in a <COMM> array.
*  Invocation :
*     CALL MEAN<T>( N, ARRAY, MEAN )
*  Description :
*     This calculates the mean value of then data contained in the given
*     <COMM> array, and returns the value in the REAL variable MEAN.
*  Arguments:
*     N = INTEGER( Given )
*        The dimension of the array.
*     ARRAY( N ) = <TYPE>( Given )
*        The array containing the data to be averaged.
*     MEAN = REAL( Returned )
*        The mean of the values contained in the array.
*                  "
*     (the rest of the standard prologue)
*                  "
*  Type Definitions:
      IMPLICIT NONE
*  Arguments Given:
      INTEGER N
      <TYPE> ARRAY( N )
*  Arguments Returned:
      REAL MEAN
*  External References:
      INCLUDE 'DCV_EXT'          ! Data conversion external functions
*  Local Variables:
      INTEGER I                  ! Array index
      <LTYPE> SUM                ! Sum of the data in the array
*  Internal References:
      INCLUDE 'DCV_FUN'          ! Data conversion statement functions
*-

*  Initialise the sum to zero.
      SUM = 0<CONST>

*  Accumulate the sum of the contents of the array.
      DO I = 1, N
        SUM = SUM + ARRAY( N )
      END DO

*  Divide by the dimension of the array to obtain the mean, using
*  DCV_<T>TOR to convert from <COMM> into REAL.
      MEAN = DCV_<T>TOR( SUM ) / REAL( N )

      END
```

In the above example, for the INTEGER version of the routine for instance, the `DCV_ITOR( SUM )` statement will be converted to `REAL( SUM )` by the statement function in `DCV_FUN`.

6.3 Range Checking

The DCV_<T1>TO<T2> functions will not check the range of the values given to them, and overflow errors are possible from some conversions (*e.g.* INTEGER to BYTE). It is recommended that an application check the range of any value given to a DCV function, to ensure it is within the allowed range. Global parameters of the form DCV__<T>MIN and DCV__<T>MAX are available for this purpose, which give the allowed range for type <T> (*e.g.* -128 to $+127$ for a BYTE). These by may included in a program using the following line.

```
INCLUDE 'DCV_PAR'
```

(in the "Global Parameters" section of the standard prologue).

Here is an example of the use of these parameters (for instance converting an INTEGER to another type).

```
*             "
*     (standard prologue)
*             "
*  Global Parameters:
      INCLUDE 'DCV_PAR'          ! DCV global parameters giving allowed
                                 ! ranges
*             "
*  External References:
      INCLUDE 'DCV_EXT'          ! DCV external function definitions
*             "
*  Local Variables:
      INTEGER IVAL               ! Integer value to be converted
      <TYPE> VAL                 ! Destination variable of type <COMM>
```

```
               "
*  Internal References:
      INCLUDE 'DCV_FUN'          ! DCV in-line statement functions
*-
              "
              "
      IF ( ( IVAL .GE. DCV__<T>MIN ) .AND.
     :     ( IVAL .LE. DCV__<T>MAX ) ) THEN

*  Value within range. The conversion is allowed.
         VAL = DCV_ITO<T>( IVAL )

      ELSE

*  Error. Value out of range ...
         <take appropriate action>
      END IF
              "
              "
```

Note that ranges are only provided for numeric data types.

An alternative (and easier) way of preventing numerical overflow from causing problems is to use the PRIMDAT routines instead of the DCV ones. The PRIMDAT routines will not crash your program if an overflow occurs. See SUN/39 for details.

A A Full Table of F77 Token Translations

This appendix gives the translations of all the tokens made by GENERIC for each of the data types. If a translation is illegal, the result will be "?" (and GENERIC will give a warning message).

Additional substitutions are made if the input file is a C source file, described in Appendix A.

To produce the BYTE version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"B"`	*Subroutine name extension*
<TYPE>	`"BYTE"`	*Argument type definition*
<LTYPE>	`"BYTE"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_BYTE"`	*HDS type*
<COMM>	`"BYTE"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified):

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`" "`	*BYTE to REAL conversion*
<EXT>	`" "`	*BYTE to REAL conversion extension*
<BRAK>	`" "`	*Brackets for BYTE to REAL conversion*
<ICONV>	`" "`	*BYTE to INTEGER conversion*
<IEXT>	`" "`	*BYTE to INTEGER conversion extension*
<IBRAK>	`" "`	*Brackets for BYTE to INTEGER conversion*
<REV>	`" "`	*REAL to BYTE conversion*
<RVBRAK>	`" "`	*Brackets for REAL to BYTE conversion*
<IREV>	`" "`	*INTEGER to BYTE conversion*
<IRVBRAK>	`" "`	*Brackets for INTEGER to BYTE conversion*

To produce the UNSIGNED BYTE version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"UB"`	*Subroutine name extension*
<TYPE>	`"BYTE"`	*Argument type definition*
<LTYPE>	`"BYTE"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_UBYTE"`	*HDS type*
<COMM>	`"UNSIGNED BYTE"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified):

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"REAL("`	*UNSIGNED BYTE to REAL conversion*
<EXT>	`"ZEXT("`	*UNSIGNED BYTE to REAL conversion extension*
<BRAK>	`"))"`	*Brackets for UNSIGNED BYTE to REAL conversion*
<ICONV>	`"INT("`	*UNSIGNED BYTE to INTEGER conversion*
<IEXT>	`"ZEXT("`	*UNSIGNED BYTE to INTEGER conversion extension*
<IBRAK>	`"))"`	*Brackets for UNSIGNED BYTE to INTEGER conversion*
<REV>	`"NINT("`	*REAL to UNSIGNED BYTE conversion*
<RVBRAK>	`")"`	*Brackets for REAL to UNSIGNED BYTE conversion*
<IREV>	`" "`	*INTEGER to UNSIGNED BYTE conversion*
<IRVBRAK>	`" "`	*Brackets for INTEGER to UNSIGNED BYTE conversion*

To produce the CHARACTER version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"C"`	*Subroutine name extension*
<TYPE>	`"CHARACTER*(*)"`	*Argument type definition*
<LTYPE>	`"CHARACTER*<CHASIZ>"`	*Local type definition (CHASIZ = 200 by default)*
<CONST>	`"?"`	*String to be appended to constant*
<HTYPE>	`"_CHAR"`	*HDS type*
<COMM>	`"CHARACTER"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified):

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"?"`	*CHARACTER to REAL conversion*
<EXT>	`"?"`	*CHARACTER to REAL conversion extension*
<BRAK>	`"?"`	*Brackets for CHARACTER to REAL conversion*
<ICONV>	`"?"`	*CHARACTER to INTEGER conversion*
<IEXT>	`"?"`	*CHARACTER to INTEGER conversion extension*
<IBRAK>	`"?"`	*Brackets for CHARACTER to INTEGER conversion*
<REV>	`"?"`	*REAL to CHARACTER conversion*
<RVBRAK>	`"?"`	*Brackets for REAL to CHARACTER conversion*
<IREV>	`"?"`	*INTEGER to CHARACTER conversion*
<IRVBRAK>	`"?"`	*Brackets for INTEGER to CHARACTER conversion*

To produce the DOUBLE PRECISION version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"D"`	*Subroutine name extension*
<TYPE>	`"DOUBLE PRECISION"`	*Argument type definition*
<LTYPE>	`"DOUBLE PRECISION"`	*Local type definition*
<CONST>	`".0D0"`	*String to be appended to constant*
<HTYPE>	`"_DOUBLE"`	*HDS type*
<COMM>	`"DOUBLE PRECISION"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified):

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"SNGL("`	*DOUBLE PRECISION to REAL conversion*
<EXT>	`" "`	*DOUBLE PRECISION to REAL conversion extension*
<BRAK>	`")"`	*Brackets for DOUBLE PRECISION to REAL conversion*
<ICONV>	`"NINT("`	*DOUBLE PRECISION to INTEGER conversion*
<IEXT>	`" "`	*DOUBLE PRECISION to INTEGER conversion extension*
<IBRAK>	`")"`	*Brackets for DOUBLE PRECISION to INTEGER conversion*
<REV>	`"DBLE("`	*REAL to DOUBLE PRECISION conversion*
<RVBRAK>	`")"`	*Brackets for REAL to DOUBLE PRECISION conversion*
<IREV>	`"DBLE("`	*INTEGER to DOUBLE PRECISION conversion*
<IRVBRAK>	`")"`	*Brackets for INTEGER to DOUBLE PRECISION conversion*

To produce the INTEGER version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"I"`	*Subroutine name extension*
<TYPE>	`"INTEGER"`	*Argument type definition*
<LTYPE>	`"INTEGER"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_INTEGER"`	*HDS type*
<COMM>	`"INTEGER"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified).

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"REAL("`	*INTEGER to REAL conversion*
<EXT>	`" "`	*INTEGER to REAL conversion extension*
<BRAK>	`")"`	*Brackets for INTEGER to REAL conversion*
<ICONV>	`" "`	*INTEGER to INTEGER conversion*
<IEXT>	`" "`	*INTEGER to INTEGER conversion extension*
<IBRAK>	`" "`	*Brackets for INTEGER to INTEGER conversion*
<REV>	`"NINT("`	*REAL to INTEGER conversion*
<RVBRAK>	`")"`	*Brackets for REAL to INTEGER conversion*
<IREV>	`" "`	*INTEGER to INTEGER conversion*
<IRVBRAK>	`" "`	*Brackets for INTEGER to INTEGER conversion*

To produce the LOGICAL version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"L"`	*Subroutine name extension*
<TYPE>	`"LOGICAL"`	*Argument type definition*
<LTYPE>	`"LOGICAL"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_LOGICAL"`	*HDS type*
<COMM>	`"LOGICAL"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified).

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"?"`	*LOGICAL to REAL conversion*
<EXT>	`"?"`	*LOGICAL to REAL conversion extension*
<BRAK>	`"?"`	*Brackets for LOGICAL to REAL conversion*
<ICONV>	`"?"`	*LOGICAL to INTEGER conversion*
<IEXT>	`"?"`	*LOGICAL to INTEGER conversion extension*
<IBRAK>	`"?"`	*Brackets for LOGICAL to INTEGER conversion*
<REV>	`"?"`	*REAL to LOGICAL conversion*
<RVBRAK>	`"?"`	*Brackets for REAL to LOGICAL conversion*
<IREV>	`"?"`	*INTEGER to LOGICAL conversion*
<IRVBRAK>	`"?"`	*Brackets for INTEGER to LOGICAL conversion*

To produce the REAL version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"R"`	*Subroutine name extension*
<TYPE>	`"REAL"`	*Argument type definition*
<LTYPE>	`"REAL"`	*Local type definition*
<CONST>	`".0E0"`	*String to be appended to constant*
<HTYPE>	`"_REAL"`	*HDS type*
<COMM>	`"REAL"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified).

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`" "`	*REAL to REAL conversion*
<EXT>	`" "`	*REAL to REAL conversion extension*
<BRAK>	`" "`	*Brackets for REAL to REAL conversion*
<ICONV>	`"NINT("`	*REAL to INTEGER conversion*
<IEXT>	`" "`	*REAL to INTEGER conversion extension*
<IBRAK>	`")"`	*Brackets for REAL to INTEGER conversion*
<REV>	`" "`	*REAL to REAL conversion*
<RVBRAK>	`" "`	*Brackets for REAL to REAL conversion*
<IREV>	`"REAL("`	*INTEGER to REAL conversion*
<IRVBRAK>	`")"`	*Brackets for INTEGER to REAL conversion*

To produce the WORD version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"W"`	*Subroutine name extension*
<TYPE>	`"INTEGER*2"`	*Argument type definition*
<LTYPE>	`"INTEGER*2"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_WORD"`	*HDS type*
<COMM>	`"WORD"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified).

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"REAL("`	*WORD to REAL conversion*
<EXT>	`" "`	*WORD to REAL conversion extension*
<BRAK>	`")"`	*Brackets for WORD to REAL conversion*
<ICONV>	`" "`	*WORD to INTEGER conversion*
<IEXT>	`" "`	*WORD to INTEGER conversion extension*
<IBRAK>	`" "`	*Brackets for WORD to INTEGER conversion*
<REV>	`"NINT("`	*REAL to WORD conversion*
<RVBRAK>	`")"`	*Brackets for REAL to WORD conversion*
<IREV>	`" "`	*INTEGER to WORD conversion*
<IRVBRAK>	`" "`	*Brackets for INTEGER to WORD conversion*

To produce the UNSIGNED WORD version of a program, the following substitutions are made.

Kernel tokens (translated by default):

TOKEN	SUBSTITUTE	COMMENT
<T>	`"UW"`	*Subroutine name extension*
<TYPE>	`"INTEGER*2"`	*Argument type definition*
<LTYPE>	`"INTEGER*2"`	*Local type definition*
<CONST>	`" "`	*String to be appended to constant*
<HTYPE>	`"_UWORD"`	*HDS type*
<COMM>	`"UNSIGNED WORD"`	*Comment describing type*

ASTERIX tokens (translated when `-a` is specified).

TOKEN	SUBSTITUTE	COMMENT
<CONV>	`"REAL("`	*UNSIGNED WORD to REAL conversion*
<EXT>	`"ZEXT("`	*UNSIGNED WORD to REAL conversion extension*
<BRAK>	`"))"`	*Brackets for UNSIGNED WORD to REAL conversion*
<ICONV>	`" "`	*UNSIGNED WORD to INTEGER conversion*
<IEXT>	`"ZEXT("`	*UNSIGNED WORD to INTEGER conversion extension*
<IBRAK>	`")"`	*Brackets for UNSIGNED WORD to INTEGER conversion*
<REV>	`"NINT("`	*REAL to UNSIGNED WORD conversion*
<RVBRAK>	`")"`	*Brackets for REAL to UNSIGNED WORD conversion*
<IREV>	`" "`	*INTEGER to UNSIGNED WORD conversion*
<IRVBRAK>	`" "`	*Brackets for INTEGER to UNSIGNED WORD conversion*

B A Full List of Conversion Functions

This appendix gives the conversion functions which are available using the DCV utility, and describes which ones are statement functions and which ones are external.

The following conversions are defined as in-line statement functions in 'DCV_FUN'. (They are either trivial, or are mentioned in the Fortran manual as legal conversions).

Function name	Conversion	
DCV_BTOB	BYTE to BYTE	
DCV_UBTOUB	UNSIGNED BYTE to UNSIGNED BYTE	
DCV_UBTOD	UNSIGNED BYTE to DOUBLE PRECISION	
DCV_DTOD	DOUBLE PRECISION to DOUBLE PRECISION	
DCV_ITOD	INTEGER to DOUBLE PRECISION	
DCV_RTOD	REAL to DOUBLE PRECISION	
DCV_WTOD	WORD to DOUBLE PRECISION	
DCV_UWTOD	UNSIGNED WORD to DOUBLE PRECISION	
DCV_UBTOI	UNSIGNED BYTE to INTEGER	
DCV_DTOI	DOUBLE PRECISION to INTEGER	
DCV_ITOI	INTEGER to INTEGER	
DCV_RTOI	REAL to INTEGER	
DCV_UWTOI	UNSIGNED WORD to INTEGER	
DCV_UBTOR	UNSIGNED BYTE to REAL	
DCV_DTOR	DOUBLE PRECISION to REAL	
DCV_ITOR	INTEGER to REAL	
DCV_RTOR	REAL to REAL	
DCV_WTOR	WORD to REAL	
DCV_UWTOR	UNSIGNED WORD to REAL	
DCV_UBTOW	UNSIGNED BYTE to WORD	
DCV_DTOW	DOUBLE PRECISION to WORD	***
DCV_RTOW	REAL to WORD	***
DCV_WTOW	WORD to WORD	
DCV_UBTOUW	UNSIGNED BYTE to UNSIGNED WORD	
DCV_UWTOUW	UNSIGNED WORD to UNSIGNED WORD	

Overflow errors are possible from the conversions marked “***”

The following conversions are also defined as in-line statement functions in 'DCV_FUN'. However, they are not mentioned in the Fortran manual and should be used with caution. Trial and error has shown that the Fortran compliers on all Starlink supported systems will make the conversions successfully :-

Function name	**Conversion**	
DCV_UBTOB	UNSIGNED BYTE to BYTE	***
DCV_DTOB	DOUBLE PRECISION to BYTE	***
DCV_ITOB	INTEGER to BYTE	***
DCV_RTOB	REAL to BYTE	***
DCV_WTOB	WORD to BYTE	***
DCV_UWTOB	UNSIGNED WORD to BYTE	***
DCV_BTOD	BYTE to DOUBLE PRECISION	
DCV_BTOI	BYTE to INTEGER	
DCV_WTOI	WORD to INTEGER	
DCV_BTOR	BYTE to REAL	
DCV_BTOW	BYTE to WORD	
DCV_ITOW	INTEGER to WORD	***

Overflow errors are possible from the conversions marked "***"

The following conversions are defined as external functions in the DCV object library :-

Function name	**Conversion**
DCV_BTOUB	BYTE to UNSIGNED BYTE
DCV_DTOUB	DOUBLE PRECISION to UNSIGNED BYTE
DCV_ITOUB	INTEGER to UNSIGNED BYTE
DCV_RTOUB	REAL to UNSIGNED BYTE
DCV_WTOUB	WORD to UNSIGNED BYTE
DCV_UWTOUB	UNSIGNED WORD to UNSIGNED BYTE
DCV_BTOUW	BYTE to UNSIGNED WORD
DCV_UBTOUW	UNSIGNED BYTE to UNSIGNED WORD
DCV_DTOUW	DOUBLE PRECISION to UNSIGNED WORD
DCV_ITOUW	INTEGER to UNSIGNED WORD
DCV_RTOUW	REAL to UNSIGNED WORD
DCV_WTOUW	WORD to UNSIGNED WORD

Overflow errors are possible in ALL these conversions, so range checking is essential.

A C Token Translations

This appendix gives the extra translations for use within C sources files that define functions intended to be called from Fortran:

Kernel tokens (translated by default):

TOKEN	COMMENT
<TC>	*Upper case C function name extension - D, I, F etc*
<TLC>	*Lower case subroutine name extension - d, i, c etc*
<CNFTYPE>	*CNF data type - DOUBLE, INTEGER, CHARACTER etc*
<CTYPE>	*C data type - double, int, char $*$ etc*