

### 7.2.6 Interfacing with Assembler Code

Assembler language routines which conform to either the Waterloo C default register convention or an OS linkage convention can be called directly from a Waterloo C function, possibly with a `#pragma linkage` directive. Alternatively, specialized interface functions can be written to handle a variety of linkage conventions.

The following routine can be called from Waterloo C with up to ten parameters. The first parameter is the address of an assembler language routine which follows the OS 370 calling conventions. The next 9 parameters are stored in memory and a pointer to them is passed in R1 to the assembler routine. The assembler routine may return a value in R0 which will be passed back as the value of the C function call.

```
@INTER    CSECT
           ENTRY INTER
INTER     STM    12,1,4*10(12)    SAVE REGISTERS
           LA     13,4*10+4*6(,12) GET FREE AREA
           L      15,0(,12)       GET ROUTINE
           LA     1,4(,12)        POINT AT PARMS
           BALR   14,15           CALL
           LR     11,0            SET RETURN VALUE
           LM     12,1,4*10(12)   RESTORE REGISTERS
           BR     13             RETURN TO CALLER
           END
```

The following is an example of an OS assembler routine which could be called using the above interface routine. This assembler routine expects R1 to point to three integer values. The sum of these values is returned in R0.

```
@ADDUP    CSECT
           ENTRY ADDUP
ADDUP     STM    15,12,12(13)    SAVE REGISTERS
           LR     12,15          GET ADDRESSABILITY
           USING  ADDUP,12       . . .
           L      2,0(,1)        GET FIRST PARM
           A      2,4(,1)        ADD 2ND PARM
           A      2,8(,1)        ADD 3RD PARM
           ST     2,16(,13)      STORE RETURN VALUE
           LM     15,12,12(13)   RESTORE REGISTERS
           BR     14             RETURN TO CALLER
           END
```

## Reference

The following C program uses the above routine by calling the `Inter` routine. The output of this program is the sum of 1, 2 and 3.

```
#include <stdio.h>

extern void ADDUP();
extern int inter( void *rtn, ... );

int main()
{
    printf( "%d\n", inter( ADDUP, 1, 2, 3 ) );
    return( 0 );
}
```

The library functions `acall` and `oscall` provide similar assembler interfaces (see the function descriptions for more details).

### 7.2.7 Waterloo C and VS FORTRAN

Although C is applicable to a wide variety of programming problems, it may be useful to be able to call existing subroutines in VS FORTRAN. For example, engineering and scientific utility packages, vector processor libraries and complex number function libraries are some of the most commonly used FORTRAN applications. Waterloo C provides facilities for C functions to call VS FORTRAN routines, as well as for VS FORTRAN programs to call C functions.

#### Calling VS FORTRAN Routines from C

Four functions are provided to allow Waterloo C programs to call VS FORTRAN routines: `fvshint`, `fvscalli`, `fvscallr` and ~~`fvscall`~~.

If the main part of the program is written in C, the `fvshint` function must be called when VS FORTRAN input or output operations are to be used.

`fvscalli` and `fvscallr` are used to call FORTRAN routines that return `int` or `double` values respectively. These routines have similar formats and perform almost identically. The first parameter is the address of the FORTRAN routine. The second parameter is the number of arguments to be passed. The remaining parameters are the addresses of the values to be passed.

For example, the following C program calls the FORTRAN function `DGAMMA`, which returns a `double` value:

```
extern double fvscallr();
extern /*FORTRAN*/ double DGAMMA();

int main()
{
    double retval;
    double parm1;

    parm1 = 5.0;
    retval = fvscallr( DGAMMA, 1, &parm1 );
    printf( "GAMMA %f = %f", parm1, retval );
    return( 0 );
}
```

## Reference

DGAMMA should have the form:

```
FUNCTION DGAMMA( PARM1 )  
  REAL*8 DGAMMA  
  REAL*8 PARM1  
  * * *
```

Each of the interface functions is described in more detail in the *Waterloo C Run-time Library Reference*.

## Calling C Functions from VS FORTRAN

To call a C function from a FORTRAN program, the C run-time environment must first be initialized by calling the function `CINIT`. This function initializes the C run-time stack, memory manager, and I/O system. A non-zero value is returned if an error condition occurs; otherwise, zero is returned. Signal handling is not initialized so that any run-time errors will use the VS FORTRAN error handling.

The routine `CEXIT` should be called after the last use of C facilities to close any opened files and to release any memory that was allocated by the C functions.

To call C functions, the interface routines `CCALL`, `ICFUNC` and `DCFUNC` are provided. `CCALL` can be used to call a C function that does not return a value. `ICFUNC` can be used to call a C function that returns an integer value and `DCFUNC` can be used to call a C function that returns a `REAL*8` value. To allow maximum flexibility, all of these routines pass the address of the VS FORTRAN argument list to the C function.

The VS FORTRAN argument list contains the addresses of variables, arrays, character strings, and subprogram arguments. The last argument address in the argument list has the high order bit set to one.

The full word preceding the argument list contains the displacement of the length list from the start of the argument list. The length list is a list of addresses of full word lengths for character string arguments. A length address entry is reserved for every argument, however it is valid only if the argument is a character string. The displacement from the beginning of this list for a length address is the same as the displacement for the corresponding argument address from the start of the argument list.

The FORTRAN argument list has the following form:

parameter list ->

offset of length address table
pointer to address of routine
address of parameter 1 address of parameter 2 address of parameter 3 ...
address of length parm 1 address of length parm 2 address of length parm 3 ...

For example, the C function `dfromi`, that returns a double value with the same value as the first parameter (of type `int`), can be called as follows:

```

EXTERNAL  CINIT, CEXIT
EXTERNAL  DCFUNC, DFROMI
REAL*8    DCFUNC
INTEGER*4 CINIT

C
  IF(CINIT().EQ.0) GOTO 20
    WRITE(6,10)
10    FORMAT(' CINIT FAILED')
    STOP
20  CONTINUE
C
  I = 10
  WRITE(6,30) ( DCFUNC( DFROMI, I ) )
30  FORMAT( ' DOUBLE VALUE = ', F20.4 )
C
  CALL CEXIT
C
  STOP
  END

```

## Reference

The C function would have the form:

```
double dfromi( int **arglist )
{
    double retval;
    int    parml;

    parml = *arglist[ 1 ];
    retval = parml;
    return( retval );
}
```

The *VS FORTRAN Version 2 Programming Guide* (SC26-4222) contains more information about VS FORTRAN linkage conventions.

## 7.2.8 Separation of Global Data – The SPLIT Option

The SPLIT compiler option causes global read/write data (modifiable variables defined outside a C function) to be placed in a separate object (and assembler) file from the read-only code and data. The option simplifies the adaptation of programs for use in run-time environments where separate read-only and read-write object code is required.

When the SPLIT option is specified, R1 (or R8, if an OS-style convention is used) is set in each function prologue to point to the start of the block of global data for the source file, if it exists. A global variable is then accessed by retrieving the difference between the address of the variable and the address of the data block, as generated by the compiler, and adding it to R1 (or R8), yielding the address of the variable.

For example, consider the following program, VARS:

```
#include <stdio.h>

/* VARS -- A Program With Read/Write Data. */

int var1;
int var2;

int main( int  argc,
          char **argv )
{
    var2 = 0;
    return( var2 );
}
```

When it is compiled with the SPLIT and ASMSRC options, code similar to the following is written to the assembler file (DD ASM1):

```
@VARS      CSECT
            EXTRN VARS@
*          1: #include <stdio.h>
*          2:
*          3: /* VARS -- A Program With Read/Write Data. */
*          4:
*          5: int var1;
*          6: int var2;
*          7:
*          8: int main( int argc,
*          9:          char **argv )
            ENTRY MAIN
```

## Reference

```
MAIN      DS      0X
          USING MAIN,11
          STM      14,2,8(12)
          LR       14,11
          L        1,#L2
          DROP     11
          USING MAIN,14
* 10:      {
* 11:      var2 = 0;
          SR       2,2
          ST        2,4(0,1)
* 12:      return( var2 );
          L        11,4(0,1)
* 13:      }
          LM       14,2,8(12)
          BR       13
#L2       DS      0H
          DC       AL4(VARS@)
          DROP     14
          EXTRN    $CSTART
          ENTRY    $REF$CST
$REF$CST DS      0X
          DC       AL4($CSTART)
          END
```

Code similar to the following is written to the secondary assembler file (DD ASM2):

```
VARS@     CSECT
          ENTRY    VAR1
VAR1      DS      0X
          DC       4X'00'
          ENTRY    VAR2
VAR2      DS      0X
          DC       4X'00'
          END
```

By specifying the `const` attribute, read-only (initialized) data can be kept in the code file.



### 7.2.9 Register-Based Global Data – The RENT Option

The RENT compiler option provides a capability to define an arbitrarily long, register-based, read/write data area. As for the SPLIT option, global data is placed in a separate object and assembler file, however variables are accessed using offsets from the read/write data area base register, R10. Separate data files are only generated when necessary.

Because of the different variable access techniques, object files that are compiled using this option should not be linked with files that are not compiled with this option.

When the RENT option is specified, a function that accesses global data sets R1 to the difference between the start of the data area for the program and the start of the data area for the file in which the function is contained. (R8 is used instead of R1 if an OS-style linkage convention was specified.) Therefore, a particular global variable is referenced by adding the address contained in R10 to the displacement contained in R1 (or R8), and then adding the difference between the relative address of the variable and the relative address of the start of the data area for the file (a constant generated by the compiler).

For example, consider the following program, VARS:

```
#include <stdio.h>

/* VARS -- A Program With Read/Write Data. */

int var1;
int var2;

int main( int  argc,
          char **argv )
{
    var2 = 0;
    return( var2 );
}
```

When it is compiled with the RENT and ASMSRC options, code similar to the following is written to the assembler file (DD ASM1):

```
@VARS    CSECT
         EXTRN $SCRWDAT
         EXTRN VARS@
```

## Reference

```
*      1: #include <stdio.h>
*      2:
*      3: /* VARS -- A Program With Read/Write Data. */
*      4:
*      5: int var1;
*      6: int var2;
*      7:
*      8: int main( int  argc,
*      9:             char **argv )
          ENTRY MAIN
MAIN      DS      0X
          USING MAIN,11
          STM     14,2,8(12)
          LR      14,11
          L       1,#L2
          DROP    11
          USING MAIN,14
*      10: {
*      11:     var2 = 0;
          SR      2,2
          ST      2,4(10,1)
*      12:     return( var2 );
          L       11,4(10,1)
*      13: }
          LM      14,2,8(12)
          BR      13
#L2      DS      0H
          DC      AL4(VARS@- $CRWDAT)
          DROP    14
          EXTRN   $CSTART
          ENTRY   $REF$CST
$REF$CST DS      0X
          DC      AL4($CSTART)
          END
```

Code similar to the following is written to the secondary assembler file (DD ASM2):

```
VARS@    CSECT
          ENTRY  VAR1
VAR1      DS      0X
          DC      4X'00'
          ENTRY  VAR2
VAR2      DS      0X
          DC      4X'00'
          END
```

By specifying the `const` attribute, read-only (initialized) data can be kept in the code file. The section *Preparing a Program for the LPA Using the RENT Option* describes a procedure that can be used to link a program that was compiled with this option.

### 7.2.10 Register-Based Global Data Using AUX Files

A method of accessing uninitialized, modifiable (read/write global) variables using a base register involves the use of an "auxiliary storage allocation information" (AUX) file. Each file in the program must be compiled with the AUX compiler option, as described in the *Compiler Options* section, which specifies a file that describes the layout of the global data area.

Note that all files in a program that reference global variables must be compiled with this option, if it is used.

Each line in an AUX file is either an offset or an origin directive, or an include directive that includes another AUX file. A line of the form

```
origin <where>
```

sets the origin (that is, where the next variable begins) to <where> bytes beyond the start of the global data area. A line of the form

```
offset <variable> <size>
```

reserves <size> bytes at the current origin for <variable>.

Variables that are to be located in this area must be declared as `extern` in any C file where they are referenced. The compiler reads the AUX file declarations to resolve any references to them. Thus, such variables should not be defined in any file.

For example, consider the following segment of C source code:

```
struct info
{
    char  name[ 20 ];
    short age;
    int   salary;
};

struct llist
{
    struct llist *back;
    struct llist *forward;
    struct info  data_area;
};

extern struct llist FirstGuy;
extern struct info  Me, MyBoss;

extern int Profits;
```

An AUX file to describe these variables would be:

```
origin 0

offset Me          26    % location 0
offset MyBoss      26    % 26
offset FirstGuy    34    % 52
offset Profits     4     % 86

% Now, origin is 90
```

The origin of each element can be calculated as the origin of the previous entry, plus the length of the previous entry. It is necessary to calculate the size of each item in bytes. For example, struct info has a size of 20+2+4, or 26 bytes; struct llist has a size of 4+4+26 or 34 bytes (each pointer takes 4 bytes and struct info has size 26 bytes).

The library initialization routine \$STARTUP sets R10 to the start of a 4K data area for global variables. Thus, in the example above, the variable Me is at the address contained in R10 and FirstGuy is at the location obtained by adding 52 to the contents of R10.

### 7.2.11 Macro Expansion Of Function Calls

Calls to the C functions `strlen`, `strcpy`, `memcmp`, `memcpy` and `memset` can optionally be processed as macro expansions. When a call to one of these functions is encountered, code to perform the corresponding operation is generated directly, using one of the System/370 MVCL, CLCL and TRT instructions. The library function is not referenced.

The `#pragma inline` preprocessor directive is used to indicate that a function is to be generated inline and the `#pragma callable` directive is used to indicate that a function call is to be generated. For example, the following sequence indicates that code for the `strlen` function should be generated inline, but code for `strcpy` is not.

```
#pragma inline strlen
#pragma callable strcpy
```

While the inline code is larger than a call to the library function would normally be, it is usually more efficient because no parameter passing or function linkage is required. This feature is most useful when one of the above functions is called many times within a frequently repeated code section.

### 7.2.12 Waterloo C Library Read/Write Data

All of the global read/write data used by the library is found in two data files.

'WCOS.CLIB.C(\$IOBRWD)' contains the data for C library FILE structures, pointers to which are returned by the `fopen` function.

'WCOS.CLIB.C(\$CLIBRWD)' contains a pointer to the library read/write data area that is used by the C functions that are called from a FORTRAN program.

### 7.2.13 Stack Overflow Checking – The STACKCHK Option

During execution, memory for auto variables, parameters and register save areas for a C function are allocated using a stack protocol from a memory pool that is defined at program startup time. The STACKCHK compiler option allows a programmer to request that functions within a source file should have stack overflow checking code generated. Stack overflow checking is accomplished by generating a call to the library routine \$STAKCHK at the start of the routine to be generated. \$STAKCHK verifies that there is sufficient space remaining on the run-time stack to allow the function to execute. If insufficient space remains, a stack overflow exception is generated.

For example, consider the HELLO program:

```
#include <stdio.h>

/* HELLO C -- A First Program. */

int main()
{
    printf( "hello, world\n" );
    return( 0 );
}
```

If this file is compiled with the STACKCHK and ASMSRC options, code similar to the following is written to the assembler file (DD ASM1):

```
@HELLO    CSECT
          EXTRN $STAKCHK
*         1: #include <stdio.h>
*         2:
*         3: /* HELLO C -- A First Program. */
*         4:
*         5: int main()
          ENTRY MAIN
MAIN      DS      0X
          USING  MAIN,11
          A      12,18(,11)
          STM    13,14,0(12)
          L      14,22(,11)
          BALR   13,14
          DC     X'00000007C00000000'
          DC     AL4($STAKCHK)
```



```

        L      13,0(,12)
        STM    12,2,4(12)
        LR     15,12
        LA     12,32(,12)
        LR     14,11
        DROP   11
        USING  MAIN,14
*      6:      {
*      7:      printf( "hello, world\n" );
        LA     2,#G146
        ST     2,0(,12)
        L      11,#P89
        BALR   13,11
*      8:      return( 0 );
        SR     11,11
*      9:      }
        LM     12,2,4(15)
        BR     13
        DROP   14
        EXTRN  $CSTART
        ENTRY  $REF$CST
$REF$CST DS    0X
        DC     AL4($CSTART)
#G146    DS    0X
        DC     X'88859393966B40A6969993841500'
        EXTRN  PRINTF
#P89     DS    0F
        DC     AL4(PRINTF)
        END

```

Note that space for parameters for functions that are called by the current function is taken into account when the check is performed.