Appendix H. Using Multibanking for Large Programs

H.1. Large Programs

The Series 1100 hardware architecture has a default 65,535 decimal word address range for the instructions of a collected program, including all user subprograms and all referenced run-time library routines. If the size of the collected program is larger than this 65K-word range, the collector produces truncation errors because it is trying to place an address which is greater than 65K into a 16-bit instruction u-field. Index registers can hold an 18-bit address. Therefore, ASCII FORTRAN generates code using index registers to hold addresses if the O option (the over-65K-address option) is used on the ASCII FORTRAN processor calls when the programs are compiled. This raises the boundary to a 262K-word address range for your collected program before truncation problems again appear.

If use of the O option results in no truncation errors during collection, your problem is solved. (Having the statement COMPILER(PROGRAM=BIG) in source programs is equivalent to using the O option when compiling them.)

However, if a program still gets truncation errors during collection, you must construct a multibanked program or place some large data items in virtual space. When the size problems are due to large user data objects, use either banked space or virtual space to solve the problem. Using virtual space is preferred since it is easier to set up and use (Appendix M). When the size problem is due to a large amount of executable code rather than large data items, you must construct a multibanked program to solve the problem. When the problem stems from both the size of the code and the size of the data objects, you must construct a multibanked program to solve the code size problem, and use either banked or virtual space to solve the data size problem. (Multibanked programs can also use virtual space.)

H.2. Banking

Banking is a Series 1100 mechanism for sharing the address space between different pieces of code or data. For example, you can use the collector IBANK directive to direct the collector to construct a bank (an I-bank) holding subprogram X, and to construct another I-bank to hold subprogram Y. These two I-banks can be created with overlapping addresses. The same thing can be done with data. You can use the collector DBANK directive to place common block CB1 into one bank (a D-bank) and common block CB2 into another parallel D-bank using the same address space. These are called paged data banks. In this general manner, you create a banked program which needs less address space than the unbanked program. You can define almost any number of I-banks and D-banks, though the collector documentation should be referenced for the exact limit (it is about 250). You must construct a collector symbolic (sometimes called

a MAP symbolic) containing a sequence of collector directives that define the banking structure of your program.

For this mechanism to work, generated code must be able to move an address window from bank to bank. A processor state register (PSR) is a hardware register that defines two address windows for an executing program, an I-bank window and a D-bank window. An LIJ (load I-bank base and jump) instruction moves the I-bank window, and an LDJ (load D-bank base and jump) instruction moves the D-bank window from one bank to another.

Series 1100 hardware has two basic types:

1. older single-PSR systems

2. newer dual-PSR systems

3. Single-PSR systems include the 1106, 1108, 1100/10, and 1100/20; dual-PSR systems include the 1110, 1100/40, 1100/60, and 1100/80.

On dual-PSR systems the two PSRs are referred to as the main PSR (PSRM) and the utility PSR (PSRU). The 1100/60 and 1100/80 have the concept of four basing registers called bank descriptor registers (BDRs). There is a one-for-one association between these four BDRs and the four windows as defined by the main and utility PSRs:

BDR	PSR Window
BDR0	I-bank PSRM
BDR1	I-bank PSRU
BDR2	D-bank PSRM
BDR3	D-bank PSRU

This appendix refers to only PSRs since there is a one-for-one equivalence to BDRs.

Besides the LIJ and LDJ instructions to switch banks, the 1100/80 and 1100/60 have a generalized LBJ instruction that you can use to switch both I-banks and D-banks.

When constructing a collector symbolic to create a multibanking structure for ASCII FORTRAN programs, you must follow certain conventions: (1) no address overlap should ever occur between I-banks and D-banks since results are unpredictable; (2) when multiple D-banks containing paged data are defined, they must start at the same address (FORTRAN's mechanism to switch D-bank basing depends upon this); (3) in any multibanked collection, one bank is defined as the control bank. The control bank is assumed to be always available and based since it contains any unbanked programs and data and also the unbanked portions of the run-time library routines.

H.2.1. General Banking Example (Dual-PSR System)

The following example consists of three separately compiled elements. MAIN1 is the main program and SUB1 and SUB2 are subroutines. The first statement in each sample routine is a directive to the compiler indicating that the final collected program is banked, and appropriate linkages (e.g. LIJ, LDJ, LBJ instructions) must be used to ensure that the correct banks are visible when necessary. The sizes of the code and data in the examples don't warrant the use of banking since these are simple examples for instruction only.

Example of a Main Program (MAIN1):

```
      COMPILER (BANKED=ALL)
      COMMON /cb1/ a,x /cb2/ b,y
      CHARACTER*1 a,b
      DATA a/'a'/, b/'b'/
      WRITE(6,100) 'reference to:', a
      WRITE(6,100) 'reference to:', b
      a = 'c'
      b = 'd'
      WRITE(6,100) 'reference to:', a
      WRITE(6,100) 'reference to:', b
100   FORMAT (1X, A13, 1X, A1)
      x = 2.
      y = sqrt(x)
      z = x + y
      WRITE(6,200) x, y
200   FORMAT (1X, 'sqrt of', F10.5, ' is', F10.5)
      CALL sub1 (a, b, x, y, z)
      END
```

Example 1 of a Subprogram (SUB1):

```
      COMPILER(BANKED=ALL)
      SUBROUTINE sub1 (a, b, x, y, z)
      CHARACTER*1 a, b
      WRITE(6,100) 'in subroutine sub1'
100   FORMAT(1X, A18)
      WRITE(6,200) a, b, x, y, z
200   FORMAT(1X, 'a=', A1, ' b = ', A1, ' x = ', F10.5, ' y = ',
     1F10.5, ' z = ', F10.5)
      CALL sub2(b, 2.0)
      END
```

Example 2 of a Subprogram (SUB2):

```
      COMPILER (BANKED=ALL)
      SUBROUTINE sub2(a, x)
      CHARACTER*(*) a
      CHARACTER*19 b3cell /'common block cb3'/
      COMMON /cb3/ b3cell
      CHARACTER*19 b4cell /'common block cb4'/
      COMMON /cb4/ b4cell
      PRINT *, 'a, len(a), x:', a, len(a), x
      PRINT *, 'common block cb3:', b3cell
      PRINT *, 'common block cb4:', b4cell
      END
```

H.2.1.1. Collection of the General Banking Example

The following collector symbolic can be used to collect the three sample program elements into a banked program. The LIB directive can be dropped if the ASCII FORTRAN library is in the system relocatable library, SYS$*RLIB$. This is the general form that you should follow for multibanking of ASCII FORTRAN programs on dual-PSR Series 1100 systems.

```
LIB FTN*RLIB.
IBANK,MRD    IBANKM     . INITIALLY BASED, MAIN PSR
    IN MAIN1
IBANK,RD     IBANK1,IBANKM
    IN SUB1
IBANK,RD     IBANK2,IBANKM
    IN SUB2
DBANK,UD     DBANK1,(040000,IBANKM,IBANK1,IBANK2)
    . INITIALLY BASED, UTILITY PSR
    IN F2ACTIV$($1)
    IN CB1
DBANK,D      DBANK2,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB2
DBANK,D      DBANK3,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB3
DBANK,CM     MAIND,(DBANK1,DBANK2,DBANK3)
    . MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
    IN MAIN1
    IN SUB1
    IN SUB2
END
```

The collection of the sample program using this collector symbolic would result in the banking structure shown in Figure H-1. The lines under the bank names are similar to the lines in a collector S-option listing in that they indicate length.

If the three FORTRAN relocatables are copied to file TPF$, and if the above collector symbolic is in element TPF$.BMAP, you can collect this program in a banked absolute and execute it with the following control images. (This assumes that there are no other relocatables in file TPF$.)

```
@MAP     BMAP,BABS
@XQT     BABS
```

The following control images do a default nonbanked collection of the program and give the same results when executed.

```
@MAP,I MAP,ABS
IN MAIN1
LIB FTN*RLIB.
@XQT     ABS
```

I-Banks Based on Main I-Bank PSR (PSRM) (start at 01000)	D-Banks Based on Utility D-Bank PSR (PSRU) (start at 040000 or over)	Control D-Bank Based on Main D-Bank PSR (PSRM) (starts after largest of D-banks based on utility D-bank PSR and may reach 262K limit)
IBANKM MAIN1	DBANK1 CB1	
IBANK1 SUB1	DBANK2 CB2	Local data, library—MCORE$ area
IBANK2 SUB2	DBANK3 CB3	
C2F$ I/O common bank		

NOTES:

1. *Program code goes into I-banks.*
2. *Named common blocks go into D-banks (paged data banks).*
3. *Data local to subprograms, blank common, any programs or named common not placed in other banks and the run-time library routines, all go into the control bank.*
4. *The C2F$ I/O common bank is not mentioned in the collection, though it is referenced at run time for all I/O activities.*
5. *The paged data banks must start at or after address 040000 to avoid address overlap with the hidden C2F$ I/O common bank.*
6. *The area after the control bank is open for the I/O complex to acquire buffer space. Executive Requests (ERs) to MCORE$ are made at run time to expand this area.*
7. *If any collected addresses go over 65K, use the O option or the COMPILER (PROGRAM=BIG) statement with all of the ASCII FORTRAN compilations.*

Figure H-1. Dual-PSR Banking Structure

The execution of the banked or nonbanked absolute results in the following output:

```
reference to: a
reference to: b
reference to: c
reference to: d
sqrt of   2.00000 is   1.41421
in subroutine sub1
a= c b = d x =  2.00000 y =  1.41421 z =  3.41421
a, len(a), x:d          1  2.0000000
common block cb3:common block cb3
common block cb4:common block cb4
```

H.2.1.2. Analysis of the Collector Symbolic

This subsection describes the collector symbolic listed in H.2.1.

The main program MAIN1 is placed in an I-bank with the M option which makes it initially based on the main PSR. (The main program must be in an initially based bank.)

The other two routines, SUB1 and SUB2, are placed into two other I-banks, each starting at the same address as the I-bank containing MAIN1. (They can also be put into the same I-bank as MAIN1 since they are so small.)

All I-banks have the R option on the IBANK directive to indicate they are read-only I-banks. As a read-only bank, there is less Executive swap file activity.

All D-banks except the control D-bank have the D option on the DBANK directive to indicate that they are dynamic banks. This means that they can be swapped out by the Executive if they are not currently based, saving on main storage usage (though possibly causing more Executive swap file activity).

The bank names given on the IBANK and DBANK directives (for example, IBANKM) are called bank descriptor indexes, or BDIs. The collector gives them integer values that are used by the LIJ and LDJ bank-switching instructions.

The paged data banks contain named common blocks and must be based on the utility D-bank PSR. The paged data bank DBANK1 is chosen to be the one initially based. The U option on the DBANK directive for DBANK1 indicates it is initially based on the utility PSR. The other paged data banks holding named common are put at the same address as DBANK1.

The location counter one code ($1 code) of the run-time activate element F2ACTIV$ is put at the beginning of initially based paged data bank DBANK1. The same is done with $3 code of F2ACTIV$ for each of the other paged data banks. The F2ACTIV$ location counter ($1) contains information to make the bank in which it resides self-identifying. This location counter is often referred to as the bank's ID area. The code under location counter ($3) in F2ACTIV$ is an exact copy of location counter one. To work properly, the location counters one and three must be collected at the same address in the banks. The IN directive of F2ACTIV$ ($3) has MAIND in parenthesis. This is called local element inclusion and bypasses possible LOCAL-GLOBAL CONFLICT messages from the collector.

The control bank MAIND is the D-bank named in the DBANK directive with the C option, and the M option on it means it is also initially based on the main D-bank PSR. Only the main program MAIN1 is included through use of an IN statement in this bank since the collector puts anything not specifically included in another bank in the control bank. The control bank MAIND is placed after the largest of the three paged data banks so that no address overlap occurs.

The area after the end of the control bank is used by the storage management complex to obtain buffer space for the ASCII FORTRAN run-time system. Executive Requests (ERs) to MCORE$ are made to acquire this space.

The three paged data banks must not be defined at less than an 040000 (octal) address and the paged data banks must start at an address higher than the highest I-bank address. This is because their address space would then overlap the C2F$ I/O common bank, and unpredictable results would occur.

The named common block CB4 is not given a home in any paged data bank. If the main program and any subprograms are explicitly included in an I-bank and a D-bank by an IN statement, CB4 falls in the control bank and, since the control bank is always based, it is not dynamically banked. (See the section on element placement in the Collector Reference, UP-8721 [applicable version].) This does not result in any problems; in fact, any subprograms not specifically included in a bank by an IN statement fall harmlessly in the control bank. As long as the control bank does not get so large as to cause collector truncation errors again, this is harmless. Any number of subprograms can be included in an I-bank, and any number of named common blocks can be included in a paged data bank. (Blank common and data local to subprograms must be in the control bank.) The criteria for the contents of a bank should be a function of the final collected size of the bank, and also a function of the locality of reference to the bank to try to minimize thrashing between banks. (Any problem caused by excessive Executive swap file activity can be minimized by carefully making selected banks static by not putting the D option on their I-bank or D-bank statements.) A reasonable size for an I-bank or paged data bank is approximately 16,000 decimal words. This means that the control bank can be up to about 32,000 words in size before truncation problems occur again.

H.2.1.3. Large Banks

If reasonable I-bank and paged data bank sizes still result in truncation errors at collection time, or, if large paged data banks (greater than approximately 30,000 words) are to be defined, the O option is needed on the ASCII FORTRAN processor calls to allow a 262K address range. In addition, the ordering of the paged data banks and the control bank must be inverted to prevent collector truncation errors on the run-time library routines in the control bank. This means that the control bank must be placed after the I-banks in the address space, and before the paged data banks in the address space. Since the ASCII FORTRAN run-time system makes the control bank larger via ER MCORE$ to obtain buffer space, you must leave enough room between the control bank and the paged data banks for I/O main storage requirements. (Appendix G contains formulas for estimating I/O main storage requirements for a program.) Allowing 10,000 decimal words is usually sufficient. However, if an ER MCORE$ results in an address overlap of the control bank and paged data banks, error termination or a hang is ensured. The F2FCA library element can also be reassembled with a sufficiently large local area in it (see G.7).

To collect your program with large D-banks, the collector symbolic (from H.2.1.1) must be changed as follows. The DBANK directive for D-bank MAIND and the IN directive on MAIN1 are moved back to just before the DBANK1 definition. Then these statements are changed as follows:

```
.
.
.
DBANK,CM   MAIND, (040000, IBANKM, IBANK1, IBANK2)
   IN MAIN1
DBANK,UD   DBANK1,(MAIND+10000)
.
.
.
```

The rest of the collector symbolic remains unchanged.

The resulting collector symbolic is:

```
LIB FTN*RLIB
IBANK,MR   IBANKM       INITIALLY BASED, MAIN PSR
    IN MAIN1
IBANK,R    IBANK1,IBANKM
    IN SUB1
IBANK,R    IBANK2,IBANKM
    IN SUB2
DBANK,CM   MAIND,(040000,IBANKM,IBANK1,IBANK2)
     .MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
    IN MAIN1
    IN SUB1
    IN SUB2
DBANK,UD   DBANK1,(MAIND+10000)     . INITIALLY BASED, UTILITY PSR
    IN F2ACTIV$($1)
    IN CB1
DBANK,D    DBANK2,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB2
DBANK,D    DBANK3,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB3
END
```

The collection then results in the banking structure of Figure H-2. The lines under the bank names are similar to the lines in a collector S-option listing in that they indicate length.

When the common blocks don't fit in the paged D-banks (truncation errors appear), put them in virtual space (see Appendix M).

H.2.1.4. Variations on the Dual-PSR Structure

The generalized example shows multiple I-banks and multiple D-banks being used at the same time. If your program is large only in the amount of I-bank code, you can simply omit the definition of the paged data banks and only define I-banks and the control bank. If the program has large common blocks causing the size problem, then the collector symbolic can be cut back to defining only one I-bank, the control bank, and multiple paged data banks.

The LIB directive tells the collector where to obtain the ASCII FORTRAN run-time library. The simple form of the LIB directive causes all run-time library routines, both code and data, to fall into the control bank MAIND since they are not explicitly included in any bank. The following form of the LIB directive directs the collector to put anything taken from FTN*RLIB into I-bank IBANKM, and D-bank MAIND, in a normal $ODD/$EVEN I-bank/D-bank split:

```
LIB FTN*RLIB.(IBANKM/$ODD,MAIND/$EVEN)
```

This can minimize the size of the MAIND control bank.

I-Banks Based on Main I-Bank PSR (PSRM) (starts at 01000)	Control Bank Based on Main D-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on Utility D-Bank PSR (PSRU) (starts after the control bank and goes up to a 262K limit)
IBANKM MAIN1	MAIND Local data, library-MCORE$ area	DBANK1 CB1
IBANK1 SUB1		DBANK2 CB2
IBANK2 SUB2		DBANK3 CB3
C2F$ I/O common bank		

NOTES:

1. *The paged data banks can extend out to the 262K address limit.*
2. *The 10K area between MAIND and the paged data banks is used by the I/O complex for buffers. If it is not sufficient, the separation must be increased or the library element F2FCA reassembled with a nonzero reserve.*
3. *I/O acquires storage in increments of eight storage blocks (4096 decimal words).*

Figure H-2. Dual-PSR Banking Structure, Over 65K

H.2.2. Banking for Single-PSR 1100 Systems

The collector symbolics described in H.2.1 through H.2.1.4 use the utility PSR of dual-PSR systems to hold an address window for paged data. This utility PSR does not exist on single-PSR systems such as the 1106, 1108, 1100/10, and 1100/20. If your program needs only multiple I-banks and not multiple D-banks, the previously described collector symbolics can be used with the definitions of the paged data banks removed. If your program needs both multiple I-banks and multiple D-banks, it simply cannot be done on single-PSR hardware. However, you can define a banking structure for multiple D-banks for single-PSR systems. A single I-bank is defined, and it is also made the control bank to hold all unbanked code and data. The paged data banks are defined to come after the control bank, but enough room must be left between them for I/O buffers (which are dynamically acquired by ER MCORE$ at runtime).

However, this type of collection has a problem resulting from the I-bank holding all unpaged data. Because of this, no common banks can be referred to at runtime to do I/O, calls to the common mathematical library (CML), etc. The run-time library used must be a very special one, having all run-time routines in relocatable form. (The ASCII FORTRAN library must be built as a type1 library, with the relocatable form of the PCIOS common I/O modules, and also the relocatable form of the CML modules.)

Another problem results from ASCII FORTRAN putting a SETMIN on all relocatables it generates, which ensures that code to refer to array elements is correct. The collector emits a warning on each FORTRAN element. For example:

```
MAIN1 MINIMUM ADDRESS IGNORED-LC0 NOT IN DBANK
```

These collector warnings can be ignored, but the I-bank must be started at address 040000 or after to ensure that the FORTRAN-generated code works correctly.

You must also edit and reassemble the ASCII FORTRAN library element F2BDREQU$ when performing a multibanked collection for single-PSR systems. Adjust the values of three EQUs as follows:

TAG	Dual-PSR (default)	Single-PSR
CBDR$	2	0
VPDR$	1	1
BKBDR$	3	2

The example in H.2.1 using MAIN1, SUB1, and SUB2 can be collected using multiple D-banks for single-PSR systems with the following collector symbolic:

```
LIB FTN*RLIBX.        . VERY SPECIAL LIBRARY !
IBANK,MC      IBANKM,040000      . CONTROL BANK NOW
    IN MAIN1,SUB1,SUB2
DBANK,MD      DBANK1,(IBANKM+10000)
    IN F2ACTIV$($1)
    IN CB1
DBANK,D       DBANK2,DBANK1
    IN(IBANKM) F2ACTIV$($3)
    IN CB2
DBANK,D       DBANK3,DBANK1
    IN(IBANKM) F2ACTIV$($3)
    IN CB3
END
```

Collection results in the following collector diagnostics:

```
MAIN1   MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB1    MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB2    MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
CB4     MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
```

The collection results in the banking structure given in Figure H-3 for the single-PSR multiple D-bank problem. The lines under the bank names in Figure H-3 are similar to the lines in a collector S-option listing in that they indicate length.

One I-Bank (the Control Bank) Based on the Main I-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on the Main D-Bank PSR (PSRM) (starts after the control bank and may go up to 262K)
IBANKM Code, library, unbanked data-MCORE$	DBANK1 CB1
	DBANK2 CB2
	DBANK3 CB3

NOTES:

1. *There are unavoidable collector diagnostics.*
2. *A totally local library must be used; no run-time common banks can be referenced.*
3. *The 10K separation between IBANKM and the D-banks must be able to satisfy all buffer requests from the ASCII FORTRAN run-time system, or it must be increased, or the library element F2FCA must be reassembled with a nonzero reserve large enough to satisfy the buffer requests.*
4. *The I-bank must start at or after address 040000.*
5. *If the paged data banks extend beyond 65K, use the O option on all of the ASCII FORTRAN compilations.*
6. *IBANKM cannot extend beyond 65K in addressing.*

Figure H-3. Single-PSR Banking Structure

H.2.3. Banking, Efficiency, and Source Program Directives

The examples in H.2.1 through H.2.2 use a simple generalized directive to the ASCII FORTRAN compiler to indicate that banking is used in the final absolute program. In fact, this generalized statement, COMPILER(BANKED=ALL), means:

- Each subprogram referenced can be in a different I-bank, in the same I-bank, or the control bank.
- Input arguments can be in paged data banks or in the control bank.
- Named common blocks can be in paged data banks or in the control bank.

Therefore, the actual banking structure is virtually unknown to the compiler, and yet it must create linkages to ensure that items are visible or based when they are referenced.

H.2.3.1. I-Bank Linkages

The ASCII FORTRAN compiler uses the LIJ instruction to link between I-banks. This linkage is fairly efficient since the bank switch is done and then the called subprogram is entered, usually for some period of time. The compiler generates a pseudo-linkage called the IBJ$ linkage. The collector replaces this linkage with an LMJ instruction if the destination is in the control bank, or the same I-bank, and with an LIJ instruction if the destination is in a different I-bank.

H.2.3.2. D-Bank Linkages

Each time the compiler generates code to reference data in a (possibly) paged data bank (which may be currently based), it must also generate an activate sequence. This activate code has several variations, but a typical sequence is two to four instructions long.

Example:

Variables A, B, and C are in named common blocks CB1, CB2, and CB3.

The FORTRAN statement A = B+C is to be compiled.

If you haven't given any directives to the ASCII FORTRAN compiler indicating that multiple D-banks are being used, three machine instructions are generated for this statement: a LOAD, an ADD, and a STORE. If you indicate to the ASCII FORTRAN compiler (through BANK statements) that multiple D-banks are being used, there are also three activate sequences generated. This results in nine instructions instead of three for the unbanked program. In addition, each activate sequence contains LBJ instructions. The LBJ instruction is simulated in the Executive for older single-PSR systems and may cause a presence-bit interrupt if the bank is swapped out on any Series 1100 system. Program efficiency is extremely dependent upon the contents of the paged data banks and in the organization of the ASCII FORTRAN code.

Because performance can be so dramatically affected, you can supply several directives, including the BANK statement (see 6.6) and several COMPILER statement options (see 8.5), to the ASCII FORTRAN compiler to help program efficiency.

H.2.3.3. Multiple I-Banks Only

If your collected program is constructed using multiple I-banks for code and does not define multiple paged data banks, use the LINK = IBJ$ option of the COMPILER statement rather than the more general BANKED=ALL option. The compiler then generates the efficient IBJ$ linkage for subprogram references and generates code assuming that data is not banked. The resulting program should be as efficient as an unbanked program.

H.2.3.4. Multiple Paged Data Banks

Once your multiple D-bank program is debugged and running, you might notice many activate code sequences in the generated code that are not necessary, since a given paged program bank may contain several named common blocks. Also, if you are actually hopping between paged data banks a lot in the generated code, you may wish for a much faster activate sequence.

H.2.3.4.1. The BANK Statement

The BANK statement associates a paged data bank BDI name with one or more named common blocks. Therefore, you can tell the compiler that common blocks CB1, CB2, and CB3 are actually in the same paged D-bank and then the compiler does not generate activate sequences when the program references them. Also, since the BANK statement tells the compiler that these items are definitely banked and the BDI is supplied, a more efficient bank switch can be done. The compiler generates a direct LBJ instruction to change which D-bank is currently based rather than calling an F2ACTIV$ run-time routine.

Using a BANK statement to associate a BDI with a common block results in more efficient code to reference that block. However, if the COMPILER (BANKED=ALL) statement is left in the program, the compiler still generates inefficient code sequences. These code sequences reference those named common blocks not specified in BANK statements. When you use BANK statements to associate BDIs with common block names for all common blocks residing in paged D-banks, you can replace the BANKED = ALL option of the COMPILER statement with the following three options:

(BANKED=ACTARG), (BANKED=DUMARG), (LINK=IBJ$)

In addition to more efficient code to reference nonbanked common blocks, the options ensure that:

- banked arguments are handled properly
- linkages to subprograms are correct

Since BANK statements associate specific BDI names with common blocks, if you change the banking structure, you must also change all the programs and recompile them.

The BANK statement and various COMPILER statement directives are described in 6.6 and 8.5.

H.2.3.4.2. Optimization and Program Organization

The ASCII FORTRAN compiler is only effective at remembering which bank is currently based and does not generate unnecessary activate sequences if global optimization is used during program compilation. (Global optimization is called with the Z option on the ASCII FORTRAN processor call.) Also, you should attempt to organize your code so that references to a given D-bank are grouped in areas of code. This is especially true for the inner loop of DO-loops. Try to have any inner loops refer to items in one paged data bank. (Unbanked data items can be referred to in any manner.)

Example:

```
      SUBROUTINE SUBX (A,IA)
      COMPILER(BANKED=ALL)
      DIMENSION  A(IA)
      COMMON/C1/A1(1000),B1(1000)
      COMMON/C2/A2(1000),B2(1000)
      BANK/BNK1/C1,C2
      COMMON WORK(1000)  @BLANK COMMON
      IL = IA-1
      DO  10  I=1,IL
10        WORK(I)=A(I)/A(I+1)+.03
      DO  20  I=1,IL
          A1(I)=WORK(I)*B2(I)/B1(I)-A2(I)
          IF(A1(I).NE.0.0) A2(I)=1/A1(I)
20        CONTINUE
      END
```

You have (possibly) banked arguments A and IA and two named common blocks that are known to be in D-bank BNK1.

Blank common can never be banked, so the program does some initial processing on the input array A and moves it to WORK in blank common. (A local array can also be used.) Since only one bank is referenced inside the first loop, the activate code sequence is moved out of the loop (if global optimization is used). The same thing is true for the second loop, and no activate sequences are done inside the loop. A single reference to an external routine inside either loop causes at least one set of activate code to be generated inside the loop since the external routine can possibly change which paged data bank is currently based.

If you had not supplied the BANK statement, the generated code would be loaded with activate sequences since the compiler must assume the worst case.

H.2.4. Banking Summary

- Programs constructed using multiple I-banks and no multiple D-banks should use the COMPILER(LINK=IBJ$) statement to indicate banking to the ASCII FORTRAN compiler.
- The COMPILER(BANKED=ALL) statement stresses ease of use for multiple D-bank programs. However, CPU efficiency suffers when compared to the use of the BANK statement for common block names.
- Programs with multiple D-banks can cut the number of activate code sequences generated, and can cause the direct generation of LBJ instructions by the selected use of BANK statements to associate named common blocks with specific paged data bank BDIs. Replace the COMPILER (BANKED=ALL) statement with COMPILER (BANKED=ACTARG), (BANKED=DUMARG), (LINK=IBJ$) to enhance efficiency.
- If BANK statements are used in a FORTRAN program to enhance efficiency, no error diagnostics occur if they are incorrect. Bad program results can occur.
- The use of global optimization cuts the number of generated activate sequences dramatically.
- Judicious organization of program logic and careful definition of the contents of paged data banks can have a very beneficial influence on performance.

- There must never be an address overlap between any I-bank and any D-bank, or between the control D-bank and any paged data banks. If the run-time library used is a normal common bank, the C2F$ I/O common bank can extend to address 037777 (octal). Therefore, no D-bank should start below address 040000.

- The control bank holds all of your unbanked data and routines, all run-time library D-bank, and any unbanked library routines. The control bank must be initially based and must never be unbased by an LIJ, LDJ, or LBJ instruction.

- If any addresses go beyond 65K in the collection, the O option (or the statement COMPILER (PROGRAM=BIG)) is needed on all ASCII FORTRAN compilations.

- Any element containing only block data subprograms must be included using an IN directive in the control bank in the collection, since the collector may otherwise ignore it. (This is true for nonbanked collections as well.)

- Control bank size can be minimized by supplying a LIB statement to the collector that causes the code or I-bank portions of the run-time library to go into one of your I-banks, for example, LIB FTN*RLIB.(IBANKM/$ODD,MAIND/$EVEN).

- Multiple D-bank operation depends upon copies of the activate code existing in each paged data bank at exactly the same relative address. The $1 and $3 F2ACTIV$ code segments are identical, and the only reason for the location counter split is to avoid local-global conflict messages during collection. The easiest way to ensure the same address for ACTIV$ code is to make each paged data bank start at the same address and to have the ACTIV$ code first in each D-bank.

- The local element inclusion of F2ACTIV$ code is also important. The $3 code (an exact copy of $1 code) has no tags, but refers to data in the control bank; therefore, it must be visible only to the control bank.

- Single-PSR 1100 systems can't have both multiple I-banks and multiple D-banks in the same collection.

- The TYPE BLOCKSIZE64 collector directive can save storage when the absolute element resulting from the collection has many banks.

- The single-PSR multiple D-bank setup needs a special library that contains a totally relocatable form of all run-time routines so that no common banks are referenced. The run-time element F2BDREQU$ must also be reassembled with some EQU values changed (see H.2.2).

Appendix I. Error Diagnostics in Checkout Mode

The diagnostics explained in Tables I-1 and I-2 are associated with the checkout mode of the FORTRAN (ASCII) compiler (see 10.6).

Table I-1. Messages Occurring During Program Load

Message	Explanation
**** CHECKOUT RELOCATION ERRORS ****	If any errors are encountered while loading your program, this message is issued and the error are messages are then printed.
WARNING: NAME IS UNDEFINED: *name*	You referenced a subprogram that is not defined in your compilation unit or the FTN library. The offending name is printed on the line following the message.
ERROR: USER PROGRAM TOO LARGE	An address generated while loading your program doesn't fit in an address field. Your program is too large. It may fit if the O option is used, or if the Z option is omitted on the processor call card.
NO MAIN ENTRY POINT, NO EXECUTION POSSIBLE	This message is produced if your program does not contain a main program. Instead the program contains only subroutines, functions, and BLOCK DATA subprograms. Interactive debug mode is entered.
BAD LINE NUMBER *n*	The line number, *n* (specified in the BREAK command), is either out of range for your program or is on a nonexecutable statement.
BLOCK DATA PROGRAM NOT FOUND	The BLOCK DATA program specified in the *p* field of the PROG command or the *p* subfield of a command doesn't exist in the FORTRAN symbolic element.

Table I-2. Messages Generated by Interactive Debugging

Message	Explanation
COMMAND NOT ALLOWED	The GO command (no fields) or the WALKBACK command can't be executed because normal execution of the FORTRAN program is not possible; that is, there is no main program in the element, or the CALL command has executed a subprogram and returned.
COMMAND NOT ALLOWED BECAUSE OF CONTINGENCY	The GO command (no fields) can't be executed because a contingency is captured by the compiler. For example, if the FORTRAN program encounters a guard mode (IGDM) contingency, then normal execution of the program can't resume. A RESTORE command can bring back an original version of the program.
CONSTANT MUST BE TYPE *data-type*	The constant in the third field of the SET command isn't the same data type as the variable in the first field. The SET command doesn't perform conversions between data types.
ELEMENT HAS NO MAIN PROGRAM	The main program is specified in the *p* field of the PROG command or in the *p* subfield of a command, but the FORTRAN symbolic element doesn't contain a main program.
ENTIRE ASSUMED-SIZE ARRAY CANNOT BE DUMPED	The range of an assumed-size array is unknown. Only individual elements of an assumed-size array can be dumped.
ENTRY POINT NOT FOUND	The entry point specified in the *s* field of the CALL command or in the parameter list of the CALL command doesn't exist in the FORTRAN source.
ERROR: NO USER PROGRAM FOUND	You are using a RESTORE command but haven't previously done a SAVE on the desired version.
FTEMP$ STORAGE DESTROYED	A subprogram's temporary storage area (for saving registers and the parameter list) is destroyed because of an error in your program. The specified variable can't be dumped.
FUNCTION HAS NOT BEEN CALLED	A reference is made to a variable that is a character function entry point, but the function has not yet been called during execution of the FORTRAN program.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
ILLEGAL COMMAND	An illegal debug command name is specified when input is solicited with ER ATREAD$, or the name specified in the *cmd* field of the HELP command isn't a debug command name.
ILLEGAL SYMBOLIC NAME	An illegal FORTRAN variable name is specified in the *v* subfield of a command, or an illegal subroutine or function name is specified in the *p* field of the PROG command or the *p* subfield of a command.
ILLEGAL SYNTAX	A general syntax error is found. This includes specifying a field for a command when none is required, or not specifying a field when one is required.
INCORRECT NUMBER OF SUBSCRIPTS FOR ARRAY *	The number of subscripts specified for the array in the *v* subfield of a command does not equal the number of dimensions declared for the array in the specified FORTRAN program unit.
IO ERROR ON LOADING USER PROGRAM, LOAD ABORTED	An I/O error occurs while accessing your program file during execution of the RESTORE command. The command is aborted. This may result in error termination also.
IO ERROR ON USER OUTPUT FILE, 'SAVE' COMMAND ABORTED	An I/O error occurs while accessing your program file during execution of the SAVE command. The command is aborted.
LABEL BREAK LIST IS FULL	An attempt is made to add an entry to the statement label break list with the command BREAK *n* L [/*p*], but the list already has eight entries.
LABEL UNDEFINED *	The statement label *n* in the BREAK *n* L [/*p*] command is not declared in the specified FORTRAN program unit.
LINE NUMBER BREAK LIST IS FULL	An attempt is made to add an entry to the line number break list with the command BREAK *n*, but the list already has eight entries.
NO BREAK SET FOR LABEL *	The statement label *n* (in the specified program unit) in the command CLEAR *n* L [/*p*] is not in the statement label break list.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
NO BREAK SET FOR LINE NUMBER	The line number *n* in the command CLEAR *n* isn't in the line number break list.
PARAMETER'S SUBPROGRAM HAS NOT BEEN CALLED	An attempt is made to reference a subroutine or function parameter, but the subprogram has not yet been called during execution of the FORTRAN program.
PROGRAM UNIT NOT FOUND	The symbolic name specified in the *p* field of the PROG command or the *p* subfield of a command doesn't exist in the FORTRAN symbolic element as a named program unit.
SETBP NOT ALLOWED ON COMPILER GENERATED FOR SINGLE-PSR MACHINE	The SETBP command can only be executed on an ASCII FORTRAN compiler generated for a dual-PSR machine. The nonreentrant ASCII FORTRAN absolute taken off the test file (file 2) of the ASCII FORTRAN release tape is a compiler generated for 1108 (single-PSR).
SUBSCRIPT OUT OF RANGE FOR ARRAY	The constant subscripts specified for the array in subfield *v* of a command are too big or too small.
****UNDEFINED SUBROUTINE ENTRY****	During execution, your program calls a function or subroutine that is undefined. The name of the subprogram was previously printed out with the checkout relocation errors.
USER FILE REJECTED, NOT FASTRAND FORMATTED	You are attempting a SAVE or RESTORE command, but the file is not a program file. Something has happened, making it unusable. The file affected is the relocatable output (RO) file specified on the @FTN processor call command.
USER INPUT FILE CANNOT BE ASSIGNED	Your program file can't be assigned to do the RESTORE command. Some other run must be using it.
USER OUTPUT FILE CANNOT BE ASSIGNED	Your program file can't be assigned to do the SAVE command. Some other run must be using it.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
VARIABLE IS AN ARRAY *	The variable in subfield *v* of a command has no subscripts, but the variable is declared as an array in the specified program unit. An array element is required.
VARIABLE IS NOT AN ARRAY *	The variable in subfield *v* of a command has subscripts, but the variable is not declared as an array in the specified program unit.
VARIABLE NOT DEFINED *	The variable in subfield *v* of a command is not declared in the specified FORTRAN program unit.
WARNING: CHARACTER CONSTANT TRUNCATED	The character constant in the third field of the SET command has too many characters to fit in the character variable. It is truncated to the declared length of the variable.

* *This error message is followed by a second printed line. This line specifies the program unit (in the FORTRAN element) from which the variable or statement label (in the command image) comes. One of the following formats:*

IN MAIN PROGRAM
IN MAIN PROG n

IN BLOCK DATA n
IN BLOCK DATA PROGRAM m

IN SUBROUTINE n [:e]

IN FUNCTION n [:e]

where:

n *is a program unit name.*

e *is an external program unit name.*

m *is an unnamed block data program sequence number.*

The specified program unit is taken from the p *subfield of the command, or from the PROG command default program unit, if* p *is not specified in the command.*

Appendix J. ASCII FORTRAN Level 8R1 and Higher Levels

J.1. General

ASCII FORTRAN levels 9R1 and higher contain all the features of the FORTRAN standard, X3.9-1978 (called FORTRAN 77). ASCII FORTRAN level 8R1 doesn't have all these features. This appendix compares ASCII FORTRAN level 9R1 and higher to level 8R1. ASCII FORTRAN level 8R1 is missing the following six statements: PROGRAM (see 7.9), INTRINSIC (see 7.2.4), SAVE (see 7.12), OPEN (see 5.10.1), CLOSE (see 5.10.2), and INQUIRE (see 5.10.3). It is incompatible with ASCII FORTRAN level 9R1 and higher in the storage allocation of character data, DO-loops, the typing of parameter constants and statement functions, and list-directed input/output. It doesn't contain the 13 new intrinsic functions: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, and LLT (see 7.3.1), or the new logical operators, .EQV. and .NEQV.

A new option, STD=66, is added to the COMPILER statement (see 8.5 and 8.5.6). This new option forces the ASCII FORTRAN compiler and library routines for level 9R1 and higher to execute as previous levels of ASCII FORTRAN do in the areas of storage of character data, DO-loops, typing of statement functions and parameter constants, and list-directed input and output.

This appendix is organized into subsections corresponding to the sections of the standard document, X3.9-1978. Each subsection of this appendix contains extensions in level 9R1 and higher over ASCII FORTRAN level 8R1 and conflicts between ASCII FORTRAN levels 9R1 and higher and level 8R1. ASCII FORTRAN extensions to level 8R1 in levels 9R1 and higher are features that have been implemented to make ASCII FORTRAN conform to the standard completely. Conflicts occur where the same construct can have different meanings in the two levels. Thus, conflicts imply that a change is made to a feature in ASCII FORTRAN level 8R1 to achieve compatibility with the standard.

J.2. FORTRAN Terms and Concepts

Extensions:

A main program can have a PROGRAM statement as its first statement. Level 8R1 has no PROGRAM statement (see 7.9).

Conflicts:

Character storage units for a datum are logically consecutive. Level 8R1 starts each character datum on a word boundary. The COMPILER statement provides an option, STD=66, to allow compatibility with previous levels on character data (see 8.5 and 8.5.6).

J.3. Characters, Lines, and Execution Sequence

Extensions:

PARAMETER statements can occur before and among IMPLICIT statements. Any specification statement that designates the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name; the PARAMETER statement must precede all other statements containing the symbolic name of constants that are defined in the PARAMETER statement. Level 8R1 does not allow typing of PARAMETER constants (see 6.3 and 6.8). The COMPILER statement provides an option, STD=66, to allow compatibility with previous untyped PARAMETER constants.

Conflicts:

None.

J.4. Data Types and Constants

Extensions:

A complex constant can be written as a pair of integer constants or real constants. Level 8R1 allows only real constants (see 2.2.1.3).

Conflicts:

None.

J.5. Arrays and Substrings

Extensions:

You can use array names in a SAVE statement. Level 8R1 does not have a SAVE statement (see 7.12).

Conflicts:

None.

J.6. Expressions

Extensions:

- Complex operands are allowed in relational expressions with the .EQ. and .NE. operators unless one operand is double-precision. The comparison of a double precision value and a complex value is not permitted. Level 8R1 does not allow complex operands in relational expressions (see 2.2.3.3.1).

- The logical operators .NEQV. and .EQV. with lowest precedence are allowed. These operators are not in level 8R1 (see 2.2.3.3.1).

Conflicts:

None.

J.7. Executable and Nonexecutable Statement Classification

Extensions:

None.

Conflicts:

None.

J.8. Specification Statements

Extensions:

- EQUIVALENCE statements can contain character substring names. Level 8R1 doesn't allow character substring names in EQUIVALENCE statement lists (see 6.4).

- Integer constant expressions are allowed for subscript and substring expressions in EQUIVALENCE statements. Level 8R1 does not allow substring expressions in EQUIVALENCE statements (see 6.4).

- In the COMMON statement, an optional comma is allowed before the slash that comes before the common block name, that is, [[,] / [*cb*] / *nlist*] . . . . No comma is allowed for level 8R1 (an error occurs) (see 6.5).

- A parameter constant can be typed in an IMPLICIT statement or in an explicit type statement (see 6.3 and 6.8). Level 8R1 does not allow typing of parameter constants. The COMPILER statement provides an option, STD=66, to allow compatibility between levels 9R1 and higher and lower levels of ASCII FORTRAN for typing of parameter constants (see 8.5 and 8.5.6).

- The name of a statement function can appear in an explicit type statement. The name of a statement function can be typed by an IMPLICIT statement. Level 8R1 does not type statement functions (see 6.3 and 7.4.1). The COMPILER statement provides the option, STD=66, to allow compatibility between level 9R1 and lower levels of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.6).

- The length in a CHARACTER type statement can be an asterisk or an integer constant expression in parentheses as well as just an unparenthesized constant (that is, (*) or (*exp*) or *const*). An entity in a CHARACTER statement must have an integer constant expression as a length specification unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name. These exceptions can have a length specification of asterisk. The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression. Neither an asterisk nor an expression is allowed in the length specification in level 8R1 (see 6.3.2).

- The length for a CHARACTER array element can occur before and after the element (that is, *a*(*d*)**length*). In level 8R1, the length can come before the subscript but an error is issued if it appears after the subscript (see 6.3.2).

- The comma is optional in the character type statement in the form:

  CHARACTER[**len* [,]] *nam* [,*nam*] . . .

  Level 8R1 issues an error message for the comma following *len* (see 6.3.2).

- In the IMPLICIT statement, the length for character entities can be an unsigned, nonzero integer constant, or an integer constant expression enclosed in parentheses that has a positive value. An unsigned, nonzero integer constant is allowed in level 8R1 but not an expression (see 6.3.1).

- A BLOCK DATA subprogram name can occur in an EXTERNAL statement. Level 8R1 does not allow the optional name for a BLOCK DATA subprogram (see 7.2.3 and 7.8.2).

- The INTRINSIC statement identifies a symbolic name as representing an intrinsic function. Level 8R1 does not have the INTRINSIC statement (see 7.2.4).

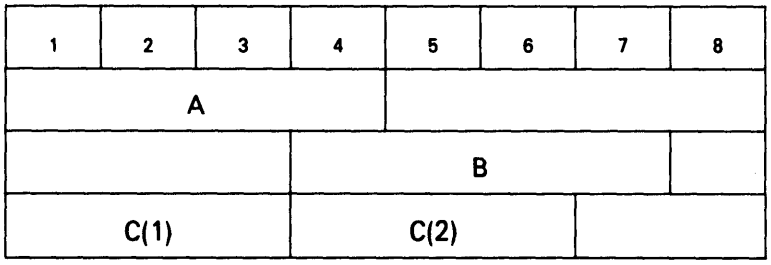
- The SAVE statement retains the definition status of an entity after execution of a RETURN or an END statement in a subprogram. Level 8R1 does not have a SAVE statement (see 7.12).

Conflicts:

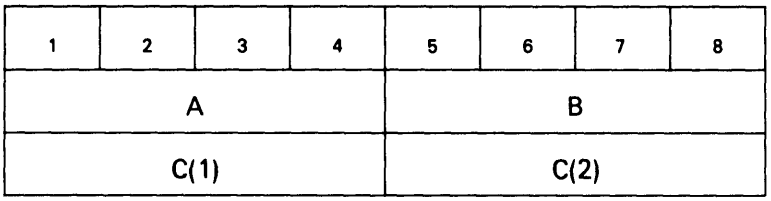
- Character equivalencing is based on character storage units in level 9R1 and higher and on words in level 8R1. This can give different results. This applies to both explicit EQUIVALENCE statements and to argument association and COMMON association. For example:

```
CHARACTER A*4,B*4,C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

Level 9R1:



Level 8R1:



The COMPILER statement provides an option, STD=66, to provide for compatibility with lower levels of ASCII FORTRAN (see 8.5 and 8.5.6).

- The PARAMETER statement gives a constant a symbolic name. If the type of the name is not default implied, the type must be specified by an explicit type statement or by an IMPLICIT statement prior to the appearance of the name in a PARAMETER statement. PARAMETER symbolic names have no type in level 8R1. Assignment of the value of the expression is done in level 9R1 and higher as in an assignment statement (that is, with type conversion, if necessary). The syntax of the PARAMETER statement is different in level 8R1 in that no parentheses can be used. Both forms of the PARAMETER statement syntax are allowed in level 9R1 and higher (see 6.8). The STD=66 option in the COMPILER statement provides for compatibility on previous levels of ASCII FORTRAN for typing of parameter constants (see 8.5 and 8.5.6).

J.9. DATA Statement

Extensions:

- The comma before the variable list is optional, that is, [[,]*nlist / clist /*] . . . . The comma is required in level 8R1. A warning is given if the comma is omitted (see 6.9.1).

- Substring names are allowed in the variable list. Level 8R1 doesn't allow substring names in a DATA statement variable list (see 6.9.1).

Conflicts:

A PARAMETER constant beginning with the letter O in the constant list of a DATA statement is interpreted by level 8R1 as an octal constant and by level 9R1 and higher as the PARAMETER constant (see 6.9.1).

For example:

```
PARAMETER (02=5.)
DATA A/02/
```

J.10. Assignment Statements

Extensions:

None.

Conflicts:

None.

J.11. Control Statements

Extensions:

The DO-variable and the DO-statement parameters can be real, double precision, or integer. Level 8R1 generates an error for noninteger DO-variables (see 4.5).

Conflicts:

In the standard, a DO loop need not be executed. The iteration count is given by MAX(INT((*m2* - *m1* + *m3*)/*m3*),0). The DO loop is not executed if *m1* > *m2* and *m3* > 0 or if *m1* < *m2* and *m3* < 0. In level 8R1, a DO loop is always executed. The iteration count is given by MAX(((*e2* - *e1*)/*e3* + 1),1). If *e1* > *e2* and *e3* is omitted, level 8R1 assumes *e3* = -1, and level 9R1 assumes *e3* = +1 (see 4.5.4.1). The STD=66 option of the COMPILER statement provides for compatibility with previous levels of ASCII FORTRAN for DO loops (see 8.5 and 8.5.6).

J.12. Input/Output Statements

Extensions:

- The internal unit identifier is a character variable or character array or character array element or character substring that specifies an internal file. Level 8R1 does not allow a character substring for an internal unit identifier (see 5.9.1 and 5.9.3).
- An empty I/O list is allowed on reads or writes to skip a record or to write an empty record. Level 8R1 compiler requires a nonempty I/O list on list-directed reads, unformatted sequential-access writes, list-directed write or print, and unformatted direct-access writes. Errors are issued when the list is missing (see 5.6.1.4, 5.6.2.2, 5.6.2.4, and 5.7.3).
- Character substring names are allowed in I/O lists. Level 8R1 only allows them in output lists (see 5.2.3).
- Character constants produced by list-directed output are not delimited by apostrophes, are not preceded or followed by a blank or comma, and do not have internal apostrophes represented by two apostrophes.

- The implied-DO list parameters are the same as the new DO-loop parameters (that is, more types, zero iterations possible). Level 8R1 doesn't allow the new parameters (see 4.5 and 5.2.3).
- The OPEN statement is not in level 8R1 (see 5.10.1).
- The CLOSE statement is not in level 8R1 (see 5.10.2).
- The INQUIRE statement is not in level 8R1 (see 5.10.3).

Conflicts:

Character variables are only blank filled to the declared size of the variable during assignment statements and I/O. Pre-level 9R1 compilers and I/O systems performed blank fill to word boundaries even though the character variable was not a multiple of four characters. The STD=66 option does not change this incompatibility. Pre-level 9R1 absolute elements that use the FORTRAN common bank C2F$ will not execute as before. That is, the old compiler will blank fill to a word boundary, but formatted I/O will not blank fill to a word boundary. Character comparisons of the data will not find any equal conditions.

J.13. Format Specification

Extensions:

If the output format is G*w*.*d*E*e* and the value of the variable fits an F format, the format used is F(*w*-(*e*+2)).*d*-*i*,(*e*+2)('*b*') where *b* is a blank. The format is F(*w*-4).*d*-*i*,4('*b*') in level 8R1 (see 5.3.1).

Conflicts:

- During list-directed input, if the first record read in a read operation has no characters preceding the first value separator, this indicates a null field. In level 8R1 it does not indicate a null field but is handled the same as any other record. If you use the STD=66 option in the COMPILER statement, execution chooses level 8R1 and earlier methods of input.
- During list-directed output, character constants always have the PRINT format (that is, no apostrophes around character output). In level 8R1, PRINT and WRITE have different formatting in that apostrophes are used during the WRITE. Also on list-directed output in level 9R1 and higher, a complex constant must be written on one record if it fits, by itself, on a record. Level 8R1 breaks it up without checking to see if it fits on one line. If you use the STD=66 option of the COMPILER statement, execution proceeds with level 8R1 and earlier types of output.

J.14. Main Program

Extensions:

The PROGRAM statement to name a main program must be the first statement in the main program if it occurs. The PROGRAM statement is not implemented in level 8R1 (see 7.9).

Conflicts:

None.

J.15. Functions and Subroutines

Extensions:

- The following intrinsic functions are not in level 8R1: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, LLT, UPPERC, LOWERC, and TRMLEN (see 7.3.1).

- Extended conversion intrinsic functions are the following: INT, REAL, DBLE, and CMPLX for all argument types. Level 8R1 gives warnings for use of complex with INT and proceeds to flag further uses of INT as a user function. FORTRAN 77 allows integer, real, complex, and double-precision arguments for INT. Level 8R1 gives warnings for any use of REAL with variables other than complex and proceeds to flag further uses of REAL as a user function. FORTRAN 77 allows integer, real, and complex as arguments for REAL. Level 8R1 gives warnings for any use of DBLE with complex variables, and sets further calls of DBLE to a user function. FORTRAN 77 allows integer, real, double-precision, and complex arguments for DBLE. Level 8R1 gives warnings for any complex variables used as arguments of CMPLX; it makes further uses of CMPLX become calls to a user function. FORTRAN 77 allows integer, real, double-precision, and complex arguments for CMPLX (see 7.3.1).

- The FUNCTION statement has the length for a character function as CHARACTER[** *length*]C(A) while level 8R1 allows it after the function name, that is, CHARACTER FUNCTION C*3(A) (see 7.4.3.2). Both forms are allowed in level 9R1 and higher.

- Empty parentheses are allowed on the SUBROUTINE statement. The FORTRAN 77 form is SUBROUTINE *sub* [([*d*[,*d*]. . .])]; the level 8R1 form is SUBROUTINE *sub* [(*d*[,*d*] . . .)] . The forms *sub* and *sub*( ) are equivalent (see 7.4.4.2).

- An actual argument for a subroutine call can be **s* , where *s* is a statement number. Level 8R1 uses currency signs ($) or ampersands (&) for the statement number (see 7.2.1 and 7.2.2).

- A dummy argument array name can be associated with an actual argument which is an array element substring as well as an array or array element. Level 8R1 passes a temporary for an array element substring.

- A dummy argument that becomes defined can be associated with a substring as an actual argument. Level 8R1 associates it with a variable, an array element, or an expression.

- Empty parentheses are allowed in the ENTRY statement. The FORTRAN 77 form is ENTRY *en* [([*d*[.*d*]. . .])] while the level 8R1 form is ENTRY *en* [(*d*[.*d*]. . .)] . FORTRAN 77 requires that the function be specified with the form *en*( ) even if the entry statement did not have the empty set of parentheses (see 7.7).

Conflicts:

- Statement functions are typeless in level 8R1, but are typed (just like other functions) in level 9R1 and higher. This can cause different results on account of implied type conversions (see 6.3 and 7.4.1). The STD=66 option of the COMPILER statement provides for compatibility of level 9R1 and higher with previous levels of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.6).

- Register A1 contains a function packet address for character function references (see K.4.6). For level 8R1 ASCII FORTRAN, register A1 contains a result address for character function references. This is an incompatibility between level 8R1 and all higher levels. If a program compiled with level 9R1 or higher refers to a character function program compiled by level 8R1, the STD=66 option of the COMPILER statement must be present in the level 9R1 or higher program.

- If the value of *e* in RETURN[*e*] is less than one or greater than the number of asterisks in the subroutine entry, in level 9R1 control returns to the CALL statement that initiates the subprogram reference. In level 8R1, an error is issued and the program continues with unknown results (see 7.6).

J.16. Block Data Subprogram

Extensions:

An optional global name can be specified for a block data subprogram, that is, BLOCK DATA [*name*]. Level 8R1 doesn't allow the optional name.

Conflicts:

None.

Appendix K. Interlanguage Communication

K.1. ASCII FORTRAN (FTN) to SPERRY FORTRAN V

The FORTRAN V subprogram must be declared as:

EXTERNAL *a*(FOR)

or:

EXTERNAL **a*

where *a* represents the FORTRAN V subprogram name.

The call to the FORTRAN V subprogram appears syntactically exactly as though it is a call to an ASCII FORTRAN subprogram.

Restrictions and Considerations:

- If both the ASCII FORTRAN and FORTRAN V programs perform I/O operations, the ASCII FORTRAN library element F2FCA must be reassembled. The allocation and releasing of buffer areas is not common between the two FORTRANs and I/O operations may fail. This problem is resolved by reassembling element F2FCA with the required amount of main storage. Refer to G.7 for determining the required amount.
- FORTRAN V Series E subprograms are restricted to symbiont types of I/O and the tag, CLOST$, will be undefined at collection time. This can be ignored since CLOST$ is defined in FORTRAN V Series T.
- Any files opened by FORTRAN V Series T must be explicitly closed using a CALL CLOSE statement in a FORTRAN V subprogram if they are to be usable after program termination.
- The FORTRAN V subprogram cannot call the EXIT service routine.
- The FORTRAN V subprogram name can't appear in a BANK statement.
- FORTRAN V subprogram arguments and function names should not be type character or double-precision complex as FORTRAN V supports neither data type.
- The walkback mechanism doesn't work if a walkback is attempted from a FORTRAN V subprogram to an ASCII FORTRAN program.

- Common blocks or local variables shared with or passed to FORTRAN V routines should not be in virtual or banked space.

K.2. ASCII FORTRAN to PL/I

The PL/I external procedure must be declared as:

EXTERNAL *a* (PL1)

where *a* represents the PL/I procedure name.

The call to the PL/I procedure appears syntactically exactly as though it is a call to an ASCII FORTRAN subprogram.

K.2.1. Restrictions and Considerations

- Level 8R1 PL/I, or later, must be used.
- Any file opened by a PL/I procedure must be closed by a PL/I procedure. Files can be shared between ASCII FORTRAN and PL/I, but they must be closed by the language which opened them before they can be accessed by the other language. A file that is opened by a given language does not have to be closed before switching to another language as long as the called language routine does not access the file.
- The PL/I procedure name can't be a dummy argument name.
- An argument to a PL/I procedure cannot be a label, subprogram name, or array name. An array element can be passed from ASCII FORTRAN to PL/I, but PL/I must declare its counterpart as a single data item; structures or cross-sections of arrays are not allowed.
- Passing an array name from ASCII FORTRAN to PL/I can be accomplished through common blocks. The PL/I procedure must use the EXTERNAL attribute on the declaration, and the ASCII FORTRAN program must specify the array in a named common block. The PL/I common block name is the variable name with the EXTERNAL attribute. ASCII FORTRAN stores arrays in column-major order while PL/I stores them in row-major order. This means that either the ASCII FORTRAN program must transpose the array so that it is in row-major order when the PL/I procedure is called or the PL/I procedure must refer to the array with the subscripts in reverse order and the array dimensioned in reverse order.

  For example:

ASCII FORTRAN	PL/I
EXTERNAL PL1SUB(PL1)	PL1SUB: PROC;
INTEGER ARR(2,4,6)	DCL 1 BLK1 EXTERNAL ALIGNED,
COMMON/BLK1/ARR	2 ARR(6,4,2) FIXED DECIMAL(10,0);
ARR(2,3,4) = 234	PUT SKIP ('WANT 234 :', ARR(4,3,2));
CALL PL1SUB	PUT SKIP;
END	END;

- If execution is stopped by the PL/I procedure using the STOP statement, any files opened by ASCII FORTRAN are not properly closed unless the ASCII FORTRAN program explicitly closes them using the CLOSE statement.

- Common blocks or local variables shared with or passed to PL/I should not be in virtual or banked space.

K.2.2. PL/I Argument Counterparts

PL/I has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	PL/I	Comment
INTEGER	FIXED BINARY (*p,q*) FIXED DECIMAL (*p,q*)	*p* can range 1-35, *q* must be 0. *p* can range 1-10, *q* must be 0.
REAL	FLOAT BINARY (*p*) FLOAT DECIMAL (*p*)	*p* can range 1-27. *p* can range 1-8.
DOUBLE PRECISION	FLOAT BINARY (*p*) FLOAT DECIMAL (*p*)	*p* can range 28-60. *p* can range 9-18.
COMPLEX	FLOAT BINARY COMPLEX (*p*) FLOAT DECIMAL COMPLEX (*p*)	*p* can range 1-27. *p* can range 1-8.
COMPLEX*16	FLOAT BINARY COMPLEX (*p*) FLOAT DECIMAL COMPLEX (*p*)	*p* can range 28-60. *p* can range 9-18.
LOGICAL	BIT (36) ALIGNED	Only the rightmost bit is used by ASCII FORTRAN. The PL/I string may not be of varying length.
CHARACTER*n	CHARACTER (*n*)	The PL/I string may not be of varying length.

K.3. ASCII FORTRAN to ASCII COBOL (ACOB)

The ASCII COBOL (ACOB) subprogram must be declared as:

EXTERNAL *a* (ACOB)

where *a* represents the ACOB subprogram name.

The call to the ACOB subprogram appears syntactically exactly as though it is a call to an ASCII FORTRAN subprogram.

Restrictions and Considerations:

- ACOB level 4R2, or higher, must be used.
- Any file opened by an ACOB subprogram must be closed by an ACOB subprogram. Files can be shared between ASCII FORTRAN and ACOB, but they must be closed by the language that opened them before they can be accessed by the other language. A file that is opened by a given language doesn't have to be closed before switching to another language as long as the called language routine doesn't access the file.

- The ACOB subprogram name can't be a dummy argument name.

- It is your responsibility to ensure the data alignment is the same for an ASCII FORTRAN argument and its ACOB counterpart. Special care must be taken when passing character type data to ACOB. If the ASCII FORTRAN argument is not word-aligned, the ACOB argument declaration must reflect the offset via the use of a structure. To ensure that an ASCII FORTRAN character scalar or array is word-aligned, place it as the first item in COMMON or equivalence the character item to an integer variable.

- The ACOB subprogram name must not be a function name since ACOB doesn't support functions.

- An argument to an ACOB subprogram must not be a label or subprogram name since ACOB has no argument counterpart.

- An array name can be passed from ASCII FORTRAN to ACOB as an argument. However, ASCII FORTRAN stores arrays in column-major order while ACOB stores them in row-major order. This means either the ASCII FORTRAN program must transpose the array so that it will be effectively in row-major order when the ACOB subprogram is called, or the ACOB procedure must reference the array with the subscripts in reverse order and the array dimensioned in reverse order. Beware of ASCII FORTRAN and ACOB alignment conventions.

  Example:

ASCII FORTRAN

```
EXTERNAL C(ACOB)
CHARACTER*5 ARR(2,4,6)
EQUIVALENCE (ARR(1,1,1),IDUM)
ARR(2,3,4) = 'ABCD'
CALL C(ARR)
PRINT *, 'WANT EFG:', ARR(2,3,4)
END
```

ACOB

```
LINKAGE SECTION.
01  BUFF.
    02 BUFA OCCURS 6 TIMES.
    03 BUFB OCCURS 4 TIMES.
    04 BUFC PIC X(5) OCCURS 2 TIMES.
PROCEDURE DIVISION USING BUFF.
C.
    DISPLAY 'WANT ABDC:',
     BUFC (4, 3, 2).
    MOVE 'EFG' TO BUFC (4, 3, 2).
EXIT PROGRAM.
```

- If execution is stopped by the ACOB subprogram, any files opened by ASCII FORTRAN don't close properly unless the ASCII FORTRAN program explicitly closes them via the CLOSE statement.

- If ASCII COBOL passes a group item with no explicit type to an ASCII FORTRAN program, the corresponding ASCII FORTRAN argument can be type INTEGER, REAL, DOUBLE PRECISION, or LOGICAL. CHARACTER type is not allowed. ASCII FORTRAN can only access that portion of a COBOL group item that is declared, either explicitly or implicitly, by the ASCII FORTRAN program.

- Common blocks or local variables shared with or passed to ACOB should not be in virtual or banked space.

K.3.1. ASCII COBOL Argument Counterparts

ASCII COBOL (ACOB) has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	ACOB	Comment
INTEGER	PIC S9(10) COMP SYNC	Ensure that the ACOB item is word-aligned.
REAL	COMP-1	
DOUBLE PRECISION	COMP-2	
COMPLEX	No ACOB counterpart	
LOGICAL	PIC 1 (36) SYNC	Only the rightmost bit is used by ASCII FORTRAN. Ensure that the ACOB item is word-aligned.
CHARACTER*(*n*)	PIC X(*n*)	Ensure that the alignment is the same for ASCII FORTRAN and ACOB.
TYPELESS †	PIC 1 (36) SYNC	Ensure that the ACOB item is word-aligned.

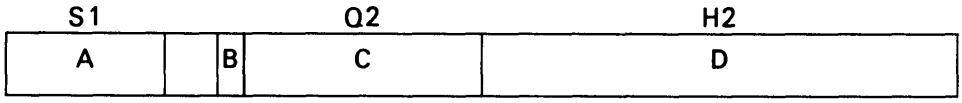
† *A typeless argument results from a typeless function, see 2.2.3.4.1.*

K.4. ASCII FORTRAN and MASM Interfaces

This section provides information needed when writing Assembler routines that call or are called by ASCII FORTRAN routines.

K.4.1. Arguments

For procedure calls with one or more arguments, ASCII FORTRAN requires an argument list. The address of the argument list is in H2 of register A0. A0 also contains the number of character arguments in S1 and the total number of arguments in Q2. Bit number 9 of A0 (left-most bit is bit 1) specifies when argument type checking is desired by the caller. The following is the format of register A0 for calling an ASCII FORTRAN program.



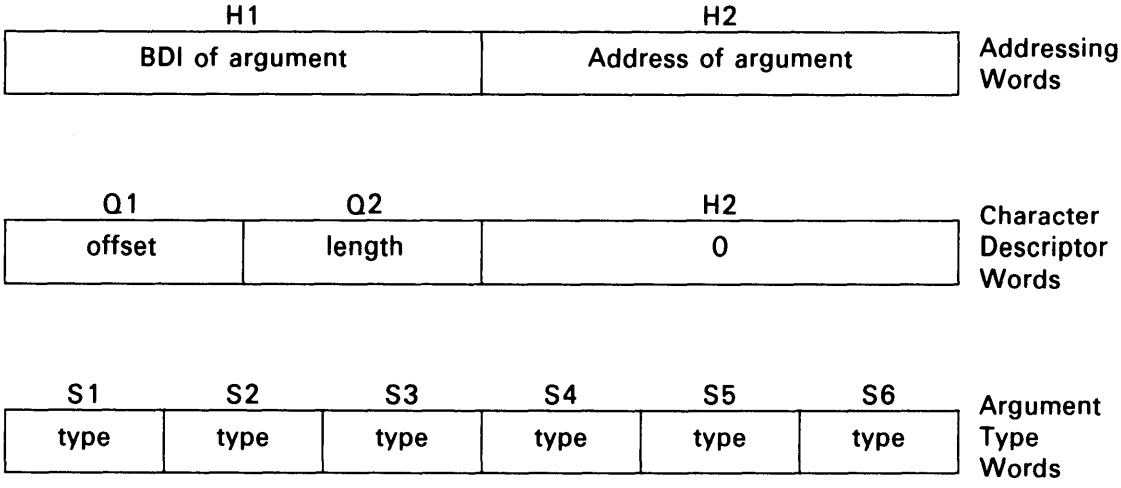
where:

A (S1 of A0) is the number of character arguments (maximum of 63).

B (Bit 9 of A0; assume bits are numbered 1-36 and the left-most bit is 1) is the argument type checking bit. If set to 1 by the caller, argument type checking will not be done. If set to 0, argument type checking will be done unless the called subprogram has disabled type checking. The called subprogram can disable type checking by using the COMPILER statement option ARGCHK=OFF or by compiling the called subprogram with optimization (Z or V option).

C (Q2 of A0) is the total number of arguments (maximum is 250).

D (H2 of A0) is the address of the argument list descriptor words that are explained in the following discussion.



The addressing words follow one another consecutively in storage. There is one addressing word for each argument. The bank descriptor index (BDI) is required if the ASCII FORTRAN subprogram being called expects banked arguments and the argument is not in the control D-bank. It is also required if the argument is an external subprogram name and the ASCII FORTRAN subprogram being called has the LINK=IBJ$ or BANKED=ALL compiler statement options present. Otherwise, the BDI is zero. If an ASCII FORTRAN program calls a MASM routine with arguments that have a BDI associated with them (that is, the actual data passed resides in a banked common block), the MASM routine is responsible for basing the argument's data bank. All D-bank basing must be done by an ASCII FORTRAN library routine in element F2ACTIV$. In other words, no LDJ or LBJ instructions should appear in your assembly code to switch the utility D-bank basing. The interface to the activate routine is:

```
LA      A0, addreswd. GET BDI and address in A0
LMJ     X11,VACTIV$ . base D-bank
```

A0 now contains the item's absolute address, and its bank is based. All registers except A0 and X11 are preserved. For details on ASCII FORTRAN banked programs, see Appendix H.

The character descriptor words follow one another consecutively in storage and follow the addressing words. A character function name or a Hollerith string passed as an argument does not have a character descriptor word. All other character types of arguments have a character descriptor word. The offset is the byte offset of the start of the character item within the word and has the value 0, 1, 2 or 3. If the character item begins on a word boundary, the offset is zero. A character item beginning on Q2 of the word has an offset of 1, an item beginning on Q3 has an offset of 2, and an item beginning on Q4 has an offset of 3. The character length, represented in number of characters, is in Q2 of the character descriptor word. If a character array is passed as an argument, the length passed is the size of an array element. If a character substring is passed as an argument, the length passed is the length of the substring.

If argument type checking is desired, bit 9 of A0 (assume bits numbered from 1 to 36 and the left-most bit is 1) must be zero and the argument type words must be present. The argument type words follow one another consecutively in storage and follow the character descriptor words. There is one type word for each six arguments. The following is a list of allowable types:

0 Subprogram
1 Integer
2 Real
3 Double-Precision Real
4 Complex
5 Double-Precision Complex
6 Character
7 Logical
8 Label
9 Hollerith

A type 0 subprogram, matches any other type. Type 9, Hollerith, matches all types except character and label. All other types must match exactly or else a run-time diagnostic message is issued when type checking is enabled. The argument type words are optional. If they are not present, either the caller must specify so by setting bit 9 of A0 to 1 or the callee must disable type checking by using the COMPILER statement option ARGCHK=OFF or compiling with optimization.

If an Assembler routine refers to an ASCII FORTRAN subprogram that has the COMPILER statement option STD=66 present or is compiled by an ASCII FORTRAN compiler lower than level 9R1, the contents of A0 and the packet format differ. S1 and bit 9 of A0 are ignored. The character descriptor words and the argument type words are not required. In addition, any character item passed to the ASCII FORTRAN subprogram must begin on a word boundary.

K.4.2. ASCII FORTRAN Register Usage

An ASCII FORTRAN subprogram saves and restores all registers that it uses except for the volatile set X11, A0-A5 and R1-R3. An ASCII FORTRAN subprogram requires register R15 to contain the address of a storage control table (F2SCT) which is used on I/O operations and by several of the ASCII FORTRAN library routines. Register R15 is loaded with the F2SCT address as part of the initialization performed by an ASCII FORTRAN main program. Once R15 has been initialized, it is the responsibility of any routine outside the ASCII FORTRAN environment to preserve its contents upon reentry to an ASCII FORTRAN subprogram.

K.4.3. Initializing the ASCII FORTRAN Environment

Under normal ASCII FORTRAN conditions, the ASCII FORTRAN environment is initialized by a call from the ASCII FORTRAN main program to the ASCII FORTRAN initialization routine. If an ASCII FORTRAN subprogram is called from an Assembler routine and there is not an ASCII FORTRAN main program, it is the responsibility of the Assembler routine to call the ASCII FORTRAN initialization routine. One of two ASCII FORTRAN initialization routines must be called by the Assembler routine before the ASCII FORTRAN subprogram is called. The ASCII FORTRAN initialization need only be called once during the program execution. The initialization routines use only the volatile set of registers. Both initialization routines acquire buffer space and initialize tables that are used by I/O and ASCII FORTRAN library routines. This space is acquired by the common storage management system. If the Assembler routine or controlling program has its own storage management system, the ASCII FORTRAN library element F2FCA can be modified and reassembled to avoid any ER MCORE$. See Appendix G for details.

One of the initialization routines also registers a contingency routine for capturing contingency interrupts, which is required for the proper execution of some ASCII FORTRAN programs. However, since some applications prefer to capture their own contingencies, a second routine which does not register contingencies is provided. The ASCII FORTRAN service routines UNDSET, OVFSET, and DIVSET register contingencies and should not be called if an application depends on another contingency registration.

The following call initializes the ASCII FORTRAN environment but doesn't register the ASCII FORTRAN contingency routine.

```
LMJ    A2,FINT$
```

The following call initializes the ASCII FORTRAN environment and registers the ASCII FORTRAN contingency routine.

```
LMJ    X11,FINT2$
```

On return from either of the initialization routines, register R15 contains the address of the ASCII FORTRAN storage control table (F2SCT). It is the responsibility of the Assembler routine to ensure that R15 contains the F2SCT address when calling an ASCII FORTRAN subprogram.

K.4.4. Terminating the ASCII FORTRAN Environment

Under normal conditions, the ASCII FORTRAN environment is terminated by a call from the main program to the ASCII FORTRAN termination routine. The function of the termination routine is to output buffered I/O to the appropriate files and close all opened files. An ER EXIT$ is then performed which terminates the program. If an Assembler routine calls an ASCII FORTRAN subprogram and control never reaches an ASCII FORTRAN main program for normal program termination, it is the responsibility of the Assembler routine to close all opened I/O files. The closing of files can best be accomplished by having the ASCII FORTRAN subprogram close them using the CLOSE statement. If the ASCII FORTRAN termination routine is called, files are closed but control is not returned to the caller. The following is the Assembler call to the ASCII FORTRAN termination routine.

```
LMJ    X11,FEXIT$
```

K.4.5. Calling an ASCII FORTRAN Subprogram

A call to an ASCII FORTRAN subprogram takes one of several forms depending upon whether the subprogram is banked or not. For a nonbanked ASCII FORTRAN subprogram, call the following Assembler linkage can be used.

```
LXI,U   X11,0
LMJ     X11,entry-point
```

ASCII FORTRAN returns to the caller via:

```
J       0,X11
```

For a banked ASCII FORTRAN subprogram call (that is, the ASCII FORTRAN subprogram has the compiler statement options BANKED=ALL, BANKED=RETURN, or LINK=IBJ$), use one of two calling sequences:

```
LXI,U   X11,bdi-of-the-subprograms-bank
LIJ     X11,entry-point
```

or:

```
LXI,U   X11,BDICALL$+entry-point
IBJ$    X11,entry-point
```

For a description of IBJ$ and BDICALL$, see the Collector Reference, UP-8721 (applicable version).

An ASCII FORTRAN banked subprogram returns by:

```
LA,H1   A4,X11-save-location
JZ      A4,0,X11
LIJ     X11,0,X11
```

K.4.6. ASCII FORTRAN Function References

If an ASCII FORTRAN (level 9R1 or higher) character function is referred to, register A1 must be set up by the calling routine to contain:

H1	H2
0	*fctn-pkt-addr*

where *fctn-pkt* has the form:

H1		H2
0		*result-addr*
0	*char-length*	0

The *result-addr* field in H2 of the first word points to the caller's storage area where the result of the function will be stored. This storage area must be in the control bank or be visible to the function. It is the caller's responsibility to ensure that the function result area is large enough to hold the function result.

The *char-length* field in Q2 of the second word of the packet is the length of the function result expressed in number of characters.

For ASCII FORTRAN levels 8R1 and lower, and whenever the compiler statement option STD=66 is used in the function being called, register A1 must contain:

H1	H2
0	*result-addr*

The *result-addr* field description is the same as for levels 9R1 and higher. A *fctn-pkt* is not required for levels lower than 9R1.

An ASCII FORTRAN character function places the function result in the caller's storage area pointed to by *result-addr*. If the value of a function isn't a character string, it is returned in registers A0, A0 through A1, or A0 through A3, depending on the function type.

K.4.7. Example

The following is an example of an Assembler routine that passes three arguments to the ASCII FORTRAN subroutine FTEST.

```
1.          AXR$
2.  $(1)
3.  MATH*   LMJ     A2,FINT$            . initialize FTN
4.  .
5.          L,U     R4,4                . loop initialization count
6.  FTNREF  LA      A0,(020003,ARGS)    . list pointer
7.          LXI,U   X11,0               .
8.          LMJ     X11,FTEST           . call nonbanked FTN rtn
9.          LA      A4,REALNO           . bump 2nd arg by 1.0
10.         FA      A4,(1.0)            .
11.         SA      A4,REALNO           .
12.         JGD     R4,FTNREF           . repeat call to FTN rtn
13. .
14.         LMJ     X11,FEXIT$          . terminate program
15. .
16. $(0)
17. CHARD   FORM    9,9,18              .
18. .
19. ARGS    +       STRING              . address of 1st arg
20.         +       REALNO              . address of 2nd arg
21.         +       STRING              . address of 3rd arg
22.         CHARD   0,3,0               . 'ASM' offset=0 len=3
23.         CHARD   3,3,0               . 'B17' offset=3 len=3
24.         +       6,2,6,0,0,0         . char real char type
25. .
26.         ASCII
27. STRING  'ASMB17'                    .
28. REALNO  +       2.5                 .
29.         END     MATH                .
```

The following is the ASCII FORTRAN subroutine FTEST that is called from the preceding Assembler routine:

```
1.        SUBROUTINE FTEST (CALTYP, X, CALID)
2.        CHARACTER CALTYP*(*), CALID*3
3.  *
4.        PRINT *, 'CALLER ID ', CALID, ' TYPE ', CALTYP
5.        PRINT *, 'SIN OF', X, ' = ', SIN(X)
6.        PRINT *, 'COS OF', X, ' = ', COS(X)
7.        RETURN
8.        END
```

The execution of the program is:

```
CALLER ID B17 TYPE ASM
SIN OF  2.5000000    =  .59847214
COS OF  2.5000000    = -.80114362
CALLER ID B17 TYPE ASM
SIN OF  3.5000000    = -.35078323
COS OF  3.5000000    = -.93645669
CALLER ID B17 TYPE ASM
SIN OF  4.5000000    = -.97753011
COS OF  4.5000000    = -.21079580
CALLER ID B17 TYPE ASM
SIN OF  5.5000000    = -.70554033
COS OF  5.5000000    =  .70866977
CALLER ID B17 TYPE ASM
SIN OF  6.5000000    =  .21511999
COS OF  6.5000000    =  .97658762
```

At line 3 of the Assembler routine, a call is made to initialize the ASCII FORTRAN environment. The ASCII FORTRAN initialization routine acquires I/O buffer storage via the common storage management system. If the Assembler routine has its own storage management system, the ASCII FORTRAN library element F2FCA needs modifications and reassembling. See Appendix G for details.

At line 6, register A0 is loaded with a literal that specifies:

1. Two character arguments (S1)

2. Perform argument type checking (bit 9 of the literal is 0)

3. Three total arguments (Q2).

The second half of A0 contains the address of the argument list.

Lines 19 through 21 contain the argument addressing words. Lines 22 through 23 contain the character descriptor words for the first and third arguments, respectively, which are character types. The first argument passed is the character string ASM, the second argument passed is the real number 2.5 (which is modified after each call by the Assembler routine), and the third argument is the character string B17.

Line 24 contains the argument type word which the ASCII FORTRAN subprogram requires for argument type checking. The first and third arguments are character types as indicated by the 6, and the second argument is a real type as indicated by the 2.

At line 14, the ASCII FORTRAN termination routine is called. This closes any opened I/O file and then terminates program execution using an ER EXIT$.

Appendix L. ASCII FORTRAN Sort/Merge Interface

L.1. General

A sort/merge interface is available from ASCII FORTRAN to the sort/merge package. The sort/merge package is described in the Sort/Merge Reference, UP-7621 (applicable version).

L.2. Sort/Merge Features Available Through ASCII FORTRAN

Enter the sort/merge interface using the CALL statement in ASCII FORTRAN. The sort/merge interface provides the following functions:

- CALL FSORT — Perform a sort.
- CALL FMERGE — Perform a merge.
- CALL FSCOPY — Specify an Assembler sort parameter table to be copied. This table can be used in subsequent sorts or merges. The use of such a table can also be inhibited.
- CALL FSSEQ — Specify your collating sequence that is used in subsequent sorts or merges. The use of such a collating sequence can also be inhibited.
- CALL FSGIVE — Deliver an input record to sort without leaving your input subroutine.
- CALL FSTAKE — Receive an output record from sort without leaving the your output subroutine.

L.3. Restrictions With Sort/Merge Interface

L.3.1. Banked Arguments Not Allowed

The data given to any area of the sort/merge interface must not be banked. The scratch area used by the sort/merge interface must not be banked.

L.3.2. Sort/Merge Interface Contains Only Formatted I/O

The sort/merge interface attempts only formatted I/O on all logical unit numbers used in the calls to do a sort or a merge. The ASCII FORTRAN I/O complex doesn't test the file to determine if you have a formatted file. This can result in errors from the sort/merge interface.

L.3.3. Use of ASCII FORTRAN Free Core Area Element (F2FCA)

When the file R$CORE is not assigned to the run (for FSORT only), the sort/merge interface attempts to get storage space from the ASCII FORTRAN library common storage management system (CSMS). If you supply a version of ASCII FORTRAN library element F2FCA so that the CSMS routines are not used, you must make element F2FCA large enough to accommodate the storage area needed by the sort/merge interface routines, the sort/merge package, and the area needed for the FORTRAN library (see G.7).

L.3.4. Use of an Asterisk as a Dummy Character Argument Length

You can't use an asterisk as a length specification for the dummy character arguments for the following user-specified subroutines:

- Input
- Output
- Comparison
- Data reduction

The length specification must be an unsigned, nonzero integer constant, or an integer constant expression enclosed in parentheses that has a positive value.

L.4. The CALL Statement to FSORT

L.4.1. The CALL Statement for a Sort

The form of the CALL statement for a sort is:

CALL FSORT (*infost* ,*inpt* ,*outpt* [,*comprt*] [,*datrd*])

where:

infost is the information string, a character string that describes various parameters to the sort/merge interface, such as record sizes, key fields, and scratch facilities for FSORT. A key field (or your comparison routine) and a record size must be specified in *infost* .

Infost contains items of information separated by commas. Blanks in *infost* are ignored. No distinction is made between uppercase and lowercase alphabetic characters. *Infost* is scanned from left to right and must be terminated by some character which is an illegal ASCII FORTRAN character, such as an exclamation point (!). An asterisk (*) must not terminate *infost* .

Infost can contain several clauses. The mnemonics used are truncated by the sort/merge interface to the first four characters. The following items can be used in *infost* for the call to FSORT:

1. RSZ= *rlch*

   Rlch is the record length in ASCII characters. This record size must be specified when a sort is to be done. The RSZ clause can appear only once in *infost*. A record size clause can appear only once in *infost*; that is, the RSZ and VRSZ clauses can't appear in the same *infost*.

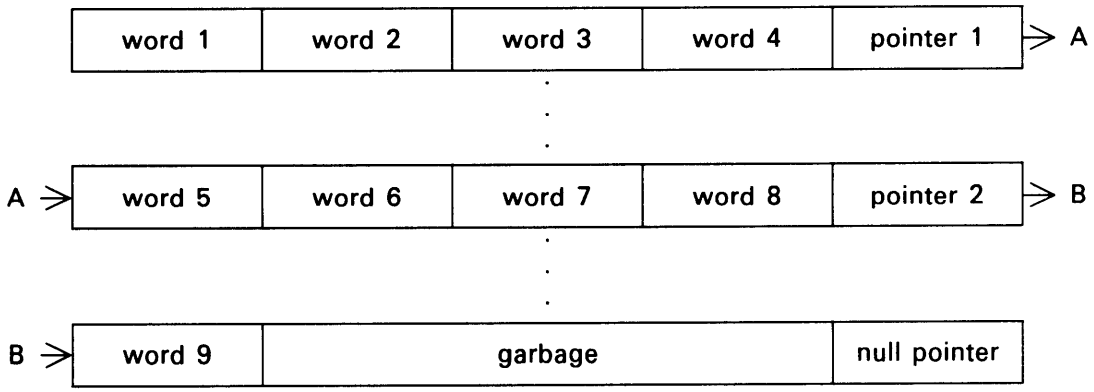
2. VRSZ= *mrlch / lnkszch*

   Mrlch is the maximum record size in ASCII characters for variable length records. *Lnkszch* is an optional parameter indicating link size in ASCII characters. When *lnkszch* is omitted, the slash (/) can also be omitted. *Lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

   KEY=(11/15,1/10/d/s)

   the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. The link size should be specified only when a comparison routine has been specified. The VRSZ clause can appear only once in *infost*. A record size clause can appear only once in *infost*; that is, the RSZ clause and the VRSZ clause can't both be used in the same *infost*.

   When sorting variable length records, the records are separated into smaller parts (links) of equal size that are joined by pointers. As an example, consider a record of nine words with the link size four words. Schematically, the record is stored as:



When the link size is given in the VRSZ clause, consider the following rules:

1. The link size should not be too small. For example, if the link size is given as one word, the core (main storage) required for each record is exactly twice the record size. This means that the sort uses many more resources (main storage, mass storage, and tapes) than necessary.

2. The link size should not be too large, since this may mean that much of the area remains unused in the last link. This causes problems because of poor use of main storage.

NOTE: The two rules are in conflict. The choice of a link size requires a compromise between these two rules.

It is frequently advantageous to sort short variable length records as if these records were fixed-length records because of the resources used by the sort/merge package. If these records are treated as fixed length, you must keep track of the record length.

3. CONS

CONS indicates that the closing messages from the sort/merge package are to be sent to the system console. If CONS is not present, the opening and closing messages from the sort/merge package are sent to the system log. The opening messages give the block sizes on mass storage and may be used to check the efficiency of the sort/merge usage of the scratch area. The closing messages give the input and output record counts and the bias of the data. The CONS specification can appear only once in *infost*.

4. DELL

DELL is used to indicate that the opening and closing messages from the sort/merge package should not be sent to the system log. This clause can appear only once in *infost*.

5. KEY=*keysp*

or:

KEY=*keyspn*

Keysp is a single key specification; *keyspn* is a multiple key specification of the form (*keysp* $_1$, *keysp* $_2$, . . .). The single key specification form, KEY=*keysp*, can occur a maximum of 40 times in *infost*. There may be a maximum of 40 *keysp* specifications within the *keyspn*. More than one KEY=*keyspn* clause can occur in *infost* but only 40 keys are allowed for each call to FSORT. The limit of 40 keys includes any keys given in the sort parameter tables copied through the COPY clause (see L.7.1). The key specification can indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

charpos / length / seq / type

where:

charpos is the position within the record of the most significant character of the key. Character positions are counted from left to right beginning with position 1.

length is an optional field that specifies the length of the key in characters. The default for *length* is 1.

seq is an optional field that specifies the sequencing order of the key. The value A is used for ascending order; D is used for descending. The default value is A.

type is an optional field that specifies the type of the key. The value of this field is B, Q, R, S, T, U, or V; U is the default value. The values for this field indicate:

B The key field contains a signed number in a Series 1100 system internal representation.

Q The key field contains a signed decimal number in 9-bit ISO character representation with a sign overpunched on the last digit.

R The key field contains 9-bit ISO characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, a minus, or a blank is set to a blank.

S The key field contains 9-bit characters.

T The key field contains 9-bit ISO characters with a sign in the last character position, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank.

U The key field contains an unsigned number in the Series 1100 system internal representation.

V The key field contains a signed decimal in 9-bit ISO characters with a sign overpunched on the first character.

The form of the bit key is:

BIT/ *wordpos / bitpos / length / seq / type*

where:

wordpos is the position within the record of the word that contains the most significant bit of the record. Words within the record are numbered from 1.

bitpos is the position of the first bit of the key in the first word of that key. Bits are numbered from left to right beginning at 1.

seq is an optional field that specifies the sequencing order of the key: A for ascending or D for descending. The default is A.

type is an optional field that specifies the type of the key. The values for *type* may be A, B, D, G, L, M, P, or U; the default is U. The values for *type* indicate:

A The key field contains 6-bit characters. All A key fields must start and end on 6-bit byte boundaries.

B The key field contains a signed number in the Series 1100 system internal representation.

D The key field contains 6-bit Fieldata characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. All D key fields must start and end on 6-bit byte boundaries.

G The key field contains 6-bit Fieldata characters with a sign in the last character, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. The key field must begin and end on a 6-bit byte boundary.

L The key field contains a number in 6-bit Fieldata characters with a sign overpunched on the first digit. The key field must start and end on a 6-bit byte boundary.

M The key field contains a number in signed magnitude representation. This means that the first bit is the sign (that is, a 1 for negative and a 0 for positive), and the rest of the field is the absolute value of the number.

P The key field contains a signed decimal number in 6-bit Fieldata characters with a sign overpunched on the last digit. The key field must begin and end on a 6-bit byte boundary.

U The key field contains an unsigned number in Series 1100 system internal representation.

6. COMP

COMP indicates the use of your comparison routine. This indicates the presence of *comprt* in the call to FSORT. The COMP clause can appear only once in *infost*.

7. COPY

COPY indicates that an Assembler sort parameter table is to be copied. The COPY clause can appear only once in *infost*. See L.6.1.

8. DATA

DATA indicates that your data reduction routine is present. This indicates the presence of *datrd* in the call to FSORT. This option can only be specified when fixed length records are sorted. The DATA clause can appear only once in *infost*.

9. SELE= *recno1*

or:

SELE= *recno1 / recno2*

or:

SELE=(*recno1* [/ *recno2*] ,*recno3* [/ *recno4*] , . . .)

Recno1 through *recno4* are record numbers. The SELE, or select, clause indicates which records are given to the sort/merge package. If the first form is used, only the record specified by *recno1* is given to the sort. If the second form is used with *recno2*, all records from *recno1* through *recno2* are given to the sort. If the third form of the SELE clause is used, the records between each pair of record numbers are given to the sort and single records are given to the sort. All records are read, but only those records specified in the SELE clause are given to the sort. Only 10 record number pairs can be used in the third form. For each pair, *recno1* must be less than or equal to *recno2*, and the last number of each pair must be less than the first number of the next pair. If *recno1* appears without *recno2*, or *recno3* appears without *recno4*, only *recno1* or *recno3*, respectively, are given to the sort. This clause can appear only once in *infost*.

10. CORE= *corsz*

Corsz is the size in words of the scratch area to be used by the sort. At least 3000 words must be used. In general, the sort runs faster if the scratch area given to sort is expanded. This clause can appear only once in *infost*. See L.9.2.

11. FILE= *file-name*

or:

FILE=(*file-name* , *file-name* , . . .)

File-name is a Series 1100 system internal file name. The second form of the FILE clause permits the specification of more than one file name within the clause. See L.9.3.1. The following restrictions apply to scratch files:

1. All scratch files must be assigned when the sort starts executing.
2. A maximum of 26 scratch files can be specified.
3. At least three tape scratch files must be used if any tape scratch files are used.
4. If tape scratch files are used, a maximum of two mass storage scratch files can be used for the sort.

12. NOCH= *chksm*

Chksm is any combination of the letters D, F, K, and T. The letter T refers to tape and D, F, and K refer to mass storage. The nocheck clause is used to omit a checksum. When the sort uses one or two mass storage scratch files, D refers to the smaller of the two (one must be at least twice the size of the other) and F refers to the larger of the two, if both files are present.

If the sort uses three or more mass storage scratch files, K refers to the checksum on all the files.

If K is specified for a sort with fewer than three mass storage files, D and F are assumed. If D or F is specified for a sort with more than two mass storage scratch files, K is assumed. This clause can appear only once in *infost*. See L.9.3.2.

13. MESH=*meshsz / device*

    Meshsz is the mesh size and *device* is any combination of the letters D, F, K, and T. If *meshsz* is not given, the value 5 is assumed. If *device* is not present, the mesh size is assumed to apply to all device types. The letter T indicates the use of tape scratch files; D, F, and K indicate the use of mass storage scratch files. D and F are used if one or two mass storage scratch files are used. D refers to the smaller of the two mass storage scratch files and F refers to the larger of these two files. The letter K indicates the checksum of three or more mass storage scratch files. The MESH= specification can appear only once in *infost*. The letters D, F, K, and T can be used only once each in the MESH specification. See L.9.3.2.

14. BIAS=*biasno*

    Biasno is the average number of records in sorted subsequences present in the input file. For example, *biasno* is 1 if the input is in exactly reverse order. For random data, *biasno* is 2. If the input file is in an almost sorted order, the bias value is higher. The BIAS clause can appear only once in *infost*. Giving the bias value, if known, improves the performance of the sort substantially. See L.9.1.

An example of an information string *infost* to sort variable-length records with a maximum length of 200 characters with four key fields is:

```
'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d),CONS!'
```

The key fields in the record are defined as:

1. The first key starts in the first character position of the record, is 10 characters long, and is sorted in ascending order with a user-specified collating sequence (if that sequence is present).

2. The second key starts in character position 11, is 10 characters long, and is sorted in descending order with an overpunch in the last character position.

3. The third key starts in character position 21, has a length of 10 characters, and is sorted in ascending order.

4. The fourth key starts in character position 31, has a length of 10 characters, and is sorted in descending order.

The CONS clause is present so that all messages are sent from the sort/merge package to the console.

To sort record images of 80 characters with a key that starts in character position 1, that has a length of 5 characters, and that is sorted in ascending order, the information string *infost* can be:

```
'RSZ=80,KEY=1/5!'
```

Infost must be the first parameter in the call to FSORT. The other parameters follow *infost*.

inpt is either a logical unit number or the name of an input subroutine. If *inpt* is the name of an input subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.1.

outpt is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.4.

comprt is the name of a comparison subroutine supplied by you. The subroutine is called whenever two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if you have not provided a comparison subroutine. This parameter must be present if the COMP clause occurs in *infost*. See L.8.2.

datred is the name of a data reduction subroutine. The name of *datred* must be declared in an EXTERNAL statement. This subroutine is called whenever two records with equal keys are found. It decides whether the two records are merged into one record or are not merged. This feature sorts fixed length records only. *Datred* must not be present if you have not specified the DATA clause in *infost*; it must be present if the DATA clause occurs in *infost*. See L.8.3.

L.4.2. Examples of Sort With Logical Unit Numbers

The following runstream contains a call to FSORT with a simple information string *infost* that contains a KEY clause and an RSZ clause. The RSZ clause gives a record size of 80 characters. The KEY clause states that the key begins in the first character position of the record, has a length of five characters, and is sorted in ascending order. *Infost* ends with an exclamation point (!). The input and output parameters are simply unit numbers 5 and 6. The sort/merge interface does formatted reads on unit 5 until all the input data is read. The interface then calls the sort/merge package to sort the data, and does formatted writes on unit 6 of the data from the sort/merge package.

```
@RUN
@FTN,SI

      CALL FSORT('key=1/5,RSZ=80!',5,6)
      END

@MAP,SIF
LIB   ASCII*FTNLIB.
@XQT
. . . data images to be sorted . . .
@FIN
```

NOTE: The RSZ clause should not be greater than 1,024 characters when you use FSORT with logical unit numbers. If the RSZ clause is greater than 1,024 characters, the output file is larger than the input file. Utilize user-supplied input and output routines to handle larger record sizes so that the input and output files can be the same size.

Another example of a simple sort appears in the following runstream. This program assumes that the source input from file IN*PUT is written to the file OUT*PUT. The records are 80 characters long with keys starting in character positions 1 and 6. Each key is five characters long. The first key is sorted in ascending order, and the second key in descending order.

```
@RUN
@FTN,SI

      CALL FSORT('rsz=80,key=(1/5,6/5/d)!',9,10)
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,A  IN*PUT
@ASG,C  OUT*PUT
@USE  9,IN*PUT
@USE  10,OUT*PUT
@XQT
@FIN
```

L.4.3. Examples of Sort With User Subroutines

The following example is a simple variation of the first example in L.4.2. The RSZ clause declares the record size to be 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of five characters, and is sorted in ascending order. The *inpt* and *outpt* parameters are user-supplied input and output subroutines that are declared in an EXTERNAL statement in the program. The subroutines contain formatted I/O statements to read from unit 5 and write to unit 6.

```
@RUN
@FTN,SI

      EXTERNAL IN,OUT
      CALL FSORT(key=1/5,rsz=80!',IN,OUT)
      END

@FTN,SI  IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(5,1,END=2) RECORD
      LENGTH=80
      IEOF=0
      RETURN
1     FORMAT(20A4)
2     IEOF=1
      RETURN
      END

@FTN,SI  OUT

      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*4 RECORD(20)
      IF (LENGTH.GE.0) WRITE(6,1) RECORD
1     FORMAT(1X,20A4)
```

```
      RETURN
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@XQT
. . . data images to be sorted . . .
@FIN
```

Another example of a sort with user I/O routines appears in the following runstream:

```
@RUN
@FTN,SI

      EXTERNAL IN,OUT
      CALL FSORT('rsz=80,key=(1/5,6/5/d),core=20000!',IN,OUT)
      END

@FTN,SI  IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(9,1,END=2) RECORD
1     FORMAT(20A4)
      LENGTH=80
      IEOF=0
      RETURN
2     IEOF=1
      RETURN
      END

@FTN,SI  OUT

      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*4 RECORD(20)
      IF (LENGTH.LT.0) GO TO 2
      WRITE(10,1) RECORD
1     FORMAT(20A4)
      RETURN
2     ENDFILE 10
      RETURN
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,A  IN*PUT
@ASG,C  OUT*PUT
@USE  9,IN*PUT
@USE  10,OUT*PUT
@XQT
@FIN
```

L.5. The CALL Statement to FMERGE

L.5.1. The CALL Statement for a Merge

The form of the CALL statement for a merge is:

CALL FMERGE (*infost*, *inpts*, *outpt* [,*comprt*])

where:

infost is the information string, a character string that describes various parameters to the sort/merge interface, such as record sizes, key fields, and scratch facilities for FMERGE. A key field (or your comparison routine) must be specified in *infost*.

Infost contains items of information separated by commas. Blanks in *infost* are ignored. No distinction is made between uppercase and lowercase alphabetic characters. *Infost* is scanned from left to right. *Infost* must be terminated by some character which is an illegal ASCII FORTRAN character, such as an exclamation point (!), but don't use an asterisk (*).

Infost can contain several clauses. The mnemonics are truncated by the sort/merge interface to the first four characters. The following items can be used in *infost* for the call to FMERGE:

1. RSZ=*rlch*

   Rlch is the record length in ASCII characters. This record is not required. If the RSZ clause is not given, the sort/merge interface assumes that the maximum record length is 1,000 words. This wastes some main storage. If the RSZ clause is present, the interface checks that the records from each input source are in sequence. The RSZ clause can appear only once in *infost*. Only one record size clause can appear in *infost* at a time. Thus, the VRSZ clause can't be used if the RSZ clause is used in *infost*.

2. VRSZ=*mrlch / lnkszch*

   Mrlch is the maximum record size in ASCII characters for variable-length records. *Lnkszch* is an optional parameter indicating link size in ASCII characters. If *lnkszch* is omitted, the slash (/) is also omitted. *Lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

   KEY=(11/15,1/10/d/s)

   the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. Specify the link size only when a comparison routine is specified. The sort/merge checks to see if the keys fit in the link size but otherwise ignores the link size for a merge. The VRSZ clause can appear only once in *infost* and only one record size clause can be specified in *infost* at a time. Thus, the RSZ clause can't be used if the VRSZ clause appears in *infost*.

3. KEY=*keysp*

or:

KEY=(*keyspn*)

Keysp is a single key specification and *keyspn* is a multiple key specification of the form $keysp_1$, $keysp_2$, etc. . . . . The single key specification form KEY=*keysp* can occur a maximum of 40 times in *infost*. There can be a maximum of 40 *keysp* specifications in *keyspn*. More than one KEY=*keyspn* clause can occur in *infost*, but only 40 keys are allowed for each call to FMERGE, including any keys copied through the COPY clause (see L.7.1). The key specification can indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

charpos / length / seq / type

where:

charpos is the position in the record of the most significant character of the key. Character positions are counted from left to right beginning with position 1.

length is an optional field that specifies the length of the key in characters. The default for the length is 1.

seq is an optional field that specifies the sequencing order of this key: A for ascending and D for descending. The default value is A.

type is an optional field that specifies the type of the key. The value of this field may be B, Q, R, S, T, U, or V; the default is U. The values for this field indicate:

B The key field contains a signed number in a Series 1100 system internal representation.

Q The key field contains a signed decimal number in 9-bit ISO character representation with a sign overpunched on the last digit.

R The key field contains 9-bit ISO characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank.

S The key field contains 9-bit characters.

T The key field contains 9-bit ISO characters with a sign in the last character position, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank.

U The key field contains an unsigned number in a Series 1100 system internal representation.

V The key field contains a signed decimal in 9-bit ISO characters with a sign overpunched on the first character.

The form of the bit key is:

BIT/ *wordpos / bitpos / length / seq / type*

where:

wordpos is the position in the record of the word that contains the most significant bit of the record. Words in the record are numbered from 1.

bitpos is the position of the first bit of the key in the first word of that key. Bits are numbered from left to right beginning at 1.

seq is an optional field that specifies the sequencing order of the key: A for ascending or D for descending. The default is A.

type is an optional field that specifies the type of the key. The values for *type* may be A, B, D, G, L, M, P, or U; the default is U. The values for *type* have the following meanings:

A The key field contains 6-bit characters. All A key fields must start and end on 6-bit byte boundaries.

B The key field contains a signed number in a Series 1100 system internal representation.

D The key field contains 6-bit Fieldata characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. All D key fields must start and end on 6-bit byte boundaries.

G The key field contains 6-bit Fieldata characters with a sign in the last character, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. The key field must begin and end on a 6-bit byte boundary.

L The key field contains a number in 6-bit Fieldata characters with a sign overpunched on the first digit. The key field must start and end on a 6-bit byte boundary.

M The key field contains a number in signed magnitude representation. This means that the first bit is the sign, that is, a 1 for negative and a 0 for positive, and the rest of the field is the absolute value of the number.

P The key field contains a signed decimal number in 6-bit Fieldata characters with a sign overpunched on the last digit. The key field must begin and end on a 6-bit byte boundary.

U The key field contains an unsigned number in Series 1100 system internal representation.

4. COMP

   COMP indicates the use of your comparison routine. This requires the presence of *comprt* in the call to FMERGE. The COMP clause can appear only once in *infost*.

5. COPY

   COPY indicates that an Assembler sort parameter table is to be copied. This clause can appear only once in *infost*. See L.6.1.

6. INPU=*inptsor*

   Inptsor is an integer constant from 2 through 24 that indicates how many input sources are given in the parameter *inpts*. The INPU clause must appear only once in *infost*.

An example of an information string *infost* that merges two files containing variable length records with a maximum length of 200 characters with four key fields is:

'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d), INPUT=2!'

The key fields in the record are defined as:

1. The first key starts in the first character position of the record, is 10 characters long, and is sorted in ascending order with a user-specified collating sequence if that sequence is present.

2. The second key starts in character position 11, is 10 characters long, and is sorted in descending order with an overpunched sign in the last character position.

3. The third key starts in character position 21, has a length of 10 characters, and is sorted in ascending order.

4. The fourth key starts in character position 31, has a length of 10 characters, and is sorted in descending order.

To merge three files containing records of 80 characters with a key starts in character position 1, has a length of five characters, and is sorted in ascending order, the information string *infost* may be:

'RSZ=80, INPUT=3, KEY=1/5!'

Infost must be the first parameter in the call to FMERGE. The other parameters follow *infost*.

inpts is two or more logical unit numbers, the names of two or more input subroutines, or a combination of logical unit numbers and input subroutines. The parameter *inpts* can contain from 2 through 24 input sources. When *inpts* contains the names of input subroutines, the subroutine names must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.1.

outpt is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.4.

comprt is the name of a comparison subroutine that you supply. The subroutine is called when two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if you don't provide a comparison subroutine. This parameter must be present if you use the COMP clause in *infost*. See L.8.2.

L.5.2. Examples of CALL Statements to Merge

The following runstream contains a call to FMERGE with a simple information string *infost* that contains a KEY clause and an RSZ clause. The RSZ clause declares a record size of 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of 5 characters, and is sorted in ascending order. *Infost* contains the clause INPUT=2 to indicate that there are two input sources contained in the input parameter *inpts*. The *inpts* parameters are the logical unit number 8 and the user-supplied input subroutine name IN. The output parameter *outpt* is the logical unit number 9. *Infost* ends with an exclamation point (!).

```
@RUN
@FTN,SI  MAIN

      EXTERNAL IN
      CALL FMERGE('rsz=80,key=1/5,input=2!',8,IN,9)
      END

@FTN,SI  IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(5,1,END=2) RECORD
      LENGTH=80
      IEOF=0
      RETURN
1     FORMAT(20A4)
2     IEOF=1
      RETURN
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,A  IN
@ASG,C  OUT
@USE  8,IN
@USE  9,OUT
@XQT
. . . the second input file on data images . . .
@FIN
```

The example that follows merges two input image files (IN*1 and IN*2) that were sorted in ascending order on columns 1 through 10 and the merged data is written to file OUT*PUT. The input subroutine ignores all records in the input file IN*2 that have a 1 in column 11.

```
@RUN
@FTN,SI

      EXTERNAL IN
      CALL FMERGE('rsz=80,key=1/10,input=2!',8,IN,10)
      END

@FTN,SI    IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER RECORD*80,I,ONE/ '1'/
1     FORMAT(A)
2     FORMAT(10X,A1)
3     READ(9,1,END=4) RECORD
      DECODE(2,RECORD) I
      IF (I.EQ.ONE) GO TO 3
      LENGTH=80
      IEOF=0
      RETURN
4     IEOF=1
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,A  IN*1
@ASG,A  IN*2
@ASG,C  OUT*PUT
@USE  8,IN*1
@USE  9,IN*2
@USE  10,OUT*PUT
@XQT
@FIN
```

L.6. The CALL Statement to FSCOPY

L.6.1. The CALL Statement to Copy an External Sort Parameter Table

This facility provides access to the Assembler procedure R$FILE.

The form of the CALL statement to copy an external Assembler sort parameter table is:

CALL FSCOPY(*table*)

where *table* is the name of an externalized entry point. *Table* must be declared in an EXTERNAL statement. The CALL statement to FSCOPY with one external argument must occur before a call to FSORT or FMERGE with the COPY clause in its information string *infost*. (See L.4.1 and L.5.1.) The call of FSCOPY establishes which sort parameter table is copied. The subroutine argument is the first word of the sort parameter table to be copied. The subroutine argument is not an ASCII FORTRAN subroutine but is an Assembler entry point.

Only one sort parameter table can be copied at one time. Each call on FSCOPY deletes the previous table that was copied. If FSCOPY is called without any arguments, a new table is not copied and the previous table is deleted.

L.6.2. Record Size When FSCOPY Is Used

Key positions, record lengths, and link sizes in Assembler sort parameters must be given as if there were an extra word in front of the record. (See L.4.1 and L.5.1.)

L.6.3. An Example of CALL Statement to FSCOPY

The following runstream contains an Assembler sort parameter table and program that calls FSORT and FSCOPY. The program sorts the source input from character positions 1 through 6 in ascending order, from character positions 7 through 12 in descending order, and from character positions 13 through 16 in ascending Fieldata order with a special collating sequence such that all Bs precede all As. The information for character positions 13 through 16 comes from the Assembler sort parameter table. The source input is on logical unit 5 and the output is placed on logical unit number 6. Note the extra word or six characters in the starting character position (19+6). This is described in L.6.2.

```
@RUN
@MASM,SI    COPIED
            R$FILE  'KEY',19+6,6,'A','A'  ; Extra word!
COPIED*             'SEQ','@','UPTO',' ', ;
                    'B','A','ALL'
            END

@FTN,SI

      INTEGER CORE(21000)
      EXTERNAL COPIED
      CALL FSCOPY(COPIED)
      CALL FSORT('key=(1/6,7/6/d),copy,rsz=80,core=20000!',
     1    5,6,CORE)
      END

@MAP,IFS
LIB  ASCII*FTNLIB.
@XQT
. . . data images . . .
@FIN
```

L.7. The CALL Statement to FSSEQ

L.7.1. The CALL Statement to Provide a User-Specified Collating Sequence

A user-specified collating sequence can't be explicitly declared in the information string *infost* of a call to FSORT or FMERGE. The use of a nonstandard collating sequence is specified only in the KEY clause field *type* in a character key with the value S. (See L.4.1 and L.5.1.) The user-specified collating sequence is set up through a call to FSSEQ with a single argument.

The form of a CALL statement to FSSEQ is:

CALL FSSEQ (*seqtbl*)

where *seqtbl* is an argument containing a character string that is 256 characters long. *Seqtbl* contains the user-defined collating sequence of the ISO character set. If *seqtbl* is not present, the previous user-defined collating sequence is deleted. Only one user-defined ISO collating sequence is in use at any one time. A second CALL statement to FSSEQ causes the previous collating sequence to be replaced with the new user-defined collating sequence.

L.7.2. An Example of the CALL Statement to FSSEQ

The following runstream contains two calls to FSORT and two calls to FSSEQ. The first call to FSSEQ contains a user-defined collating sequence in array SEQTAB. The collating sequence is the same as the standard ISO collating sequence except that the letters A and B (uppercase and lowercase) are interchanged. The first call to FSORT uses the user-defined collating sequence. The input is read from unit 5. The input is sorted according to:

1. The first key that starts in character position 1 of the record, has a length of 10 characters, and that is sorted in ascending order.

2. The second key that starts in character position 11 of the record, has a length of five characters, and is sorted in descending order according to the user-defined collating sequence specified in the call to FSSEQ.

3. The third key that starts in character position 16 of the record, has a length of five characters, and is sorted in ascending order.

The result is written by the user output routine OUT.

The second call to FSSEQ deletes the previous user-defined collating sequence and does not set up another sequence. This means that the normal ISO collating sequence is used when sorting. Note the use of the CORE= clause in *infost* in both calls to FSORT.

```
@RUN
@FTN,SI

      EXTERNAL OUT
      INTEGER SEQTAB(64),CHAR
*     set up collating sequence
          CHAR(I)=BITS(SEQTAB(1+I/4),1+9*MOD(I,4),9)
      DO 1 I=0,255
1         CHAR(I)=I
      CHAR(65)=CHAR(65)+1
      CHAR(66)=CHAR(66)-1
      CHAR(97)=CHAR(97)+1
      CHAR(98)=CHAR(98)-1
*     give collating sequence to sort
      CALL FSSEQ(SEQTAB)
*     do the first sort
      CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
     1    core=20000!',5,OUT)
*     remove collating sequence
      CALL FSSEQ
*     do the second sort
      CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
     1    core=20000!',10,6)
      END

@FTN,SI    OUT

      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*80 RECORD
      IF (LENGTH.LT.0) RETURN
      WRITE(10,1) RECORD
      PRINT 2,RECORD
      RETURN
1     FORMAT(A)
2     FORMAT(1X,A)
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,T  TEMP
@USE  10,TEMP
@XQT
. . . data images . . .
@FIN
```

L.8. User-Specified Subroutines

The sort/merge interface lets you provide subroutines to do the following:

- Read records
- Compare records
- Examine fixed-length records with equal keys and optionally merge the records
- Write records

You are not required to supply any of these subroutines. The sort/merge interface and package handles all these areas when you don't wish to supply any subroutines.

L.8.1. User-Specified Input Subroutine

An input subroutine can be supplied to be called by the sort/merge package to read the records. (See L.4.1 and L.5.1.) The input subroutine is called with three arguments. The first argument is an array that contains the input record to be returned to the sort/merge package. The second argument is an integer that contains the length of the input record in characters. The third argument is an integer that indicates when the last record is delivered.

The input subroutine can do the following:

1. Read a record and return the record to the sort.
2. Return the null string with a record length of zero and the third argument set to a one to indicate the end of the input file.
3. Read a record and call FSGIVE with that record as an argument. The input subroutine can enter FSGIVE several times before returning an input record or an end-of-file mark to the sort/merge package.

The end of the file can be signaled through FSGIVE. Control is not returned to the instruction following the call to FSGIVE in the input subroutine. Control returns to the sort/merge package.

L.8.1.1. An Example of a User-Specified Input Subroutine

The following input subroutine reads the source input from unit 5 and returns a record length of 80 characters. The third argument is set to 0 if the end of the file is not reached and set to 1 if the end of the file is reached in the input file.

```
      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(5,1,END=2) RECORD
      LENGTH=80
      IEOF=0
      RETURN
1     FORMAT(20A4)
2     IEOF=1
      RETURN
      END
```

L.8.1.2. The CALL Statement to FSGIVE

The call to FSGIVE provides the capability of giving a record to the sort without leaving the user-specified input subroutine. Call FSGIVE with three arguments:

1. the input record for sort

2. the length of the record given to the sort

3. the flag given to sort to indicate that the end of the file is reached

When the flag is 0, the end of the file was not reached, while a nonzero flag indicates that the end of the file is found.

These arguments are similar to the arguments for the input subroutine.

L.8.1.3. An Example of User-Specified Input Subroutine with FSGIVE

The following input subroutine reads characters separated into words by blanks or the end of the line from input unit 5. Any character except a space can be part of a word.

The input subroutine reads from unit 5 when first entered. The subroutine moves each word that it finds to the record area and then calls FSGIVE for each word that is not the last word on a data image. The input subroutine IN is reentered each time a new data image is needed from the input file. The end of the input file is signaled by the input subroutine IN. The end of the input file can also be indicated by setting the third argument to FSGIVE to a nonzero value. The calls to FSGIVE is intermixed with calls to the input subroutine IN.

```
      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER RECORD*80,CARD(80),BLANK/' '/
1     READ(5,2,END=99) CARD
2     FORMAT(80A1)
*     find last nonblank character
      DO 3 IMAX=80,1,-1
3        IF (CARD(IMAX).NE.BLANK) GO TO 4
*     The input record was blank, so read a new record
      GO TO 1
4     I=1
*     find first blank separator
5     DO 6 J=I,IMAX
6        IF (CARD(J).EQ.BLANK) GO TO 8
*     record has no more blanks - deliver
      ENCODE(80,2,RECORD) (CARD(J),J=I,IMAX)
7     LENGTH=80
      IEOF=0
      RETURN
*     At least one blank was found
8     IF (J.NE.I) GO TO 9
*     It was a leading blank - ignore it
      I=I+1
      GO TO 5
*     A word was found - deliver
9     ENCODE(80,2,RECORD) (CARD(K),K=I,J-1)
      CALL FSGIVE(RECORD,80,0)
```

```
      GO TO 5
*     This is end of input - tell the sort
99    IEOF=1
      RETURN
      END
```

L.8.2. A User Comparison Routine

If a comparison routine is present, it is called whenever the sort/merge package must compare two records. The COMP clause must be present in the information string of the call to FSORT or FMERGE. The parameter *comprt* must also be specified in the call to FSORT or FMERGE. (See L.4.1 and L.5.1.)

The compare subroutine is called with three arguments. The first two arguments are the two records compared when the records are fixed-length records or the first links of the two records to be compared when the records are variable length. The third argument is an integer whose value informs the sort of the result of the comparison done by the comparison subroutine. The result can be:

- The value 1 if the first record precedes the second record
- The value 2 if the order of the records is immaterial
- The value 3 if the second record precedes the first record

Care is necessary when using a comparison subroutine together with keys specified in the information string because the sort/merge package translates the key fields according to certain rules. The Sort/Merge Reference, UP-7621 (applicable version), contains a description of the translation rules.

L.8.2.1. An Example of a User Comparison Subroutine

For the following example, assume the first five characters of each record contains a signed, nonzero number between -49999 and 49999. A negative number X is represented by 50000-X. If key translation is not used, the following comparison subroutine can be used:

```
      SUBROUTINE COMP(FIRST,SECOND,CODE)
      INTEGER FIRST(2),SECOND(2),CODE,F,S
      DECODE(1,FIRST) F
      DECODE(1,SECOND) S
1     FORMAT(I5)
      IF (F.GE.50000) F=50000-F
      IF (S.GE.50000) S=50000-S
      IF (F-S) 2,3,4
2     CODE=1
      RETURN
3     CODE=2
      RETURN
4     CODE=3
      RETURN
      END
```

L.8.2.2. An Example of a Runstream With a Comparison Subroutine

In the following example of a comparison subroutine, the first two characters of the records given to the comparison subroutine contain an integer that indicates the starting position of the key within the record. The key is five characters long and contains a right-justified integer value. A complete runstream for using this comparison subroutine is:

```
@RUN
@FTN,SI

      EXTERNAL COMP
      CALL FSORT('comp,rsz=80,core=20000!',9,10,COMP)
      END

@FTN,SI    COMP

      SUBROUTINE COMP(R1,R2,CODE)
      INTEGER CODE
      CHARACTER R1*80,R2*80,F1*8,F2*8
*     compute the key values
      DECODE(4,R1) I1
      DECODE(4,R2) I2
      ENCODE(8,5,F1) I1
      ENCODE(8,5,F2) I2
      DECODE(F1,R1) I1
      DECODE(F2,R2) I2
*     do the comparisons
      IF (I1-I2) 1,2,3
1     CODE=1
      RETURN
2     CODE=2
      RETURN
3     CODE=3
      RETURN
4     FORMAT(I2)
5     FORMAT('(',I2,'X,I5)')
      END

@MAP,SIF
LIB        ASCII*FTNLIB.
@ASG,A     IN*PUT
@ASG,C     OUT*PUT
@USE       9,IN*PUT
@USE       10,OUT*PUT
@XQT
@FIN
```

L.8.3. User Data Reduction Subroutine

The data reduction subroutine is called by the sort/merge package whenever the sort/merge package finds two records whose order is immaterial. The data reduction subroutine may or may not merge the two records into the first record. The data reduction subroutine can only be specified when sorting fixed-length records. The DATA clause must be specified in the information string in the call to FSORT. The *datred* parameter must be present in the call to FSORT. Two restrictions must be remembered:

1. The records, if merged, must always be merged into the first record (that is, the first argument).

2. The data reduction routine can't change key fields.

The data reduction subroutine is called with three arguments. The first two arguments are the two records. The third argument is an integer result assigned by the data reduction subroutine with the following possible values:

- The value 1 indicates that the two records are merged.

- The value 2 indicates that the two records are not merged.

The sort/merge subroutines translate the key fields according to certain rules. Exercise care when using a data reduction subroutine together with keys specified in the information string in the call to FSORT. The translation rules are described in the Sort/Merge Reference, UP-7621 (applicable version).

L.8.3.1. A Simple Example of a Data Reduction Subroutine

The following data reduction subroutine assumes that any input record that contains a 1 in character position 6 is chosen over any other record. If both records contain a 1 in character position 6, the first record is chosen over the second record.

```
      SUBROUTINE DATA(FIRST,SECOND,CODE)
      INTEGER CODE
      CHARACTER FIRST*80,SECOND*80,TEST,ONE/1H1/
      DECODE(1,FIRST) TEST
1     FORMAT(5X,A1)
      IF (TEST.NE.ONE) GO TO 2
      CODE=1
      RETURN
2     DECODE(1,SECOND) TEST
      IF (TEST.EQ.ONE) GO TO 3
      CODE=2
      RETURN
3     FIRST=SECOND
      CODE=1
      END
```

L.8.3.2. An Example of a Runstream With a Data Reduction Subroutine

This example with a data reduction subroutine assumes that two records with equal keys are merged if character position 11 of at least one of the records is blank. The record with the blank in character position 11 is retained. If both records have character position 11 blank, the first record is retained.

This runstream chooses the decision field outside the key fields to avoid any problems with key field translation.

```
@RUN
@FTN,SI

      EXTERNAL DATA
      CALL FSORT('key=(1/5,6/5/d),rsz=80,data reduction
     1    user code,core=20000!',9,10,DATA)
      END

@FTN,SI    DATA

      SUBROUTINE DATA(R1,R2,CODE)
      INTEGER CODE,BLANK/1H    /
      CHARACTER*80  R1,R2
      DECODE(1,R1) IB
1     FORMAT(10X,A1)
      IF (IB.NE.BLANK) GO TO 2
      CODE=1
      RETURN
2     DECODE(1,R2) IB
      IF (IB.EQ.BLANK) GO TO 3
      CODE=2
      RETURN
3     R1=R2
      CODE=1
      END

@MAP,SIF
LIB         ASCII*FTNLIB.
@ASG,A      IN*PUT
@ASG,C      OUT*PUT
@USE        9,IN*PUT
@USE        10,OUT*PUT
@XQT
@FIN
```

L.8.4. User-Specified Output Subroutine

The output subroutine is called by the sort/merge package when a record is written. The output subroutine is called with two arguments. The first argument is the record to be written and the second argument is the length in characters of the record to be written. The second argument is also a flag to your output subroutine to indicate when the sort delivers the last record to be written. The length is normally a positive number indicating the size of the record in characters. If the length is negative or zero, the last record is delivered to the output subroutine.

L.8.4.1. A Simple Example of a User-Specified Output Subroutine

The following output subroutine outputs records to unit 6 through a formatted write:

```
      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*80 RECORD
      IF (LENGTH.GT.0) PRINT 1,RECORD
1     FORMAT(1X,A)
      RETURN
      END
```

L.8.4.2. The CALL Statement to FSTAKE

Your output routine can indicate to the sort/merge package when the output routine needs a new output record. This is done by a CALL statement to FSTAKE with two arguments. The first argument is the record received from the sort/merge package. The second argument is the length in characters of the new record. If the length argument is negative after returning from FSTAKE, the last record is delivered from the sort/merge package.

L.8.4.3. An Example of FSTAKE in an Output Subroutine

The following example moves records containing one word each into card images with exactly one space between the words, then writes the record when the card image becomes full. The sorted records are assumed to contain 80 characters.

```
      SUBROUTINE OUT(RECORD,LENGTH)
      INTEGER POS/1/
      CHARACTER CARD(80),CR(80),BLANK/ '  '/,RECORD*80
      IF (LENGTH.LT.0) GO TO 99
*     Blank the output record
      DO 1 I=1,80
1         CARD(I)=BLANK
*     Place each character of the record in a word
2     DECODE(80,3,RECORD) CR
3     FORMAT(80A1)
*     Find actual length of record
      DO 4 IL=80,1,-1
4         IF (CR(IL).NE.BLANK) GO TO 5
*     This is a blank record---ignore the record
      GO TO 10
5     IF (POS+IL.LE.81) GO TO 8
*     The card image to print is full--go print it
      PRINT 6,CARD
```

```
6       FORMAT(1X,80A1)
        DO 7 I=1,80
7           CARD(I)=BLANK
        POS=1
8       DO 9 I=1,IL
9           CARD(POS-1+I)=CR(I)
        POS=POS+IL+1
*       get next record to get next word
10      CALL FSTAKE(RECORD,LENGTH)
        IF (LENGTH.GE.0) GO TO 2
99      IF (POS.GT.1) PRINT 6,CARD
        RETURN
        END
```

L.9. Optimizing Sorts

An understanding of this subsection is not required to do a sort. This information is provided for those who need to sort larger data sets than the standard scratch assignments (main storage and mass storage) allow. This information also helps those who need to minimize the resources used in a sort. You also need to use this information if the sort/merge package error B5 is given for a sort.

The standard scratch file assignments are:

- 19,000 words of main storage
- Six disk files of 512 tracks each (initial reserve 0)

This amount of storage should be sufficient to sort some 200,000 to 250,000 card images. If a very large sort (that is, a multiple cycle sort requiring operator intervention) is necessary, you should consult the Sort/Merge Reference, UP-7621 (applicable version). The performance of a sort is mainly determined by the following three factors:

1. The bias of the input data
2. The size of the main storage scratch area
3. The scratch files used

The CPU time used by the sort is decreased slightly by inhibiting the checksum on the sort's scratch files or by increasing the size of the checksum mesh.

L.9.1. The Bias of the Input Data

The bias can be defined as the average number of records in sorted subsequences present in the input file. (See BIAS=*biasno* in L.4.1.) For example, if the input file is exactly in reverse order, the bias is 1. Also, random data has a bias of 2. Generally, the bias is greater if the input file is almost sorted; that is, the more nearly sorted the input file, the greater the bias.

If a bias is specified, the sort is able to use available resources optimally so that more data can be sorted using the same amount of scratch storage. You should specify the bias whenever it is known and when the bias is less than 1.4 or greater than 3.

The bias is specified by the form:

BIAS=*biasno*

L.9.2. The Size of the Main Storage Scratch Area

The size of the main storage scratch area is specified two ways:

1. Assign the file R$CORE with a suitable maximum granule value.
2. Specify the size of the main storage scratch area in the information string for the sort or merge.

If the size of the main storage scratch area is given by both methods, the R$CORE value overrides the size of the main storage scratch area given in the information string. (See L.4.1.)

L.9.2.1. The Use of R$CORE

The size of the main storage scratch area in words is specified at run time by assigning the file R$CORE with a suitable maximum size. For example, if 20,000 words of main storage scratch area are desired, the following control statement guarantees that 20,000 words of storage are available to the sort/merge interface and package:

```
@ASG,T  R$CORE,///20
```

L.9.2.2. The Use of the CORE Clause in the Information String

The amount of main storage scratch area for the sort is specified by the following CORE clause in the information of the call to FSORT:

CORE=*corsz*

where *corsz* is the size of the main storage scratch area in words. The sort/merge interface rejects any size that is less than 3,000 words. The sort generally executes faster when it is given more main storage.

When you use this clause in the information string, don't assign the file R$CORE.

L.9.3. The Scratch Files Used and Checksum

Avoid the use of tape scratch files when possible. Tape sorts are slower and require operator intervention. The sort/merge package distinguishes two cases for mass storage files:

1. One or two mass storage files

2. More than two mass storage files

The first case is more suitable when only one or two mass storage units are available to the sort. However, this case requires a careful assignment of main storage and mass storage scratch resources. The optimal amount of main storage will depend on how much mass storage is available to the sort. A suitable assignment of facilities appears in the Sort/Merge Reference, UP-7621 (applicable version). Different mass storage scratch files should be kept on separate mass storage units if possible.

L.9.3.1. Scratch Files Named in the Information String

Scratch files are specified by the FILE clause in the information string. (See L.4.1.) The following restrictions apply when the FILE clause is used:

1. All scratch files must be assigned when the sort is started.

2. A maximum of 26 scratch files can be specified.

3. At least three tape scratch files must be used if any tape scratch files are used.

4. If tape scratch files are used, a maximum of two mass storage files are used for the sort.

L.9.3.2. Checksum and the Sort

A checksum is normally done on all tape and mass storage files. You can omit the checksum on one or more device types (mass storage or tape). You can also specify a checksum mesh size for each device type. For example, if a mesh size of 5 is given, only every fifth word of each block written to tape or mass storage is included in the checksum.

You can omit the checksum by specifying the nocheck clause (NOCH) in the information string for the call to FSORT. You provide a mesh size by specifying the MESH clause in the information string. These clauses are described in the CALL statement to FSORT. (See L.4.1.)

L.10. Sorting Very Large Amounts of Data

When it is not practical to assign enough scratch storage to hold all of the data to be sorted, a multicycle sort must be done. For that case, the ASCII FORTRAN program must be executed with the P option (@XQT,P) and some sort/merge package parameter data images must be prepared. These data images are fully described in the Sort/Merge Reference, UP-7621 (applicable version).

The SMRG parameter data image format is:

'SMRG','*outptprefx*', *nbrrecds*, *nbrreel*

where:

outptprefx is a string two characters long that identifies the intermediate output tapes.

nbrrecds specifies the number of records sorted in each cycle. It is optional.

nbrreel specifies the number of reels to be produced in each cycle and is optional. If the number given in *nbrrecds* specifies more records than the assigned hardware can hold, *nbrrecds* is ignored.

The parameter data images are read after the call to FSORT but before the first input record is read or before your input routine is first called.

You must use the following control image after the last sort/merge parameter data image:

```
@EOF A
```

If you wish to rerun interrupted multicycle sorts, refer to the Sort/Merge Reference, UP-7621 (applicable version).

L.10.1. An Example of a Large Single-Cycle Sort

The following runstream contains a sort that must run as efficiently as possible. The records are in nearly reverse order (the bias is about 1.2). A checksum is not done. About 400,000 records must be sorted, so the standard scratch assignments cannot be used. Ample amounts of main storage and mass storage are available for the sort.

The first step is to calculate the sort volume. This is the record size times the number of records times a safety factor:

```
20 * 400000 * (1 + .1)
```

which equals about 9 million words.

For a big sort, use six equal-size files. This makes each file about 1.5 million words, or about 850 tracks.

A suitable amount of main storage scratch area is about 50K words.

The scratch files must be assigned before the sort begins. The runstream for the sort can be:

```
@RUN
@FTN,SIO

      CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
     1    bias=1.2,files=(M1,M2),nocheck=dft,
     1    files=(m3,m4,m5,m6)!',9,10)
      END

@MAP,SIF
LIB        ASCII*FTNLIB.
@ASG,A     IN*PUT
@ASG,T     OUT*PUT,T,REELNO
@ASG,T     M1,///850
@ASG,T     M2,///850
@ASG,T     M3,///850
@ASG,T     M4,///850
@ASG,T     M5,///850
@ASG,T     M6,///850
@USE       9,IN*PUT
@USE       10,OUT*PUT
@XQT
@FIN
```

L.10.2. An Example of a Multiple-Cycle Sort

The following runstream is used for a multicycle sort. The information is much the same as the large single-cycle sort except that there are about 20 million records sorted, using the same amount of main storage and mass storage. In addition, four scratch tape files are used.

```
@RUN
@FTN,SIO

      CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
     1    bias=1.2,file=(M1,M2,T1,T2,T3,T4),noch=dft,
     1    files=(m3,m4,m5,m6)!',9,10)
      END

@MAP,SIF
LIB        ASCII*FTNLIB.
@ASG,A     IN*PUT
@ASG,TV    OUT*PUT,U9V/2,REEL1/REEL2/REEL3
@ASG,T     M1,///1117
@ASG,T     M2,//POS/715
@ASG,T     T1,T
@ASG,T     T2,T
@ASG,T     T3,T
@ASG,T     T4,T
@USE       9,IN*PUT
@USE       10,OUT*PUT
@XQT,P
'SMRG','EX'
@EOF A
@FIN
```

L.11. Error Messages From a Sort or a Merge

Two different types of error messages can be produced during a sort or a merge. The first type is written to the console and its form is:

XXXX ERROR CODE *Y Z*

where *XXXX* is SORT or MERGE, *Y* is a letter, and *Z* is a digit. This message is immediately followed by an ER ERR$ exit. This type of message is produced by the sort/merge subroutines and is described in the Sort/Merge Reference, UP-7621 (applicable version).

The second type of message is produced by the sort/merge interface with the form:

FTN SORT/MERGE ERROR CODE *NN strg*

where *NN* is a 2-digit error code. The error codes and an explanation for each follows. *Strg* is a four-character string that provides further information on the error.

01 The mnemonic in the information string whose first four characters are given in *strg* is not known to the routine called (for example, SELE is not allowed for merges and UNKNOWN is not allowed for sorts or for merges).

02 The routine specified in *strg* is called with the wrong number of arguments.

03 The character position of the most significant character of a key is negative or too large.

04 A key length is negative or too large.

05 An erroneous key type (such as A for a character key) is specified.

06 The sorting sequence is not A, D, or a null string.

07 The word position of the most significant bit of a bit key is negative or too large.

08 The bit position of the most significant bit of a bit key is incorrect (0 or greater than 36).

09 The translation table in FSSEQ does not contain the full ISO set. *Strg* contains the octal code for the first character found that cannot be translated.

10 The maximum record size (RSZ) given in the information string is negative or greater than 65K.

11 No record size (RSZ) is specified in the information string for a sort.

12 An impossible link size (0 or greater than the maximum record size) is specified.

13 No keys and no user comparison routine are given in the information string.

14 An error exists in the collating sequence (FSSEQ). The given string is less than 256 characters.

15 The auxiliary main storage area is full. For remedial action, please submit a software user report (SUR).

16 An overflow occurs in the sort parameter table. For remedial action, please submit a software user report (SUR).

17 At least one key extends beyond the record.

18 A bit key of type A, D, G, L, or P does not start on a 6-bit byte boundary.

19 The length in bits of a bit key of type A, D, G, L, or P is not divisible by 6.

20 An erroneous return code is given on exit from your comparison routine.

21 An erroneous record length is given on exit from your input routine.

22 The link size is not specified for variable length records and no key specifications are given.

23 A forbidden character was found in the information string.

24 The output routine/file or your comparison routine is not in the argument list.

25 The COPY specification is given in the information string, but FSCOPY is not called (or the most recent call has no arguments).

26 For a sort, an input file/routine is not in the argument list. For a merge, either too few (less than two) or too many (more than 26) input files/routines are given in the argument list.

27 The bias is given as less than 1.

28 The mnemonic whose first four characters are in *strg* appears more than once in the information string. If *strg* is RSZ, VRSZ may have appeared before (and vice versa).

29 The size of the main storage scratch area is given as less than 3,000 words or greater than 262,141 words.

30 An illegal character is given in the NOCH specification (only D, F, K, and T are accepted to the right of the equals sign) in the information string.

31 Your data reduction routine is not in the argument list.

32 A given scratch file is not on mass storage. The most common reason is that the file is not assigned to the run.

34 More than 26 scratch files are specified in the information string.

35 The first member of a select pair (SELE clause) is not greater than the previous pair's second member.

36 The second member of a select pair is less than the first member.

37 An erroneous return code is given on exit from your data reduction routine.

38 A facility reject status is generated in the attempt to assign one of the sort scratch files. *Strg* specifies which of the six standard files can't be assigned. The next line gives the FAC REJECT code.

39 Some keys overlap. *Strg* gives the number of the major key of the pair that overlaps (the most major key is number 1, the next number 2, etc.).

40 An illegal sign appears in an arithmetic field. *Strg* gives the first four characters found after (and including) the one in error.

41 An illegal character appears in a numeric field. *Strg* gives the first four significant digits.

42 The field in *strg* is not followed by an equals sign.

43 A numeric field given in *strg* appears when an alphabetic field is expected.

44 Nonblank characters appear between an equals sign and a left parenthesis.

45 The first field of a *keyspn* in a KEY clause is alphabetic and is not BIT.

46 An alphabetic field in *strg* appears when a numeric field is expected.

47 An integer field contains a decimal point.

48 An invalid delimiter in *strg* is found.

49 The name of a scratch file contains more than 12 characters.

50 The first instruction of your routine is illegal.

51 Too many parameters in the call to FSORT or FMERGE exist.

52 No main storage scratch parameter argument is given and an OWN clause is in the information string.

53 The data reduction routine is specified for a sort of variable-length records.

55 The first word of the user-provided sort parameter table is wrong.

63 The mesh size is previously specified for a device given in *strg* (*strg* has a value D, F, K, or T).

64 An impossible mesh size is specified.

65 An illegal device type in *strg* is used in a MESH specification.

66 An illegal delimiter is used in a MESH specification.

67 Banked data arguments are not allowed.

68 Only the first argument to FSORT, FMERGE, FSGIVE, or FSTAKE can be of type CHARACTER.

69 The logical unit number for input or output is a reread unit or outside the defined range of logical units.

Appendix M. Virtual FORTRAN

M.1. General

When a collected FORTRAN program doesn't fit in the traditional 65,535 words of main storage (or 262,143 words if the FORTRAN programs were compiled with the O option) and collector truncation diagnostics result, consider putting large objects, such as common blocks or local arrays, in virtual space.

Putting a large object in virtual space reduces the main storage requirements of the collected absolute element. For example, a subroutine that processes two REAL argument arrays of extent NxN needs six local arrays of the same size (or larger) as working storage during its processing. Since a local array can't be dimensioned as NxN, a reasonable maximum size must be picked and the subroutine must be compiled with that size. But, the following problems exist:

- When NxN is 150x150, this takes up 22K words of main storage per local array or 132K words total in local storage for this one routine.
- When a typical N in an execution is only 50, most of the storage space just mentioned goes to waste.
- When using an N over 150, the element must be recompiled with a larger size to handle it.
- When using an N over 180, the main storage requirements are too large to fit in the 262,000-word address range limit of the Series 1100 architecture.

But when these six arrays are put in virtual space, the following occurs:

- The collected size drops by about 132K words.
- The local arrays are dynamically allocated in virtual space on subprogram entry, and freed on subprogram exit.
- During execution, main storage requirements are only marginally larger than the collected size.
- Virtual space is exempt from the 262,000-word limit of the current Series 1100 architecture, and you can pick a value for N that is rarely exceeded (thereby forcing recompilation) with minimal extra overhead.

The following example shows the same subroutine written with and without virtual space:

Program before change:

```
SUBROUTINE SUB(N,AR1,AR2)
PARAMETER (M=150)                @ marginal size
REAL L1(M,M),L2(M,M),L3(M,M)
REAL L4(M,M),L5(M,M),L6(M,M)
REAL AR1(N,N),AR2(N,N)
```

Program after change:

```
SUBROUTINE SUB(N,AR1,AR2)
VIRTUAL L1,L2,L3,L4,L5,L6
PARAMETER (M=300)                @ comfortable size
REAL L1(M,M),L2(M,M),L3(M,M)
REAL L4(M,M),L5(M,M),L6(M,M)
REAL AR1(N,N),AR2(N,N)
```

Named common can also be put into virtual space as shown:

Program before change:

```
PARAMETER (M=100)                @ marginal size
COMMON/C1/A(M,M)B(M,M)
COMMON/C2/D(6000),F(99000),G(M)
COMMON/C3/PIVOT(M)
```

Program after change:

```
VIRTUAL /C1/,/C2/
PARAMETER (M=400)                @ comfortable size
COMMON/C1/A(M,M)B(M,M)
COMMON/C2/D(6000),F(99000),G(M)
COMMON/C3/PIVOT(M)
```

A maximum virtual address range of 32 million words is currently available using virtual space. The default size is 6.5 million words. A big difference exists between theoretical limits and practical limits. We recommend that a casual user keep within two or three times the real memory size of his machine to keep thrashing levels acceptable.

Compiler-generated code that references a virtual object is generally less efficient than that for references to objects in nonvirtual space. (An object refers to a scalar variable or an array in common, or local to, the subprogram.) Only the larger common blocks and local arrays that cause size problems should be put into virtual space. For example, when arrays dimensioned as NxM are put in virtual space, smaller single-dimension arrays dimensioned by N or M to hold a row or column should be left in normal nonvirtual space.

Several routines are available to you that enhance CPU performance of virtual or banked programs. For more information, see M.17.

M.2. Method

Use of multiple D-banks materializes virtual space. Each D-bank contains a page of virtual space. A virtual object from your program is dynamically allocated by the ASCII FORTRAN run-time system and gets as many pages as are necessary to hold the object. Since the Series 1100 executive currently allows a maximum of 251 banks to an executing program, this puts an upper limit on the number of D-banks used for virtual pages. The amount of virtual space available is the maximum number of banks times the virtual page size. You can select the maximum number of pages allocated for virtual space and their size. Defaults are 200 pages and 32K words, giving a default of 6.5 million words of virtual space. (K means 1,024 words.) Page sizes are a power of 2 and can be 4K, 8K, 16K, 32K (default), 64K, or 128K words. A total of 246 banks of size 128K words gives a range of 32 million words. Large page sizes give a large virtual address space, but smaller page sizes minimize main storage requirements and dramatically reduce potential thrashing problems of a large page size.

The D-bank pages used for virtual space are defined in the FORTRAN main program's relocatable by use of a new collector INFO-11 directive. Use of these directives in the generated relocatable defines a set of initially void D-banks to the collector. COMPILER statement options can be used in the main program to change the size or the number of these D-banks from the default values. When the main program does not contain a VIRTUAL statement or is not written in FORTRAN, virtual space can still be used. The run-time library element VSPACE$ then supplies the INFO-11 directives defining virtual space. Default values can be altered by changing tags in the procedure VIRTPROC$ in the MASM procedure element FTNPROC, and then reassembling elements VSPACE$ and VFTNEQU$.

Virtual space is allocated and initialized dynamically during execution of your program. When an external subprogram is entered for the first time, it calls the virtual storage allocator to allocate space for the static virtual objects belonging to it, or to any of its internal subprograms. These static virtual objects can be named common blocks and selected large local arrays. Generated code saves the virtual addresses of these objects returned by the virtual storage allocator. After allocation, execution of code performs any initialization needed resulting from DATA statements on the virtual items. On subsequent entries to the subprogram, the above process is skipped for the allocation and initialization of static virtual space; it is only done once per external program unit. When a named common block has already been allocated by a previous program unit, the virtual allocation routine simply returns its virtual address.

Each D-bank holding a page of virtual space has a 64-word ID area at the beginning of the bank. This area makes the bank self-identifying, and is used by generated code to speed up execution times.

Virtual local arrays in the automatic storage class (those that do not have SAVE statements specifying them) are allocated and initialized on each entry to an external subprogram, and are freed on each return from the subprogram. Local virtual space defaults to the automatic storage class and local nonvirtual space defaults to the static storage class.

Expect the following when virtual space is allocated for a virtual object:

- The object is allocated on a 64-word boundary in a virtual page.
- The first 2,048 words of a virtual object are guaranteed to be in one virtual page. This permits more efficient code to be generated to reference virtual objects when they are in the first 2K of their allocation.

M.3. Restrictions for Declaration Matching

Since special code sequences are required to reference virtual objects, all FORTRAN routines that reference a virtual object must declare the object as a virtual object. However, arguments don't need to be specified in a VIRTUAL statement. A VIRTUAL statement in a FORTRAN element guarantees that arguments are correctly referenced when they reside in normal, banked, or virtual space. For more information on banking, see Appendix H.

Example:

One element:

```
VIRTUAL L1,/C1/
COMMON/C1/C1(999)
COMMON/C2/C2(10000)
REAL L1(9999)
CALL X(L1(I),1.0,I)
END
```

Separate element:

```
SUBROUTINE X(A1,XL,I)
COMMON/C1/C1(999)
VIRTUAL/C1/
COMMON/C1/C1(999)
REAL A1(*)
C1(I)=A1(I)
        .
        .
        .
END
```

Description:

When subprogram X doesn't contain a VIRTUAL statement, or when common block C1 is not named in its VIRTUAL statement, fatal run-time diagnostics result, indicating storage mismatch.

In current FORTRAN library files holding FORTRAN relocatables, often a simple COMPILER(BANKED=ALL) statement is put in each FORTRAN element in case any program using the library has multiple D-banking present. Obtain the same results by inserting a VIRTUAL statement into each element in case any arguments are in virtual space. (This also handles banked arguments.) When any named common blocks are in virtual space, they must be named in VIRTUAL statements in any subprogram that declares them.

M.4. Initialization of Virtual Objects

Initialization for objects in virtual space is needed when they have initial values specified in DATA, DIMENSION, or type statements. This initialization is accomplished by code that executes after the allocation calls. A DATA statement consisting of a mismatch of storage classes presents a problem, as the following example shows:

```
VIRTUAL /DBVIRT/,LOCALD,LOCALS
COMMON /CB1/A(1000)              @ standard common
```

```
COMMON /CBVIRT/V(1000000)          @ virtual common block
DIMENSION LOCALD(1000000)          @ virtual dynamic local
DIMENSION LOCALS(1000000)          @ virtual static local
SAVE LOCALS                        @ put LOCALS in static class
DATA A(1),V(1),LOCALD(1),LOCALS(1)/1.,2.,3,4/
```

The DATA statement in the above example calls for:

- Initialization of nonvirtual common block CB1. (This is normally done by ROR packets, that is, code is normally not executed to initialize items in nonvirtual common.)
- Initialization of virtual common block CBVIRT. (This must be done only once by generated code after allocation.)
- Initialization of the static local array LOCALS. (This must be done only once by generated code after allocation.)
- Initialization of the dynamic local virtual array LOCALD. (This must be done by generated code after allocation, on each entry to the subprogram.)

The ASCII FORTRAN compiler can't separate the initializations of these various types so that they can be done in their respective manners. Therefore, when any item in any initialization statement is in virtual space, the entire statement is accomplished by generated code. When any item in an initialization is in the virtual automatic storage class, all items must be in the automatic storage class or the statement is flagged in error. (The preceding example is in error.) When an initialization statement has a virtual/nonvirtual mismatch on static class items, a warning is issued at compile time, and code generates to accomplish the entire statement at execution time. The warning is issued since dynamically initializing nonvirtual common can lead to incorrect program results. Initialization done by the collector resulting from ROR packets is insensitive to program execution flow, whereas initialization done by generated code exactly follows the program execution flow and can result in an item being initialized after it is used.

To prevent problems in an initialization statement, you should ensure that:

- A mixture of static storage class virtual items and automatic storage class virtual items are not contained in an initialization statement
- A mixture of virtual items and nonvirtual items are not contained in an initialization statement.

The following shows the compiler action on combinations of various storage classes in a single initialization statement:

		VIRTUAL			NONVIRTUAL		
		common	*static local*	*automatic local*	*common*	*static local*	*automatic local*
VIRTUAL	*common*	X					
	static local	OK	X				
	automatic local	E	E	X			
NONVIRTUAL	*common*	W	W	E	X		
	static local	OK	OK	E	OK	X	
	automatic local	E	E	OK	OK	OK	X

NOTE: E means an error diagnostic; W means a warning diagnostic.

M.5. BLOCK DATA

BLOCK DATA subprograms initializing common in virtual space must appear in an element holding one or more executable program units. The first program unit entered in the element also causes the BLOCK DATA initializations to be done dynamically. Due to this, we recommend that you place the BLOCK DATA subprograms in the element holding the main program. Always place BLOCK DATA subprograms in an element holding executable code because the collector ignores the BLOCK DATA element unless the element is part of the collection.

M.6. DATA = AUTO

The COMPILER(DATA=AUTO) statement activates the ASCII FORTRAN automatic storage feature. Programs using this feature can also use the VIRTUAL statement. However, one or more noncommon static D-bank cells are allocated per program unit, and the prolog code is slightly longer.

M.7. Error Detection

M.7.1. Insufficient Space

A request for virtual space that cannot be met results in a run-time diagnostic and error termination. You must either limit your storage requirements, or increase the virtual address space available by putting COMPILER statements in your main program that specify larger banks or more banks for virtual space.

Once the Executive is changed to allow the loading of a program containing over 251 banks, the full 2,047 banks allowed by the collector can be used, and the maximum virtual address range rises to 262 million words.

Most run-time diagnostics are accompanied by a walkback to aid in debugging. The walkback is available only when one or more of the compilations uses the F option.

M.7.2. Bad Allocation or Initialization

Because of the dynamic allocation and initialization of named common in virtual space, the first program unit that calls the virtual storage allocator for a given common block must have the same or larger size for the common block of any other program unit and must be the only program unit with DATA initialization on it. An exception to this is when there are BLOCK DATA program units in the same element that are executed first. A program unit presenting a request to the virtual storage allocator for an allocated common block results in a run-time diagnostic when either:

- The requested size is greater than that originally allocated, and the allocation can't be expanded
- Initialization is to be performed by this program unit

The virtual storage allocator keeps track of nonvirtual common blocks. When one subprogram indicates a common block is a virtual object and another says it is not, a fatal run-time diagnostic occurs.

When a nonvirtual common block has dynamic initialization due to a DATA statement in a given program unit and the common block is already referenced by another program unit, a run-time diagnostic also occurs.

M.7.3. Page Spanning

A major restriction of virtual FORTRAN is:

No portion of a scalar or array element may span from one virtual page to the next.

The compiler is not aware of the page size to use unless a main program is present in the compilation unit. Thus, for many instances it can't diagnose a scalar or array element that illegally spans a page boundary. So, the compiler calls a span-check routine for each referenced variable that has a potential spanning problem. A run-time diagnostic occurs when a problem exists.

Example:

```
VIRTUAL /C1/
COMMON /C1/ R,DP(400000),R2(100000)
COMPILER(PAGESIZE=4K)
READ R,R2
DOUBLE PRECISION DP
        .
        .
        .
```

This program is in error. The double-precision array DP will have spanning problems on page sizes from 4K to 64K, and can also span on a 128K page size when the common block is allocated far enough into the virtual page. Change the program to place the double-precision array DP on a double word boundary. One method is:

```
COMMON /C1/ R,TRASH,DP(400000),R2(100000)
```

Insertion of the variable TRASH puts array DP on a double word boundary. Run-time diagnostics appear when bank spanning is detected.

Arguments also pose a problem. Bank spanning can occur when the size or type declared for the dummy argument isn't the same as that declared for the actual argument passed. Examples include:

- passing a REAL array, but declaring it as DOUBLE PRECISION or COMPLEX in the called routine
- a mismatch on size declarations on actual and dummy character arguments (these can be either scalars or arrays)

To detect a bank-spanning problem on arguments from any of the above causes, a span-checking routine is called on entry to a subprogram to check its arguments. A run-time diagnostic occurs for each argument with a problem.

This span-checking routine is not called for arguments when optimization is used or when a COMPILER(ARGCHK=OFF) statement is in the called routine. It is handled like the normal ASCII FORTRAN argument type-checking routine (for being called or not being called).

Example:

```
      VIRTUAL /C1/,/C2/
      COMMON /C1/C4(9000)
      COMMON /C2/R(9000)
      CHARACTER*4 C4
      REAL R
*
*     Arrays C4 and R can't incorrectly span banks;  they start on acceptable boundaries.
*
      CALL SUBR(C4(1),R(2))
              .
              .
              .
```

```
      SUBROUTINE SUBR(C,DP)
      VIRTUAL
      COMPILER(ARGCHK=OFF)
      CHARACTER*11 C(4000)
      DOUBLE PRECISION DP(3000)
*
*     These actual to dummy argument type-
*     mismatches may cause spanning problems.
*
            .
            .
            .
```

Program execution errors can result because bank spanning can occur on some array elements, depending on virtual page size, where the common blocks are allocated in them, and which array elements are referenced. When dummy argument arrays have an asterisk (*) as the last dimension, the extent of the array is unknown and no span-checking function takes place.

Example:

```
      SUBROUTINE INTX(C,X,Y,N,OK)
      VIRTUAL
      CHARACTER*(*) C(*),COK(9991,N)
      DOUBLE PRECISION X(N,*)
      COMPLEX*16 Y(100,N,*)
```

Description

Only array COK can be span-checked in this example.

M.8. Character Arrays

Code generated to reference a character array uses a compiler-generated variable referred to as a virtual origin variable. The virtual origin variable contains the address of the array manipulated to simplify index calculations. The virtual origin of a character array whose element size is not a multiple of four characters (including all CHARACTER*(*) arrays) is kept in character address form (for example, base address times four). When the array is in virtual space, its base address is a virtual address. When the run-time library element F2BDREQU$ (see M.10) is changed to use BDR3 for referencing virtual space, the D-bit of the BDR field is shifted up to make the virtual origin value overflow and become negative when using 128K banks for virtual space. All other virtual bank sizes don't cause a problem. Therefore, the following restriction is placed on the use of virtual FORTRAN:

A program defining a virtual or dummy character array whose element size is not a multiple of four characters results in a run-time diagnostic when the collected bank size is 128K and BDR3 is used for virtual space.

There is no problem with 128K banks when the default BDR1 references virtual space. The span-check routines issue the diagnostics for this nonmultiple-of-four character array problem for 128K banks.

M.9. Banking and BDR Use

The D-banks used for virtual pages start near the end of the standard 262,143-word address space: 262,144 minus page size minus 512. Don't overlap this address space with the control bank or other multiple banks.

M.10. Hidden User Banks

When on dual-PSR machines you can use the utility PSR that is not used for the virtual banks, as long as its use is hidden from ASCII FORTRAN. You must define the banks in the collector symbolic and manage all basing of banks under that window. ASCII FORTRAN treats items in these user banks as if they are in the control bank. BDIs should never be passed for arguments in user banks, instead, a BDI field of zero should be passed so that ASCII FORTRAN treats the items as unbanked (that is in the control bank). This happens automatically for FORTRAN programs when nonvirtual common blocks are included by the IN directive in the user banks.

When hidden user banks are used, the following rules apply:

- When a named common block is in a hidden bank, it may not be referenced from a FORTRAN subprogram with a COMPILER(BANKED=ALL) statement in it.
- User banks can't overlap in address space with any other program banks (I-banks, control bank, or virtual banks).
- Two BDRs simultaneously basing banks with overlapping address space are not allowed.
- A virtual bank must not be based under a BDR used for user banks.
- A request for storage by an MCORE$ statement on the control bank must not overlap address space with user banks.
- No FORTRAN-generated code (location counters 0, 1, and 4 in an ASCII FORTRAN compilation) should be placed in hidden user banks.

The utility I-bank BDR (BDR1) is used to base the virtual banks. When you are using this BDR for FORTRAN banked space, or for your own use hidden to FORTRAN, the utility D-bank BDR can be used instead for virtual space. The run-time library element F2BDREQU$ holds EQU values for the BDRs, as follows:

VBDR$	BDR used for virtual space
BBDR$	BDR used for banked space
CBDR$	BDR used for the control bank

When your program is not collected according to these conventions, you must edit F2BDREQU$ and reassemble it.

M.11. Performance

Virtual FORTRAN adds the ability to define and use large user objects such as scalar variables and arrays to the Series 1100 ASCII FORTRAN system. Referencing a virtual object requires more code than referencing a nonvirtual object, so only put larger objects in virtual space (an object here means an array or scalar variable). When a large common block has mostly large

arrays, you can place the larger arrays in a new common block that is then placed in virtual space. This leaves the smaller items in nonvirtual space, which is more efficient to reference.

The software paging of virtual FORTRAN is similar to the virtual memory architectures of other manufacturer's machines. Improper use of a large user address space in these machines often means disaster for you. This is also true for the virtual FORTRAN 1100 system. CPU performance and page traffic depend on what you do and don't place in virtual space, as well as on how you use it. It is possible to cause dramatic CPU performance and page-traffic changes by a simple reordering of key loops in some programs. A program that goes through a large amount of virtual space referencing one item per page doesn't execute in a reasonable amount of time.

A program that uses a large amount of virtual space takes some setup time. This is because even static virtual space is dynamically allocated and initialized, and each bank must have its correct size allocated by an ER to MCORE$. The overhead involved with allocation of virtual space is incurred only once per program. For a program with a large amount of virtual space, this overhead is not unreasonable.

M.12. Thrashing

On virtual storage machines, your user address space is broken into pages for purposes of swapping. A reference to something in a nonresident page automatically brings that page in real memory. Resident pages not currently being referenced are swapped out by the operating system when storage becomes scarce.

In the ASCII FORTRAN virtual system, the pages reside in the D-banks defined by the main program's INFO-11 directives. The allowable page sizes are 4K, 8K, 16K, 32K, 64K, and 128K words. You select the page sizes in your main program. Large page sizes can cause thrashing when your reference pattern doesn't result in a reasonably sized working set of pages. Avoid large page sizes unless necessary for a larger virtual address range. Programs that need a very large address space (i.e., 4 million words or more) run when the system is lightly loaded.

When a program takes excessive wall clock time to execute in comparison to the SUP times accumulated, thrashing is occurring. Look for loops in your program that reference virtual objects in an inefficient manner.

Example:

The following program defines and references its data in an inefficient manner:

```
      SUBROUTINE COMP
      VIRTUAL /CB1/,VEC
      COMMON /CB1/A1(2000,2000),B(2000,2000)
      REAL VEC(2000)
      DO 5 K = 1,2000
5           VEC(K) = 0.0
      DO 10 J=1,2000
      DO 10 I=1,2000
10          VEC(I) = A1(J,I) - SIN(B(J,I))
            .
            .
            .
```

Description

The inner loop of this program references three separate virtual pages on each iteration, and two of these pages are referenced for only a very few iterations before a bank change occurs. The local vector VEC isn't that large (2,000 words) and should not be placed in virtual space when used in this manner.

Example:

The following variation on the above program segment is essentially equivalent, and greatly reduces page traffic and thrashing:

```
      SUBROUTINE COMP
      VIRTUAL /CB1/
      COMMON /CB1/A1(2000,2000),B(2000,2000)
      REAL VEC(2000)
      DO 5 K = 1,2000
5           VEC(K) = 0.0
      DO 10 J=1,2000
      DO 10 I=1,2000
10          VEC(I) = A1(J,I) - SIN(B(J,I))
            .
            .
            .
```

Description

In this code sequence, there are many references to a page before it is no longer needed. The working set of pages referenced by the program changes very slowly with time. In addition, page traffic and bank swapping by the operating system are substantially reduced.

The dynamic allocator (DA) in the executive controls storage allocation and swapping; thus, its operation directly influences the thrashing potential of a program. An executing program must have the ability to accumulate a reasonable working set of resident pages or thrashing occurs. Recent DA modifications improve its operation in this area. The changes are called the PEF-1 package. This package is available starting with EXEC level 38R1. This DA enhancement is needed for a program that heavily uses a large amount of virtual address space.

M.13. CPU Performance

M.13.1. Generated Code

When you make a reference to a virtual object, the ASCII FORTRAN compiler must generate a decomposition code sequence. This can add three to seven machine instructions to a simple reference of an array element. One of these instructions is either an LBJ, which is a very slow instruction, or a link to an activation run-time routine. There are several variations of code sequences that reference items in virtual space. Some of these sequences have a test around the LBJ instruction or around the link to an activation run-time routine. These test sequences are used when the probability of a bank change occurring is low. For example:

```
A (I) = A (I+100) - A (I-1)
```

The decomposition code sequences are generated when referencing an array that is known to be in virtual space by its declaration, or is a dummy argument. It is not known whether an

argument is in virtual space or not, so a special virtual code sequence must be used when a VIRTUAL statement is in the subprogram.

Scalar arguments or scalars in virtual space must also be activated. No decomposition sequence needs to be done, but an LBJ (and possibly a preceding test instruction) must be executed to reference the scalar. Since scalars take up little space, we strongly recommend that you place as few scalars as possible in virtual space. For heavy use of scalar dummy arguments in a subprogram, it is faster to move them to local variables in the subprogram to cut down on unnecessary LBJ instructions that can cause unwanted page traffic to occur.

A scalar in virtual space that resides wholly under a relative of address 2K in its location counter has better code generated to reference it. A full decomposition sequence need not be generated, though the LBJ activation is still needed. The LBJ is expensive and can cause page traffic, so we again recommend that you keep scalars and small arrays out of virtual space.

ASCII FORTRAN-generated code uses index registers to reference all data when the O option (over 65K addressing option) is used. This is normally needed when the last address of the collected program is over 65K. Use of this option may be needed in a virtual FORTRAN program when the control bank that holds all nonvirtual data is too large, and truncation errors result during collection. However, the location and size of the banks defining virtual space do not affect use of the O option. The code generated to reference nonvirtual data is faster without the O option. Therefore, the O option is not assumed by default.

M.13.2. Input/Output

M.13.2.1. Striping and Implied-DOs

A system that needs a large amount of virtual space may also need to do a large amount of I/O on these large arrays for input and results. Unformatted I/O on a whole large array does striping to avoid heavy addressing arithmetic and LBJ usage.

Example:

```
VIRTUAL A
REAL A(2000,1000)
READ(10,END=20,ERR=22) A
```

An implied DO in the I/O statement causes CPU usage on the statement to increase dramatically, because the compiler generates codes for each element reference and a control transfer occurs between this code and the I/O complex for each array element.

Example:

```
VIRTUAL A
REAL A(2000,1000)
READ(10,END=20,ERR=22)((A(I,J),I=1,2000),J=1,1000)
```

You often want to do a large block of I/O from a contiguous area of an array but cannot simply use the whole array name as a list item because of one of the following:

- The starting point is a variable
- The end point is a variable

- You only want to do one column of a multidimensional array

Thus, you can only use the inefficient implied DO in the I/O list.

Example:

```
SUBROUTINE X(NSTRT,NEDN,A,INX)
COMMON/CX/C(100000),C2(4000,20)
DIMENSION A(2000,1000)
READ(10) (A(I),I=NSTRT,NEND)
READ(1 I) (C(I),I=NSTRT,NEND)
WRITE(12)(C2(I,INX),I=1,4000)
END
```

You can change the previous example to do more efficient whole-array I/O by the method shown below:

```
SUBROUTINE X(NSTRT,NEND,A,INX)
COMMON/CS/C(100000),C2(4000,20)
CALL ARY10(.TRUE,10,A(NSTRT),NEND-NSTRT+1)
CALL ARY10(.TRUE,11,C(NSTRT),NEND-NSTRT+1)
CALL ARY10(.FALSE,12,C2(1,INX),4000)

SUBROUTINE ARY10(READ,UNIT,ARY,SIZE)
LOGICAL READ
INTEGER UNIT,SIZE
REAL ARY(SIZE)
IF (READ) THEN
    READ(UNIT) ARY
ELSE
    WRITE(UNIT) ARY
ENDIF
END
```

NOTE: This method applies to FORTRAN programs that either do or do not use virtual space.

M.13.2.2. Buffer Sizes

When you do unformatted I/O on large arrays, the OPEN statement should specify a reasonable block size and segment size. The defaults of 111 words for segment size and 224 words for block size are not appropriate when transferring millions of words of data (see Appendix G and 5.10.1). Control bank space is used for I/O buffers. You cannot define very large buffer sizes such as 300K words.

M.13.2.3. NTRAN$

The NTRAN$ service routine can do very efficient, primitive I/O on any size virtual object. (See 1.10.)

M.13.3. Intrinsic Functions

The ASCII FORTRAN compiler moves banked or virtual array elements to local storage when these are passed as arguments to selected mathematical intrinsic functions.

M.14. Timings

The following tests show the effect of various addressing decomposition code sequences on execution times. The results are also compared to nonvirtual FORTRAN timings. The basic test is a simulation of the following FORTRAN program segment:

```
      REAL A(1048576),B(1048576),C(1048576),D(1048576)
      DO 10 I = 1,1048576
10    A(I) = B(I)*C(I) + D(I)
```

Several different decomposition code sequences are run (called A, B, C, D, E, etc. in Table M-1), some having tests around the LBJ instruction or activation call.

This program shows ASCII FORTRAN nonvirtual code-generation capabilities in the best light, since all array references are strength-reduced, and a free auto-increment can be done on X-registers for each array reference. It is an inefficient example for the virtual approach since each array reference requires a full decomposition sequence and LBJ. Real programs are never nearly as optimizable for ASCII FORTRAN as this one, and are hopefully not as inefficient as this one is for virtual FORTRAN.

Test setups:

- TEST1: Four simulated one-million-word arrays, each getting eight 128K banks. (The arrays are 1,048,576 words each, octal 04000000.)
- TEST2: Four 32K arrays all in one bank. An outer loop brings the number of iterations up to be the same as the four one-million-word arrays. The purpose of this test is to see what effect the test instructions skipping around the LBJ or activation calls has.

The following control tests are done for comparison purposes using ASCII FORTRAN level 10R1 with OZ compile options. An outer loop brings the number of iterations up to one million.

- **TEST5**: No banking.
- **TEST6**: Banking, all arrays in one bank, a BANK statement is supplied to the compiler.
- **TEST7**: Banking, each array in a different bank, four BANK statements are supplied to the compiler.
- **TEST8**: Banking, all arrays in one bank, no BANK statements, but BANKED=ALL used to indicate banking.
- **TEST9**: Banking, each array in a different bank, no BANK statements, but BANKED=ALL used to indicate banking.

In Table M-1, the VA/addr column gives the registers that the virtual address is taken from, and the register that the absolute address resides in after activation. A*x* means a single A*x*-register such as A0, A1, A2, A3. Pair means an even-numbered non-A*x*-register pair, such as A4-A5, A6-A7.

Example:

A/A*x* The VA is in an A-register, and the absolute address is in an A*x*-register after activation.

Table M-1. Timings (in seconds)

	Code Sequence	Added Instr. /Ref.	VA/Addr.	TEST1 Time	TEST2 Time	Activate Link
Nontest Sequences	C	5	A*x*/A*x*	20.93	20.93	LBJ
	EAO	6	Pair/A*x*	21.15	21.15	LBJ
Nonargument Test Sequences	TLEABS1	4	A*x*/A*x*	27.71	10.44	LMJ
	FAO	6	Pair/A*x*	25.16	10.87	LMJ
Argument Sequences	TLARGB	5	A*x*/A*x*	29.82	12.60	SLJ
	TLARGU	6	Pair/A*x*	31.11	15.09	SLJ

Control Tests:

	Test	Time	Comments
FTN 10R1:	TEST5	2.663	Control test, no banking done
	TEST6	2.647	One LBJ per 32K iterations
	TEST7	14.991	Four LBJs per iteration
	TEST8	6.748	One SLJ, total
	TEST9	33.902	Four SLJs per iteration

These tests were done on a Series 1100/80 using a UNISCOPE 200 display terminal. We obtained CPU timings by executing a TIME program before and after each execution.

M.15. Efficiency Suggestions

- Avoid using the ALL option of the VIRTUAL statement because it places all named common into virtual space. Separate out the largest arrays, and put those arrays into one or more common blocks in virtual space. (Large local arrays can also be named in the VIRTUAL statement.)

- Keep scalers (and small arrays) out of virtual space as much as possible. When heavy use (especially in loops) of scalar arguments occurs, move them to local variables after subprogram entry and use the local copies in the subprogram. (The compiler often does this itself when it is safe to do so.)

- When a subprogram has many array arguments that you know are usually in nonvirtual space, drop some of these arguments and use nonvirtual common instead.

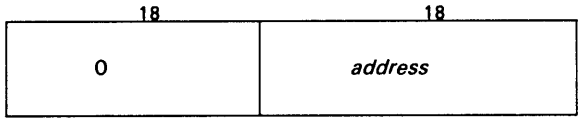
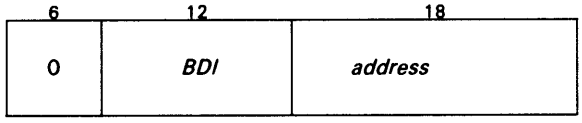
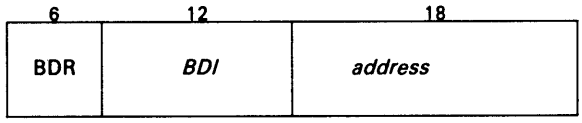
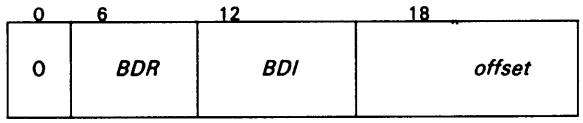
- When possible, organize your code so that references to items in virtual space iterate in small increments, rather than randomly or in large increments. This can minimize bank switching and thrashing.

NOTE: Use of the MOVWD$ service routine to move data from one array to another often aids efficiency.

- Increase the default buffer sizes for files having heavy unformatted I/O on large arrays. Use whole arrays for I/O list items when possible. See the method shown in M.13.2.1.
- Use the smallest bank size possible for virtual space to help minimize thrashing potential.

M.16. Argument Forms

When arguments pass to a FORTRAN program, a packet is generated that contains one address word for each argument. There are four address forms that can pass for an argument in ASCII FORTRAN.

Form:	Fields:	Description:
A		Unbanked form
B		Banked form without BDR
C		Banked form with BDR
D		Virtual address form

The 6-bit BDR field in forms C and D is set as follows (the bits are numbered from left to right, starting at one by the FORTRAN convention):

- Bit 1: 0
- Bits 2 - 3: BDR number.

  BDR1 Utility I-bank
  BDR2 Main D-bank
  BDR3 Utility D-bank

- Bits 4 - 6: 0

The different forms are used for the following purposes:

- Form A is passed for arguments known to be in the control bank at compile time. This includes nonvirtual and nonbanked data (local and common), constants, and temporaries (expression results).

- Form B is passed for arguments that are banked items declared with the standard banking mechanism (the BANKED=ALL COMPILER statement option, or a BANK statement with a common block list).

  Form B is also used in I/O packets and packets for character run-time routine calls. In addition, it is used internally by generated code for certain character array arguments when VIRTUAL is on. Normally, a form B address converts to a form C address for internal use by compiled code.

- Form C is used when passing a dummy argument as an actual argument when BANKED=DUMARG is on, and for passing a dummy scalar argument as an actual argument when VIRTUAL is on.

- Form D, the virtual address form, is passed for variables in virtual space. The BDR can be 1 or 3 (it is defined in element F2BDREQU$). It cannot be 0 or 2 because then a virtual address is not unique (that is, forms C and D look identical). Form D is used for virtual items only. The bit size *n* is defined according to the virtual bank size as follows:

Virtual Bank Size	Value of *n*
4K	6
8K	5
16K	4
32K	3
64K	2
128K	1

NOTE: K means 1,024 words.

There is no ambiguity between the four forms because of the BDR field used in forms C and D. This field is used by subprogram prolog code and run-time routines to distinguish between the forms.

To get this uniqueness of forms, the following assumptions are made concerning the maximum number of pages in the program:

500	when banking with no virtual
500	when virtual with bank size 4K
1,000	when virtual with bank size 8K
2,000	when virtual with bank size 16K, 32K, 64K, or 128K

NOTE: The defaults are 200 pages of 32K words each.

Nonvirtual FORTRAN with standard banking handles forms A, B, and C. Virtual FORTRAN handles all four argument forms.

M.16.1. Processing Dummy Scalars

In a virtual FORTRAN subprogram-generated code, dummy scalar references use a banking sequence that involves an LBJ (or test and LBJ), but no virtual address decomposition. Therefore, dummy scalars must have their argument words converted to form C by a run-time call in the subprogram prolog code. In nonvirtual FORTRAN with standard banking, a banking sequence and form C is used for all dummy scalars and dummy arrays.

M.16.2. Processing Dummy Arrays

There are a large number of decomposition code sequences that can be used on a virtual address. Arguments pose a special problem since it cannot be determined at compilation time that they are in virtual space. When these dummy arguments are not in virtual space, they need not be activated by an LBJ since they are always visible. A coding sequence manipulating fields of a virtual address has a problem when used on an array argument that is in nonvirtual space. Once the array is indexed past the collected virtual page size, a BDI switch occurs. This means that a program using 4K-sized virtual pages cannot safely pass a control bank (nonvirtual) array larger than 4K words in size as an argument. This is a poor restriction. There are other activation sequences that do not do a simple decomposition into fields.

Assume that two words in each ID area (RANGEABS and RANGEABS+1) give the absolute address range for that virtual page. Another ID area word (MAGICCELL) is structured such that the virtual address of any word in that bank added to it gives the absolute address of that word. Assuming that register X10 points at the virtual ID area, the following range test code sequence is possible for items declared in virtual space.

Sequence TLEABS1:

```
.
.                                  .VA - > A0, VA means virtual address
.
A    A0,MAGICCELL,X10              .Make absolute address if visible
TLE  A0,RANGEABS+1,X10             .Address too big?
TLE  A0,RANGEABS,X10               .No. Too small?
LMJ  X11,ACTA0                     .Wrong bank visible, fix it.
.
.                                  .Use address in A0
.
```

An argument may or may not be in virtual space, so the canned X10 pointer to the virtual ID area is insufficient; a control bank argument always causes an out-of-line activation call. So, each argument needs an associated ID area pointer to be used by a variation of the above sequence. (Even the control bank has an ID area.)

Sequence TLARGB:

```
.
.                                   .VA - > A0, VA of elt of X to A0
.
L    X11,ARGXIDPTR                  .Ptr to ID area for arg X
A    A0,MAGICCELL,X11               .
TLE  A0,RANGEABS+1,X11              .
TLE  A0,RANGEABS,X11                .
SLJ  X11,ACTA0X                     .
.
.                                   .Use address in A0
.
```

The control bank ID area defines its absolute address range in cells RANGEABS and RANGEABS+1 as FIRST$ to 0777777. The control bank ID area has a MAGICCELL of zero and argument prolog code uses an absolute address for control bank array arguments rather than a virtual address when creating the virtual origin variable for a dummy array. Use of range test sequences like the above on array arguments means that:

- the code sequence works correctly for both virtual and nonvirtual arguments
- no bank needlessly activates when already based, even control bank arguments
- artificial size restrictions do not have to be placed upon nonvirtual arrays when they are passed as arguments (even when they are in banked space)

These test sequences are used to reference dummy array arguments.

M.16.3. Example of Argument Forms Passed

The following example has items in virtual space, banked space, and the control bank:

```
SUBROUTINE x(e,b)
VIRTUAL /C1/
COMPILER(BANKED=ALL)
COMMON/C2(1000)
DIMENSION e(100000)
COMMON /C1/d(10000000
CALL y( 1, b, c(i), d(i), e(i) )
END
```

The following list shows the address forms in the parameter packet passed to subroutine y from subroutine x:

Argument	Description of Argument	Address Form
1	Constant (control bank)	A
b	Dummy scalar	C
c (i)	Element of banked array	B
d (i)	Element of virtual array	D
e (i)	Dummy array element	A, C, or D

When dummy array e is a control bank array, from A is passed; when e is in virtual space, form D is passed; and when e is an array in banked space, form C is passed.

M.17. Library Utility Routines

M.17.1. VACTIV$

Purpose:

Use VACTIV$ to base an item's bank and return its absolute address. Only experienced programmers should use this routine.

Form:

```
L    A0,address
LMJ      X11,VACTIV$
```

Description:

Use the VACTIV$ linkage protocol only from MASM routines. It is called from the FORTRAN library, including I/O, and user MASM routines to base an item's bank (if it has not already been based) and returns its absolute address. It replaces the old ACTIV$ routine that could not handle virtual addresses. The linkage is LMJ X11,VACTIV$. VACTIV$ is in the control bank, so an IBJ$ or LIJ need not be done. All registers are restored except X11 and A0.

Example:

Input:

A0 has an address in one of the four forms: A, B, C, or D.

Output:

H2 of A0 contains the item's absolute address. Also, the item's bank is based.

NOTE: H1 and A0 may be nonzero on the return.

Several routines aid you in enhancing CPU performance of virtual or banked programs. With the exception of the MOVWD$ and MOVCH$ routines, the following routines should only be used by programmers skilled in the use of assembly language, and familiar with multibanking and the ASCII FORTRAN subprogram linkage conventions. The MOVWD$ and MOVCH$ routines on the other hand are easy to use, and their use in a few key places enhances performance. All of these routines can be called from FORTRAN programs.

Several examples can be found in M.17.4, following the presentation of all of the routines.

M.17.2. Service Routines

M.17.2.1. LOCV$

Purpose:

LOCV$ is an extension of the ASCII FORTRAN LOC service routine that handles virtual addresses.

Form:

I = LOCV$(*arg*)

where: *arg* is a FORTRAN variable. The argument can be in the control bank, in banked space, or in virtual space.

Description:

The address of the passed-argument is returned. The form A address word is returned when the argument is in nonvirtual space. A form D address word is returned if the argument is in virtual space.

NOTE: When the item is type character, its offset is ignored. The address word passed for the argument can be in one of the forms A, B, C, or D. An item in virtual space can only have a form C or D address word.

M.17.2.2. CVVA$F

Purpose:

CVVA$F returns information about the bank in which the argument resides.

Form:

DP = CVVA$F(*arg*)

where: *arg* is a FORTRAN variable.

Description:

This routine returns information in registers A0 and A1. The address returned in register A0 is in one of three forms: A, C, or D. The form A address word is returned for a control bank argument, form C is returned for an argument in banked space, and form D is returned for an argument in virtual space. Register A1 holds the ID area pointer for the bank in which the argument resides.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

NOTE: An ID area pointer for a bank has the BDR+BDI in H1, and H2 is the address of the 64-word ID area for that bank.

M.17.2.3. CVBK$F

Purpose:

CVBK$F returns information about the bank in which the argument resides.

Form:

DP = CVBK$F(*arg*)

where *arg* is a FORTRAN variable.

Description:

This routine returns information in registers A0 and A1. The address returned in register A0 can be one of two forms: A or C. The form A address word is returned for a control bank argument, and form C is returned for an argument in banked or in virtual space. Register A1 holds the ID area pointer for the bank in which the argument resides.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

M.17.2.4. CVBK$I

Purpose:

CVBK$I returns information on the BDI portion of the address contained in its argument.

Form:

DP = CVBK$I(*arg*)

where *arg* is a FORTRAN variable holding an address.

Description:

CVBK$I is an indirect version of CVBK$F. This routine returns information in registers A0 and A1. The address returned in register A0 is in one of two forms: A or C. The form A address word is returned for a control bank address, and form C is returned for an address representing banked or virtual space. Register A1 holds the ID area pointer for the bank that the address represents.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

M.17.2.5. LBJ$IT

Purpose:

LBJ$IT performs an LBJ instruction to base a desired bank.

Form:

CALL LBJ$IT(*arg*)

where *arg* is a one-word FORTRAN variable or array element. It holds a form C address word for a bank that is to be based. The address can be for virtual space, banked space, or the control bank.

Description:

An LBJ instruction is done on the passed argument to base the desired bank. Then a return takes place to the caller.

NOTE: The argument is not checked for validity. H2 of the argument (the address) is not used and need not be a valid address in the bank.

M.17.2.6. FTNWB$

Purpose:

FTNWB$ performs a full walkback trace.

Form:

CALL FTNWB$

NOTE: There are no arguments.

Description:

This walkback call doesn't pull the FTNPMD complex into your collected program. When the FTNPMD complex is collected in your program, this call activates a full walkback trace from the point of call; then normal execution resumes. When the FTNPMD complex is not collected in your program, the call has no effect.

NOTE: The FTNPMD complex is brought into a collection when one or more FORTRAN relocatables are compiled with the F option, when you include element FTNPMD1 in your collection with the INCLUDE directive, or when you call the FTNPMD complex entry points FTNPMD or FTNWB in the FORTRAN program.

M.17.2.7. MINE$F

Purpose:

The MINE$F routine is a strip-mining routine that is callable by a FORTRAN program. It is meant to be used as a primitive in strip-mining basic operations on virtual banks.

Form:

I = MINE$F(*present*,*direct*,*next*)

where (the arguments on input to MINE$F are):

present is a variable holding an address in one of the forms A, B, C, or D. You want to know the number of words remaining in the bank indicated by this address.

direct is an integer variable that indicates the direction taken to check the amount remaining in the bank.

When *direct* .GE. zero, movement is forward.

When *direct* .LT. zero, movement is backwards.

The values that the arguments to the MINEF$ routine contain when the function returns are:

present Unchanged

direct Contains a signed integer value that indicates the remaining number of words in a virtual bank. When the *present* argument is not for virtual space, the value returned is a positive or negative 0777776 depending on the direction as specified by the *direct* argument on input.

next Contains the form D address of the start of the next virtual bank when the flag in *direct* is positive. When the flag is negative, the virtual address points to the end of the previous virtual bank. When the *present* argument does not hold an address for virtual space, the address returned in *next* is 0.

I The function value returned is the original item address as held on entry in *present*, but possibly changed to another form. Two forms of output for I are:

1. The function result is a form A address when *present* holds a control bank address.

2. The function result is a form C address when *present* holds an address for banked or virtual space.

Description:

The caller wants to know the number of words remaining in a virtual bank from some present point (*present*). The FORTRAN variable *present* holds an address in one of four forms: A, B, C, or D. The direction can be forward or backward as indicated by *direct* being positive or negative. The function result can be used in an LBJ instruction to base the bank where the stripe resides.

M.17.2.8. MOVWD$

Purpose:

The MOVWD$ routine moves word-oriented items efficiently. It can also broadcast a scalar item to an array. Either *source* or *target* or both items can be banked or virtual.

Form:

CALL MOVWD$(*source*, *srceincr*, *target*, *trgtincr*, *precision*, *itercount*)

where

source is the item that is moved to a target.

srceincr is an integer expression indicating the increment in elements to find the next source item to be moved. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar and the scalar is broadcast in the target array.

target is the destination item that is filled from the source.

trgtincr is an integer expression indicating the increment in elements to find the next target item to be stored to. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar.

precision is an integer expression indicating the number of words of storage per array element of the source and target items. The precision is assumed to be the same for both source and target. The values for *precision* are: one, two, or four words.

itercount is an integer expression indicating the number of iterations or loops of this move. For zero or negative values of *itercount*, no moves are done. This form of a zero-trip DO loop allows you to easily replace simple loops that move data with CALL statements to MOVWD$.

NOTE: Character items can be used for source and target items only when the character items have no offset and the size of their array elements are one, two, or four words.

M.17.2.9. MOVCH$

Purpose:

Use the MOVCH$ routine to move character items efficiently. It can also broadcast a scalar item to an array. Either *source* or *target* or both items can be banked or virtual.

Form:

CALL MOVCH$(*source*,*srceincr*,*target*,*trgtincr*,*itercount*)

where

source is the character item to be moved to a target.

srceincr is an integer expression indicating the increment in elements to find the next source item that is moved. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar and the scalar is broadcast in the target array.

target is the destination character item that is filled from the source.

trgtincr is an integer expression indicating the increment in elements to find the next target item to be stored to. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar.

itercount is an integer expression indicating the number of iterations or loops of this move. For zero or negative values of *itercount*, no moves are done. This form of a zero-trip DO loop allows you to easily replace simple loops that move data with CALLs to MOVCH$.

NOTE: The target and source items do not have to be the same character lengths. The normal rules of truncation or filling with blanks for character assignment statements are followed by the MOVCH$ routine.

M.17.3. Virtual Storage Allocator

There are three routines to allocate virtual storage. Two allocate static virtual storage for virtual common blocks and static local virtual variables. One allocates and frees dynamic virtual space for local virtual storage in the automatic class. These routines are called by generated code, or explicitly called by FORTRAN programs.

NOTE: The virtual storage allocators acquire about 500 words of control D-bank storage by use of the MCORF$ routine. Other utilities that you use can also acquire storage in a similar manner.

M.17.3.1. SALC$P

Purpose:

The SALC$P routine allocates static virtual storage in a packed manner.

Form:

VA = SALC$P(*cbnam* ,*size* ,*flag* ,*base*)

where

cbnam is a one-word code (or name) used as a tag on this allocation. It must be unique for all allocations. The compiler uses the common block name in Fieldata for virtual common blocks. It uses a created name for local static virtual space.

size is the size in words of this virtual allocation.

flag is either .TRUE. or .FALSE. When .FALSE., this allocation is dynamically initialized by the caller to simulate DATA statement operation. It prints run-time diagnostics.

base is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator.

Description:

The function result is the form D virtual address of the first word of this allocation. The virtual storage allocator SALC$P allocates *size* words of virtual space to the static virtual object of name *cbnam*. When the object already exists in virtual space, the existing allocation is used. When *size* is greater than the existing size on previously allocated object, an attempt is made to expand it. Expansion is possible when the object is the last one allocated in static virtual space, or when there is room between the end of its current allocation and the next allocation.

When expansion is not possible, a nonfatal run-time diagnostic occurs. When the object is already allocated, and *flag* is .FALSE., indicating that the object undergoes dynamic initialization on the return, a nonfatal run-time diagnostic occurs.

When the object is a common block and another program unit has already used the common block but did not indicate it as being a virtual object, error termination occurs. When not enough virtual space is available for the allocation, error termination also occurs.

NOTE: The size and number of D-banks that virtual space uses is defined by the main program, or by the library element VSPACE$ when the main program is not FORTRAN or does not contain a VIRTUAL statement.

The routine SALC$P allocates virtual space in a packed manner and is the compiler default. This means that each allocation starts where the last allocation left off instead of at the beginning of a new D-bank. However, all virtual allocations start on a 64-word boundary in virtual space, and the first 2,048 words of an allocation are in one D-bank. (The 64-word boundary minimizes potential bank-spanning problems, and the first 2,048 words being in one bank allows efficient coding to reference scalars in the first 2,048 words of a virtual common block.)

M.17.3.2. SALOC$

Purpose:

The routine SALOC$ allocates static virtual space in an unpacked manner.

Form:

VA = SALOC$(*cbnam* ,*size* ,*flag* ,*base*)

where

cbnam is a one-word code (or name) used as a tag on this allocation. It must be unique for all allocations. The compiler uses the common block name in Fieldata for virtual common blocks. It uses a created name for local static virtual space.

size is the size in words of this virtual allocation.

flag is either .TRUE. or .FALSE. When .FALSE., this allocation is dynamically initialized by the caller to simulate DATA statement operation. It prints run-time diagnostics. When .TRUE., this allocation is not dynamically initialized.

base is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator.

Description:

SALOC$ operates identically to SALC$P, except that it allocates virtual storage in an unpacked manner, starting a new allocation at the beginning of a new D-bank.

M.17.3.3. DALC$P

Purpose:

The routine DALC$P allocates and frees dynamic local virtual space.

Form:

VA = DALC$P(*size* ,*base*)

where

size is the size in words of this virtual allocation. When negative, it is a deallocation call.

base is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator (allocation call only).

Description:

The function result is the form D virtual address of the first word of this allocation. Dynamic virtual space is allocated in a LIFO (last-in-first-out) manner. A deallocation call *size* must be the negative of the allocation size, or error termination occurs. Dynamic virtual space is allocated from the last virtual D-bank backwards, and static virtual space is allocated from the first virtual D-bank forwards. This separates the two forms of allocation so that fragmentation of virtual space does not occur. When not enough virtual space is available for the allocation, error termination occurs. On a deallocation call the *base* argument isn't set, and no function result is returned.

M.17.4. Examples Using the Virtual Feature

Example 1:

Example 1 shows changes that can be done to enhance performance of programs that use virtual or banked space heavily. This program does a simple sum reduction of a REAL array. The code is:

```
      PARAMETER(N=<size>)
      REAL A(N), SUM
      SUM=0
      DO 10 I=1,N
10    SUM=SUM + A(I)
      PRINT*,SUM
```

Now let's define a service routine that performs a sum reduction, and replace the loop with a call to it:

```
      PARAMETER(N=<size>)
      REAL A(N),SUM,SUMRED
      SUM=SUMRED(A,N)
      PRINT*,SUM
```

The SUMRED service routine does a sum-reduction by strip-mining portions of its input array, basing the stripe to make it visible, and performing several adds in its inner loop to hide the costs of storing the sum and the JGD instruction. It references the source array using base-offset type of referencing, which means it can access the source array in virtual space, banked space, or the control bank.

```
      REAL FUNCTION SUMRED(SOURCE,ITER)
      VIRTUAL
      COMPILER(PROGRAM=BIG),(U1110=OPT)
      IMPLICIT INTEGER(A-Z)
      REAL SOURCE(*),DUM(1),DUMY
      DEFINE DUMY(i)=DUM(i+OFF)

      SUMRED=0
      REMAIN=ITER
      NOWDS=1                          @ go forwards in source
      START=CVVA$F(SOURCE)             @ starting point in input array
      LOCDUM=LOCV$(DUM)                @ abs. addr. of local array DUM
C
C     Calculate a stripe size and address, set up for
C     base-offset referencing using DUMY, base the stripe,
C     do a sum reduction on the stripe.
C
1     START=MINE$F(start,nowds,next)
      IF (NOWDS.EQ.0) RETURN
      IF (BITS(START,1,18).NE.0)CALL LBJ$IT(START)
      OFF=BITS(START,19,18)-LOCDUM     @ offset to stripe

      NUM=MIN(REMAIN,NOWDS)            @ # words in this stripe
      LITTLE=MOD(NUM,4)
      DO 10 I=1,LITTLE                 B do remains of MOD 4 first
10    SUMRED=SUMRED+DUMY(I)
      DO 20 I=LITTLE+1,NUM,4           @ do most of the stripe
20    SUMRED=SUMRED+DUMY(I)+DUMY(I+1)+DUMY(I+2)+DUMY(I+3)

      REMAIN=REMAIN-NOWDS
      IF(REMAIN.LE.0)RETURN
      START=NEXT
      GO TO 1
      END
```

This SUMRED example can be extended to have increments other than one. Similar routines can handle primitives other than sum reductions.

This program was executed on a Series 1100/80 with an array size of 67,000 words. It was timed with the array in virtual space, and in the control bank. The original program executed once with the simple loop and once with the modification to call SUMRED.

The following shows the Series 1100/80 CPU timings in seconds:

	Nonvirtual Array (in Control Bank)	Virtual Array (32K Banks)
Original Program	0.1141	0.3118
Program Modified to Call SUMRED	0.0637	0.0638

There are 67,000 floating-point adds performed. Using the 0.0638-second time, that is over one million floating-point operations per second (one MFLOP).

Even the nonvirtual sum reduction has its CPU time cut approximately in half by calling SUMRED. This is due to the SUMRED routine stringing-out the inner loop by doing four operations per iteration. The original program can also do this.

Example 2:

The following example shows the use of the MOVWD$ call to enhance performance on an inner loop:

```
      SUBROUTINE SUB(ARG,N)
      VIRTUAL/CX/
      COMMON/CX/C(2000,2000)
      REAL LOCAL(2000),ARG(2000)
      DO 10 I=1,2000

      LOCAL(I)=C(I,N)
10    ARG(I)=C(I,N+1)
        .
        .
        .
```

The loop can be replaced by:

```
      CALL MOVWD$(C(1,N),1,LOCAL,1,1,2000)
      CALL MOVWD$(C(1,N+1),1,ARG,1,1,2000)
```

The subprogram SUB is called 100 times from a driver, where the N passed as the second argument is the loop index. Timings are done on the original program, the program modified to call MOVWD$, with variations of the argument array being in the control bank or in virtual space. The default page size (32K words) is used for virtual space.