HP64000 Logic Development System

Pascal/64000 Compiler Supplement 6809

HEWLETT PACKARD

FOLD HERE

HEWLETT
PACKARD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Attn: Logic Publications Dept.
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form
will be greatly appreciated. Thank you.

READER COMMENT SHEET

Part Number: 64813-90903

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1	2	3	4	5	Covers everything

2. The information in this book is accurate:

Too many errors	1	2	3	4	5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1	2	3	4	5	I can find info quickly

4. The Index and Table of Contents are useful:

Helpful	1	2	3	4	5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1	2	3	4	5	Very helpful
Too many now	1	2	3	4	5	I'd like more

6. What about the writing style:

Confusing	1	2	3	4	5	Clear

7. What about organization of the book:

Poor order	1	2	3	4	5	Good order

8. What about the size of the book:

too big/small	1	2	3	4	5	Right size

Comments: ______________________________

Particular pages with errors?

____

Name (optional): ______________________________
Job title: ______________________________
Company: ______________________________
Address: ______________________________

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

Pascal/64000 Compiler Supplement 6809

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number on the back cover changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions.

First Printing........... May 1981 (Part Number .... 64813-90903)
Second Edition ....February 1983 (Part Number .... 64813-90903)
Change 1 .... March 1984

Table of Contents

Table of Contents (Cont'd)

List of Tables

Chapter 1

Pascal/64000 Compiler 6809

Introduction

General

This compiler supplement is an extension of the Pascal/64000 Compiler Reference Manual. It contains all the processor-dependent compiler information for use with the 6809 microprocessor.

Descriptions of compiler features, options, and their use are supplied. A detailed discussion of the run-time libraries required by the 6809 code generator is included. In addition, a brief discussion of the features, capabilities, and limitations of Pascal program development using the emulation is provided.

Pascal Program Design

Pascal programs should be designed to be as processor and implementation independent as possible, yet certain concessions must be made when the processor has unique characteristics. Programs written to run on a large mainframe computer with megabytes of virtual memory may not run on a 6809 with a maximum of 64k-bytes of addressable memory. Most large mainframe computer implementations have enough memory to allocate a stack area and a heap for dynamic memory allocation with no prompting by the user. In a limited memory system these factors must be communicated to the compiler in some manner. For the 6809, the user must specify the location of the stack and, if needed, the location of a memory pool for dynamic allocation routines. The following sections describe subjects related to programming and compiling Pascal/64000 for the 6809 processor.

How to Implement a Program

The usual process of software generation is as follows:

a. Create a source program file using the editor.

b. Compile the source program.

c. Link the relocatable files.

d. Emulate the absolute file.

e. Debug as necessary.

This chapter will provide insight into each of these processes.

The Source File

The Pascal/64000 compiler takes as input a program source file created with the editor. The basic form of a source file is:

```
"6809"
PROGRAM Name;
  .              {comments}
  .
CONST
    ...;
    ...;
TYPE
   ...;
   ...;
VAR
   ...;
   ...;
PROCEDURE Procedure_name(Parameter1 : Type);
   .
   .
   BEGIN
     .
     .
     .
   END;
BEGIN
  .
  .              {main program code}
  .
END.
```

When source file editing is complete, it is ready for compilation. Notice in the example form that the first line of the source program specifies the 6809 processor. This first line must be the special compiler directive indicating the processor for which the program was written.

Within a Pascal source program, the compiler only recognizes upper-case keywords, but identifiers may be lower case. When using a 64200 emulator, the global identifiers must begin with an upper-case letter if the user wishes to access these names symbolically during emulation. (During emulation, only emulation command keywords may start with a lower-case letter.)

The compiler output may be in two forms, a relocatable file and a listing file (if specified). Descriptions of these files are as follows:

Relocatable file:	If no errors were detected in the source file (called FILENAME:source), a relocatable file (called FILENAME:reloc) will be created. This file will be used by the linker to create an executable absolute file.
Listing file:	If a listfile is specified when the compiler is evoked, a file containing source lines with line numbers, program counter, level numbers, errors and expanded code (if specified) will be generated.

Linking

After all program modules have been compiled (or assembled), the modules may be linked to form an executable absolute file. The compiler generates calls to a set of library routines for commonly used operations such as multiply, divide, comparisons, array referencing, etc. These routines must be linked with the program modules. There are two libraries which may be linked.

The first is a debug library file called DLIB_6809:C6809. This library of relocatable procedures contains some extra code to detect errors such as division by 0, or overflow on multiplication.

The second library is called LIB_6809:C6809. This library, which has only a limited set of error-detection code, should execute slightly faster and take up less space in memory. This library may be linked in place of the debug library after reasonable assurance that the code is error free.

The linker is evoked and the questions asked should be answered as follows:

```
link ...

Object files:  SETSTACK_,MODULE1,MODULE2

Library files: DLIB_6809:C6809

.
.
.

Absolute file name:  PROGRAM
```

In the link listfile, the library routines that are referenced by the compiled code are linked at the end of the last user relocatable PROG and/or DATA areas. This fact must be considered for the proper choice of the stack pointer location, and PROG and DATA link addresses.

Emulation of Pascal Programs

After all modules have been compiled (or assembled) and linked, the absolute file may be executed using the emulation facilities of the Model 64000. The emulator is initialized with the memory mapped in keeping with the target system and the stack pointer initialization in the code.

A program which is designed to run in read-only memory (ROM) should have been compiled with the $SEPARATE$ option. The memory should be mapped to have ROM and RAM as illustrated below.

Linking with Real Numbers

When using real numbers for the 6809, the user must link with the real number support library: RealLIB:C6809. This library supports the Model 64000 Pascal implementation of the IEEE real number standard for both long and short floating point numbers (Pascal data types REAL and LONGREAL). To allow mixed REAL and LONGREAL expressions, all internal real operations are performed using an unpacked real number format with a 64-bit mantissa (fraction), a separate sign bit, and a 16-bit signed exponent.

RealLIB:C6809 will load subroutines in the PROG relocatable area and use the DATA relocatable area for: local data, a default stack area, and a message buffer for error detection.

Since the use of floating point numbers will require additional stack space for temporary computations, this library has a module, BIGSTACK, which will supply a default stack size of 1024 bytes (much larger than that supplied by the default stack in DLIB_6809:C6809 and LIB_6809:C6809. If you have not defined your own stack area and you want to use the default stack, you should load the real library before loading the standard library of your choice.

If you do not supply your own versions of the real error reporting routines, INVALID and REAL_OVERFLOW, the real library will supply them plus a DATA relocatable buffer area for reporting the error condition. See the section on real number libraries in Chapter 4 for more information on real number error detection.

Linking with Pascal File I/O

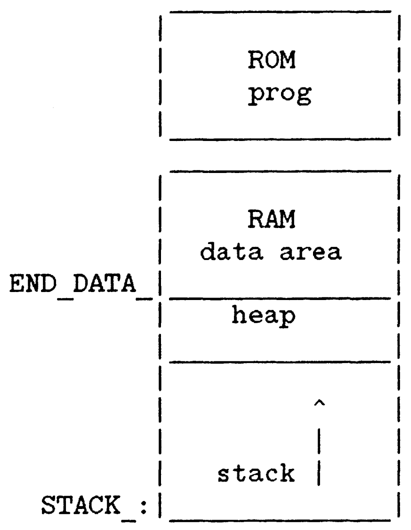
When using the Pascal File I/O features with the 6809, the user must link with the Pascal File I/O support library: PIOLIB:C6809.

If the simulated I/O feature of the emulation subsystem is used, the user should also link the simulated I/O support library: SIMLIB:C6809.

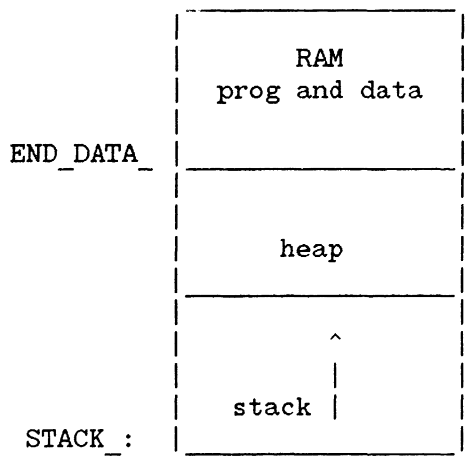
The Pascal/64000 Reference Manual contains a complete machine independent description of the routines in these libraries.

Both libraries are compiled with the options $SEPARATE ON,RECURSIVE OFF$. They will load subroutines in the PROG relocatable area and use the DATA relocatable area for local data and a message buffer for error detection.

See the section on Pascal File I/O in Chapter 5 for more information about the I/O support libraries.



For a program that is designed to run completely in random access memory (RAM), the memory mapping should look like the following:



The transfer address will have been set by the linker so that simply loading the absolute file, and stepping or running the program is all that is required. Note that program execution does not start at address 0000H if the program contains local procedures or functions. However, the <program name> identifier in the program heading is a global symbol and the label of the program transfer address. This program may be executed within emulations by the command:

```
run from <program name>
```

Debugging with DLIB__6809:C6809 Library

When initializing the emulator, it is a good idea to answer yes to the "stop processor on illegal opcode?" question since execution errors may result in a jump into the error handler file, Derrors:C6809.

If, while watching the execution of the code, the status line should indicate "illegal opcode executed at address XXXXH", note the address and enter the command:

```
display local_symbols_in Derrors:C6809
```

The list will roll off the screen; do not stop it with the reset key, since the information which rolls off is not important. When the list has stopped, scan the upper portion of the list for the address at which the illegal opcode occurred. The error type will be listed at the left of this address. (Descriptions of run time errors are given in Appendix A.) The list will also be generated when using library LIB_6809:C6809 by entering the following command:

```
display local_symbols_in Zerrors:C6809
```

The display will now appear as follows:

NOTE

The addresses will change depending upon the link.

Label	Address	Data	
Z_END_PROGRAM	1242H	C3H	Scan this portion
Z_ERR_RANGE	1270H	22H	for the address
Z_ERR_CASE	1258H	08H	where the illegal
Z_ERR_DIV_BY_0	124AH	08H	opcode occurred. The
Z_ERR_HEAP	1268H	08H	data field in this
Z_ERR_OVERFLOW	123CH	08H	portion is not
Z_ERR_SET_CONV	1251H	08H	significant.
Z_ERR_UNDERFLOW	1243H	08H	
Z_ERR_STRING	1235H	00H	
Z_CC_FLAGS	1296H	89H	The data field in
Z_ACC_A	1297H	8FH	this portion may
Z_ACC_B	1299H	F6H	contain useful
Z_REG_X	1298H	F5H	information. The
Z_REG_U	129BH	F0H	addresses in this
Z_CALLER_H	1295H	69H	portion are not
Z_CALLER_L	1294H	A0H	significant.

Some of the errors will load locations with register and stack information.

NOTE

It is important to remember that during emulation of Pascal/64000 programs, a Pascal program may be debugged symbolically (using global symbols in the source program) or by source program line numbers of the form: #1. This is a feature that provides a powerful tool for emulation.

Linking with Real Numbers

When using real numbers for the 6809, the user must linkwith the real number support library RealLIB:C6809.

The library, RealLIB:C6809, supports the Pascal/64000 implementation of the IEEE real number standard for both long and short floating point numbers (Pascal data types LONGREAL and REAL). To allow mixed REAL and LONGREAL expressions, all internal real operations are performed using an unpacked real number format with a 64-bit mantissa (fraction), a separate sign bit and a 16-bit signed exponent.

NOTE

This compiler can generate duplicate symbols in the assembler symbol file for legal Pascal programs. These symbols can be generated by nested procedures with identical names or by procedure or function names that conflict with labels generated by the compiler, i.e., E, R, C, and D procedures labels. Refer to the Pascal Compiler Reference Manual for a description of these labels.

These duplicate symbols can cause ambiguities with some HP 64000 logic analyzer measurements since a reference to a duplicated label may produce an incorrect result.

The compiler produces a warning message whenever it generates a duplicate label to warn the user that use of that symbol in an analysis product may result in an incorrect address being traced. This potential problem can be solved by changing one of the duplicate function names, or by moving one of the functions to another file.

Example Warnings:
```
  *****WARNING ?? - Symbol: Y, is duplicated in the asmb_sym file.
  *****WARNING ?? - Symbol: RY, is duplicated in the asmb_sym file.
```

Chapter 2

Pascal/64000 Programming

6809

Programming Considerations

Introduction

This chapter covers some important requirements of the run-time environment for 6809 Pascal/64000 programs. Although some requirements may not be necessary for every program, the programmer should become familiar with the information supplied in order to use it when the structure of a 6809 program requires it. The specific areas to be discussed are stack pointer initialization, multiple module programs, heap initialization for use with dynamic memory routines (NEW, DISPOSE, MARK, and RELEASE), interrupt processing with Pascal programs, and optional code generation.

Direct Addressing Mode

The 6809 direct page register (DP) is concatenated with any 6809 direct access address to generate the complete run-time address of an object. For example, if the instruction

```
LDA   <25H
```

is generated as a direct addressing instruction, the object that will be loaded into register A will be found at address 25H if the direct page register contents equal 00H. If the direct page register contents equal 0FEH, for example, then the object that will be loaded into register A will be found at address 0FE25H.

The DP register will be initialized to 00H by a 6809 Restart Interrupt. It will never be modified by the 6809 compiler.

The 6809 compiler will generate direct addressing instructions for any object known by the compiler to be located within the address range 00H and 0FFH. For the following Pascal variable declaration:

```
VAR
 $ORG = 20H$
  FLAG: BOOLEAN;
  INFORMATION: INTEGER;
 $END_ORG$
```

the 6809 compiler will generate direct addressing instructions to access variables FLAG and INFORMATION, since their addresses are known by the compiler to be between 00H and 0FFH.

Stack Pointer Initialization

The stack pointer is a hardware register maintained by the processor. Prior to use, however, it must be initialized by the user. A program that has a main code section must generate the following stack initialization statements in the relocatable file:

```
EXT STACK_  .
LDS #STACK_  .
```

Since the EXT statement implies that the label STACK_ has been declared global (GLB) by another program module, the compiler will build a relocatable file, leaving assignment of the STACK_ value for the linker.

If the label STACK_ has not been declared global by any program module, the linker will search the applicable library for a default value. Depending upon which library has been selected by the user, one of the following default values will be selected:

a. If the DLIB_6809:C6809 library is linked, the stack will be assigned 512 bytes in the program (PROG) area of the linked modules.

b. If the LIB_6809:C6809 library is linked, the stack will be assigned 512 bytes in the data (DATA) area of the linked modules.

NOTE

Whenever the LIB_6809:C6809 library is linked, a DATA area location must be specified.

The user should allocate a larger stack when necessary. In particular, recursive programming will generally require a much larger stack than normal to run properly.

Another approach to stack pointer initialization is to define a global variable called STACK_ as shown in the following example:

```
     (file MODULE1:source)
          .
          .
          .
          VAR
             ...;
             ...;
             $GLOBVAR ON$
             $ORG 3F80H
             STACK_AREA : ARRAY[1..128] of BYTE;
             STACK_ : BYTE;
             $END_ORG$
             $GLOBVAR OFF$


          BEGIN
            ...;
            ...;
            ...;
          END.
```

The compiler will generate relocatable code which sets the stack pointer to the address of STACK_ (4000H in this example), and use an area of 129 bytes (3F80H..4000H) for the stack.

This technique will produce both a GLOBAL and an EXTERNAL reference for the symbol STACK_. The relocatable file will produce the proper results when linked. However, if the ASM_FILE option is in effect, the ASM6809:source file will produce an EG (external/global) error. The user should edit the ASM6809 file and delete the EXT STACK_ line before assembling the file.

The use of an absolute address for the stack as in the above example has the user convenience of assigning a fixed block of memory for the stack. It may be better, however, to allow the compiler to actually preserve a relocatable data area for the stack by leaving out the ORG and END_ORG options. This will help prevent accidental reuse of the assigned stack area by another module.

An approach when linking assembly language files is to include the initial stack pointer value or a stack area in an assembly file such as:

```
"6809"
            GLB STACK_   .
STACK_      EQU 2000H      ;puts initial stack
            .              ; pointer at 2000H
            .
            .
```

or:

```
"6809"
            GLB STACK_   .
            DATA
STACKBOT    RMB <stacksize>   ;puts stack
                              ; storage in the
STACK_:     RMB 1             ; DATA area of
            .                 ; the program
            .
            .
```

Note that the address of STACK_ will receive the first data byte being pushed. This file may then be linked with the other program modules generated by the compiler as follows:

```
Object files:   ASMFILE1,MODULE1,MODULE2....
```

Stack Format During Program Execution

Integer values are pushed on the stack: low_byte,high_byte, and popped: high_byte,low_byte.

Execution of PSHS X:

Stack before execution:

```
 ______________________
|          :           |
|__________:___________|  <----------(S)
```

Stack after execution:

```
 ______________________
|          :           |
|          .           |
|______________________|
|   Low_byte  of (X)   |
|______________________|
|   High_byte of (X)   |
|______________________|  <---------(S)
```

Execution of PULS X:

Stack before execution:

```
 ______________________
|          :           |
|          .           |
|__________:___________|
|   Low_byte  of (X)   |
|______________________|
|   High_byte of (X)   |
|______________________|  <----------(S)
```

Stack after execution:

```
 ______________________
|          :           |
|          .           |
|__________:___________|  <----------(S)
```

Static Links. At execution time of any routine X, the compiler needs to have access to routine X's data and parameters; also, it must have access to data of other routines to which routine X is accessible. The data area addresses of non-recursive routines are known at compile time; however, for recursive routines these addresses are pointers to the stack and are determined at execution time. These pointers are called static links.

When any recursive routine is entered in the 6809 compiler, the library routine RENTRY_ is called. RENTRY_ will allocate the routine's data area. It will copy and arrange the parameters. In addition, it will create the necessary static links.

In the following sample program:

```
"6809"
PROGRAM TEST;
VAR I1,I2,I3: INTEGER;

{the $RECURSIVE$ option is on by default}

  PROCEDURE P1(II: INTEGER);
  VAR II1,II2: INTEGER;

    PROCEDURE P2_A(III: INTEGER);
    VAR III1: INTEGER;

      PROCEDURE P3(IIII: INTEGER);
      VAR IIII1: INTEGER;
      BEGIN { P3 }
       III1:=10;
      END;

    BEGIN { P2_A }
     III1:=10;
     P3(III1);
    END;

    PROCEDURE P2_B(III: INTEGER);
    VAR III1: INTEGER;
    BEGIN { P2_B }
     P2_A(III1);
    END;

  BEGIN { P1 }
   P2_B(II1);
   II1:=0;
  END;

BEGIN { Program TEST }
 P1(I1);
END.
```

The stack format at line #19 of the execution of TEST is:

```
                  _________________________
    FFFF         |                         |
    FFFE         |           RA            |
    FFFD         |_________________________|
     :           |                    ^    |
     .           |   P1's data and    |    |
                 |  parameters.       |    |
                 |      (level #1)    |    |
                 |_________________________|
       ,<--<-    |     Static Link #1      |
       `->-->    |_________________________|   <---<--,
                 |           RA            |          |
                 |_________________________|
                 |                    ^    |          |
                 |   P2_B's data and  |    |
                 |  parameters.       |    |          |
                 |      (level #2)    |    |          |
                 |_________________________|
         ,<--<-  |     Static Link #2      |          |
         |       |_________________________|          |
         |       |     Static Link #1      |   ->--->-'
         `->-->  |_________________________|          |
                 |           RA            |
                 |_________________________|          |
                 |                    ^    |
                 |   P2_A's data and  |    |          |
                 |  parameters.       |    |          |
                 |      (level #2)    |    |
                 |_________________________|          |
         ,<--<-  |     Static Link #2      |          |
         |        - - - - - - - - - - - - -           |
         |       |     Static Link #1      |   ->--->-'
         `->-->  |_________________________|  <---------------(S)
```

The stack format at line #16 of the execution of TEST is:

```
    FFFF            |----------------------|
    FFFE            |          RA          |
    FFFD            |______________________|
     :              |                  ^   |
     .              |  P1's data and   |   |
                    | parameters.      |   |
                    |     (level #1)   |   |
                    |______________________|
          ,<--<-    |    Static Link #1    |
          `->-->    |______________________|   <---<--,
                    |          RA          |          |
                    |______________________|
                    |                  ^   |          |
                    |  P2_B's data and |   |
                    | parameters.      |   |          |
                    |     (level #2)   |   |          |
                    |______________________|
           ,<--<-   |    Static Link #2    |          |
           |        |______________________|          |
           |        |    Static Link #1    |   ->--->-'
           `->-->   |______________________|          |
                    |          RA          |
                    |______________________|          |
                    |                  ^   |
                    |  P2_A's data and |   |          |
                    | parameters.      |   |          |
                    |     (level #2)   |   |          |
                    |______________________|          |
           ,<--<-   |    Static Link #2    |          |
           |        ------------------------          |
           |        |    Static Link #1    |   ->--->-'
           `->-->   |______________________|   <--<-, |
                    |          RA          |        |
                    |______________________|          |
                    |                  ^   |        |
                    |  P3's data and   |   |        | |
                    | parameters.      |   |        | |
                    |     (level #3)   |   |
                    |______________________|        | |
           ,<--<-   |    Static Link #3    |        |
           |        |______________________|        | |
           |        |    Static Link #2    |   ->->-' |
           |        |______________________|          |
           |        |    Static Link #1    |   ->-->--'
           `->-->   |______________________|  <---------------(S)
```

The stack format at line #31 of the execution of TEST is:

```
     FFFF           |  ______________________  |
     FFFE           |           RA             |
     FFFD           |__________________________|
      :             |                      ^   |
      .             |    P1's data and     |   |
                    |   parameters.        |   |
                    |      (level #1)      |   |
                    |__________________________|
          ,<--<-    |    Static Link #1        |
          `-->-->   |__________________________| <----------------(S)
```

Recursive Routines - Calling and Returning Sequences

An example of a portion of a program that would be used for calling a recursive routine with no parameters is as follows:

```
             LBSR        receiving_routine
```

An example of a portion of a program that would be used for calling a recursive routine with parameters is as follows:

```
               LDr1          par#1
                .
                .
                .
               LDrx          par#x
               PSHS          r1,...,rx,PC
               LDr1          par#x+1
                .
                .
                .
               LDrx          par#n
               PSHS          r1,...,rx
               LBSR          receiving_routine
```

When returning from a RENTRY_ routine, the stack format at exit will be as follows:

```
 ________________________
|                        |
| RA(calling routine)    |
|________________________|
|                  ^     |
|      Par.'s      |     |
|                  |     |
 - - - - - - - - - - - - -
|                  ^     |
|      Var.'s      |     |
|                  |     |
|________________________|
|                        |
|    Static Links        |
|                        |
|________________________| <------------------ (S)
```

The returning sequence is as follows:

```
LEAS  var_area_size +par_area_size +level*2 ,S
RTS
```

Multiple Module Programs

Only one module in an absolute program file should contain a Pascal program with a main code section. All other modules should contain procedures and functions only, with a period at the end of the procedure declarations to indicate an empty program block.

Example:

```
(file MODULE1:source)

     PROGRAM MODULE1; {this is the main module}

     CONST
         ...;
     TYPE
         ...;
     VAR
         ...;

     PROCEDURE X(Parameter : Type);EXTERNAL;
     PROCEDURE Y;EXTERNAL;

     BEGIN
         ...;
         ...;             {main code}
         ...;
     END.                 {period signals end of program, main code
                           exists so stack initialization code is
                           generated}
```

NOTE

The transfer address is set to cause execution to begin in the main code section of the program module.

```
(file MODULE2:source)

     PROGRAM MODULE2; {this module contains the procedures and
                       functions used in MODULE1}

     $GLOBPROC ON$
     PROCEDURE X(Parameter : Type);
         BEGIN
           ...;
           ...;
         END;
     PROCEDURE Y;
         BEGIN
           ...;
           ...;
         END;
            .        {The period signals the compiler that the
                      program has ended.  Since no main code
                      exists, the compiler does not generate any
                      stack initialization code or linker
                      transfer address}
```

Dynamic Allocation Heap Initialization

Before using standard procedures NEW and MARK, the block of memory that you wish to have managed as a dynamic memory allocation pool must be initialized by calling the external library procedure:

```
INITHEAP(INTEGER(ADDR(END_DATA_)),
  INTER(ADDR(STACK_))-INTEGER(ADDR(END_DATA_))-40H);
```

The procedure must be declared EXTERNAL in the declaration section. The start address should be the smallest address of the memory block to be used. For example, if the block to be used is located from 4000H to 5FFFH, the initialization should appear as follows:

```
PROGRAM Test;

CONST
  .
TYPE
  .
VAR
  .
PROCEDURE
INITHEAP(Start_address,Length_in_bytes:INTEGER);EXTERNAL;
  .
  .
BEGIN  {main program block}
  INITHEAP(4000H,2000H);
  .
  .
  .
END.
```

If the desired location of the heap is at the end of the DATA area, the address of the external library variable END_DATA_ may be used as the start address and as part of an expression to give a length.

Example:

```
BEGIN
   INITHEAP(ADDR(END_DATA_),(ADDR(STACK_)-ADDR(END_DATA_)-40H));
        .
        .
END.
```

This example would reserve 41 hex (or 65 decimal) bytes for the stack and the remainder of the memory from the end of the DATA area to the initial stack pointer -41H for the dynamic allocation routines. This implies that the stack is in a contiguous block with the DATA area. For example, if END_DATA_ is address 1000H and STACK_ is address 2000H, then ADDR(STACK_) - ADDR(END_DATA_) -40H is equal to 0FC0H. The heap will be from address 1000H through 1FBFH (0F00H bytes), and the STACK will be from address 1FC0H through 2000H (see below).

```
                 _______________
                |               |
                | prog and data |
                |               |
     END_DATA_  |_______________|
      (1000H)   |               | Begin_heap
                |      heap     |
                |               |
                |_______________| End_heap (1FBFH)
                |               | End_stack
                |     stack     |
        STACK_  |_______________| Start_stack
       (2000H)
```

Six bytes are used each time the heap is initialized or marked. When an item of four bytes or less is to be allocated, four bytes will be removed from the free list even if less is needed. Likewise, when an item of four or less bytes in size is deallocated, four bytes will be returned to the free list.

Interrupt Vector Handling

The run-time programming environment of Pascal/64000 programs on the 6809 processor has been designed to impose a minimum amount of constraints on the user. As a result the code produced by the compiler is safely interruptable as long as the interrupt driven process restores the registers (which have been automatically pushed onto the stack when the 6809 recognized the interrupt) with a return from interrupt (RTI) instruction.

The 6809 processor supports four types of interrupts: a reset (or powerup) interrupt, a non-maskable interrupt, a maskable interrupt, and a software interrupt. The first three of these are enabled by external control signals to the processor, while the last one is enabled by software program control. When the processor detects one of these interrupts it saves the current status of the processor and jumps to the address in the interrupt vector for that type of interrupt. These vectors are in the last 14 bytes of memory.

For the rest of this discussion assume that the following assembly module defines the interrupt vectors.

```
FILE: IRQ:C6809          HEWLETT-PACKARD: 6809 Assembler

LOCATION OBJECT CODE LINE     SOURCE LINE

                        1 "6809"
                        2 NAME "Interrupt Vector Definition"
                        3
                        4 EXT SOFT_INT_3, SOFT_INT_2, SOFT_INT_1
                        5 EXT FIRQ_INT, IRQ_INT, NMI_INT,
                          RESTART_INT
                        6
                        7 ORG 0FFF2H
                        8
 FFF2    0000           9 FDB SOFT_INT_3
                       10
 FFF4    0000          11 FDB SOFT_INT_2
                       12
 FFF6    0000          13 FDB FIRQ_INT
                       14
 FFF8    0000          15 FDB IRQ_INT
                       16
 FFFA    0000          17 FDB SOFT_INT_1
                       18
 FFFC    0000          19 FDB NMI_INT
                       20
 FFFE    0000          21 FDB RESTART_INT

Errors=    0
```

A Pascal/64000 main program may logically be used as the RESTART_INT to be called on RESTART interrupt. A main program initializes the run time environment for Pascal program execution and ends with the jump to a tight loop at Z_END_PROGRAM (generated by the compiler), thus fitting all the requirements of the RESTART_INT routine.

Pascal/64000 allows the user to define procedures as routines to be called in the interrupt vector by using the $INTERRUPT ON$ option. The $INTERRUPT$ option is only recognized for procedures defined at the outer block of a program. An interrupt procedure needs to be declared global so its address can be available at link time to load the proper interrupt vector. Nothing special is done upon entry to the $INTERRUPT$ procedure. At the end of the procedure the compiler generates a return from interrupt (RTI) instruction instead of a return from subroutine instruction (RTS). An $INTERRUPT$ procedure may not be called like a normal Pascal/64000 procedure because of the RTI return instruction.

The interrupt procedure can have no parameters but it may be compiled in either the $RECURSIVE ON$ or $RECURSIVE OFF$ modes. The $RECURSIVE ON$ mode is required if it is possible to be processing multiple interrupts at the same time.

Any special treatment of interrupts would require some assembly language modules since instructions associated with interrupts are not available in Pascal (SYNC, CWAI, ORCC, ANDCC).

With the previously defined interrupt vector definition the user should compile procedures IRQ_INT, NMI_INT, and SOFT_INT with the $INTERRUPT ON$ option enabled. Care must be taken to turn off this option explicitly. The RESTART_INT should be compiled as a main program, i.e. PROGRAM RESTART_INT.

Set Space Allocation

The 6809 compiler allocates sets by bytes. The Pascal statements:

```
PROGRAM TEST;
 VAR S1: SET OF 0..31;
```

will allocate four bytes of data to the set S1. The bits in the set will be numbered as follows:

```
          7  6  5  4  3  2  1  0   15  14  13  12  11  10  9  8
         _______________________________________________________
        |                          |                            |
S1_addr |         Byte #0          |          Byte #1           |
        |__________________________|____________________________|

         23 22 21 20 19 18 17 16   31  30  29  28  27  26 25 24
         _______________________________________________________
        |                          |                            |
        |         Byte #2          |          Byte #3           |
        |__________________________|____________________________|
```

User Defined Operators

Pascal/64000 allows the user to define his own special operators (user defined operators). User defined operators are created by using the option: $USER_DEFINED$ during the declaration of a user type. The option will apply to the declaration of one user type.

For user defined operators, the compiler will not generate in-line code to perform the operations, instead, it will generate calls to user provided run-time routines. The run-time routine names will be a composite of the user's type name and the operation being performed: TYPENAME_OPERATION. The first eleven characters of the user's type name are concatenated with an underscore and three characters identifying the operation.

Operations

The following is a list of operators that can be user defined and the run-time routine names that the compiler will create when the operations are used on a user type:

	Operation	Symbol	Run-time Routine
1.	Add	+	<typename>_ADD
2.	Negate	-	<typename>_NEG
3.	Subtract	-	<typename>_SUB
4.	Multiply	*	<typename>_MUL
5.	Divide	/ or DIV	<typename>_DIV
6.	Modulus	MOD	<typename>_MOD
7.	Equal Comparison	=	<typename>_EQU
8.	Not Equal Comparison	<>	<typename>_NEQ
9.	Less Than or Equal to Comparison	<=	<typename>_LEQ
10.	Greater Than or Equal to Comparison	>=	<typename>_GEQ
11.	Less Than Comparison	<	<typename>_LES
12.	Greater Than Comparison	>	<typename>_GTR

The compiler will provide the user with a Store routine. The 6809 compiler will use the multi-byte move routine (MBmove).

Parameters

The parameters are passed to this routine by reference, i.e., the addresses of the parameters are passed. For the 6809, the parameters are passed in the following registers:

```
Input:  D contains the address of the first parameter
        X contains the address of the second parameter
        Y contains the address of the third parameter

Output: The result should be assigned through register Y
```

Register Y will not be defined for relational operations. The result should be assigned to register B and the Z flag in register CC should be set according to the following:

```
TRUE  - B set to 1, Z flag is set

FALSE - B set to 0, Z flag is reset
```

Register Y will not be defined for the unary operation Negate; register X will contain the result.

The routines for 6809 user-defined operators can be written in PASCAL; routines can be either static or recursive.

The routines for binary operators should be defined as procedures with three VAR parameters: argument1, argument2, and result. Argument1 always corresponds to the left hand side operand and argument2 to the right hand side operand, as follows:

```
result := argument1 OPERATOR argument2
```

The unary operator Negate routine should be defined as a procedure with two VAR parameters: argument and result.

Finally, the routines for relational operators should be defined as functions with two VAR parameters: argument1 and argument2 and a result of type boolean. Care should be taken to assure that the Z flag is set to the correct value. This may be accomplished by making certain that the last thing the function does is to assign the function result.

NOTE

User operator routines may be defined either as procedures or as functions where the result of the operation is the function result.

Example:

The following program defines and uses the user type "REAL":

```
"6809"
PROGRAM USER_TYPE;
  TYPE
  $USER_DEFINED$
   REAL = RECORD
           MANTISSA: ARRAY[0..2] OF BYTE;
           EXPONENT: BYTE;
          END;

  VAR
   R1,R2,R3: REAL;
   SEMAPHORE: BOOLEAN;

BEGIN
 R1 := R2 - R3 * R1;
  { Compiler generated code for this statement:      }
  {    LDD   #R1                 ;address of R1       }
  {    LDX   #R2                 ;address of R2       }
  {    LDU   #compiler_temporary;address of result    }
  {    LBSR   REAL_MUL                                }
  {    LDD   #R2                                      }
  {    TFR   Y,X   ;result of previous multiplication}
  {    LDU   #R1                                      }
  {    LBSR  REAL_ADD                                 }
  {                                                   }
 IF  -R1<R2 THEN R1 := R2;
  {  Compiler generated code for this statement:      }
  {     LDD  #R1                                      }
  {     LDX  #compiler_temporary                      }
  {     LBSR REAL_NEG                                 }
  {     TFR  Y,D     ;result of negation              }
  {     LDU  #R2                                      }
  {     LBSR REAL_LES                                 }
  {     LBEQ else_label  ;Z flag should be 0 if false}
  {     LDD  #R1                                      }
  {     LDX  #R2                                      }
  {     LDU  #4          ;number bytes in type REAL   }
  {     LBSR MBmove                                   }
  {  else_label                                       }
  {                                                   }
 SEMAPHORE := R1 <= R2;
  {  Compiler generated code for this statement:      }
  {     LDD  #R1                                      }
  {     LDX  #R2                                      }
  {     LBSR REAL_LE                                  }
  {     STB  SEMAPHORE   ;B should be set accordingly}
  {                                                   }
END.
```

Options

OPTIMIZE
Default OFF.

Forward Branches - The 6809 has short branch instructions that can be used when the location to be branched to is within 128 bytes from the branch location. The compiler optimizes all backward branches since it knows the distance to be branched to at compile time. Forward branches, on the other hand, are always assumed to be long (the distance to be branched to is not known). Since most forward branches have been found to be short, an optimization has been added to the 6809 compiler such that when the option $OPTIMIZE$ is ON, the compiler assumes that all forward branches are short. This will cause a compiler error (Pass 3 ERROR-- 1200) if the branch happens to be out of range; if this occurs, $OPTIMIZE$ should be turned OFF around the branch where the error occurs. The error reads: Long range error, turn off OPTIMIZE for this line.

Recursive Parameter Addressing - The 6809 has three different stack offset sizes: 5 bits, 8 bits, and 16 bits. The parameters on a recursive routine are always allocated on the stack and their stack offsets will depend on the number and size of the variables and temporaries used in the routine. Since the stack offset size for the parameters is not known until the end of the routine, all stack offsets for recursive parameters are assumed to be 16 bits long. When the option $OPTIMIZE$ is ON, however, the compiler makes an "educated guess" on the size of the parameter stack offset. This can cause a compiler error if the actual offset size does not match the compilers guess, and the compiler program counter will differ from the actual program counter. The error will appear on the next label following the statement where the parameter is accessed. When this error occurs, turn OPTIMIZE OFF at least for the statement where the parameter is accessed. The error has a rippling effect, so it is recommended that $OPTIMIZE$ be turned OFF for the entire routine. The error reads: Program counters do not agree.

DEBUG
Default OFF.

The DEBUG option is used to check for arithmetic errors on arithmetic operations for the standard types. Operations which may normally be performed with in-line code (such as a BYTE add), will be performed using a subroutine call if the $DEBUG$ option is ON. The library routines in the debug library have checks to detect arithmetic errors. The routines of the same name in the nondebug libraries perform the same arithmetic operation but do not detect any error conditions.

Position Independent Code

Parts of the 6809 compiler are not position independent.

Case-statements have position-dependent code (absolute jump). This can be avoided by writing if-statements instead of case-statements.

Addressing of constants. Some addresses and dope vectors are allocated in a constant data area with the label "CONST_prog". Possible cases that require constants are: the use of the "IN" function, multi-dimensional or integer element array references, pointers, user-defined constants, etc. Compile your program with options expand and check the assembly code for the label "CONST_prog" to verify that the compiler has used the constant area. The constant area is always allocated at the end of the main program.

Addressing of static variables and temporaries. A program with no static variables and all procedures being recursive will not have this problem.

External procedures. Any external procedure requires an address (determined by the linker); this includes the run-time routines.

Pass2 Errors

Pass2 will be displayed on the screen with the message:

```
LINE # <line number>--PASS2 ERROR # <error number>
```

In addition, if a listing file has been indicated for the compilation it will indicate pass2 errors where they occurred. It will also give you a listing of the meaning of each error.

Pass2 error numbers will always be >=1000. Errors with numbers between 1000 and 1099 are fatal errors. Errors with numbers >=1100 are non-fatal errors.

Pass2 will stop generating code after a fatal pass 2 error. If a listing file has been indicated for the compilation, pass 3 will give you a listing with errors. Non-fatal errors are output to the display and to the listing file (if one exists), but compilation continues after appropriate action has been taken to correct the error. A list of pass 2 errors is given in Table 2-1.

Table 2-1. 6809 Pass2 Errors

1000 - "Out of memory"
The 6809 code generator has run out of memory, break up your program and recompile.

1001 - "Size not implemented"
An integer larger than 16 bits has been detected.

1002 - "Size error"
A size larger than the maximum size allowed for a type has been detected.

1003 - "Type not implemented"
A real or other unimplemented type has been detected.

1004 - "Type error"
An operation with an incorrect type of operands has been detected; for example, a negation of an unsigned value.

1005 - "Unimplemented feature"
An attempt has been made at using a feature not implemented on the 6809 code generator.

1006 - "Compiler error. Contact Hewlett-Packard"
This is a compiler level error. Please report this error to Hewlett-Packard as soon as possible.

1007 - "Expression too complicated"
The compiler can not handle the level of complexity of this expression, simplify your expression.

1008 - "Register needed but not available"
The compiler can not generate more code without additional registers; add temporary results for your operations.

1010 - "Too many constants"
More than 256 constant values required during code generation. Eliminate duplicate real constants or break up module and recompile.

Table 2-1. 6809 Pass2 Errors (Cont'd)

1103 - "Interrupt procedure must not have parameters"
An interrupt procedure can not have parameters. The compiler will ignore the parameters and continue to generate code.

1104 - "Interrupt procedure call not allowed"
An interrupt routine can only be accessed through an interrupt vector, since it will return with an RTI instead of an RTS. The compiler will ignore calls to interrupt routines.

1106 - "Program counter overflow"
The program will wrap around OFFFFH. Other errors may occur if this is not corrected.

1107 - "Data counter overflow"
The data counter will wrap around OFFFFH.

1110 - "Defined a static routine within a recursive one"
Static routines can not be defined within recursive routines because of the difference in addressing. The compiler makes the routine recursive and continues to generate code.

1111 - "Interrupt routines must be at level one"
All interrupt routines must be at level one. For routines defined at levels greater than 1 with $INTERRUPT ON$, the compiler will ignore the option, i.e. it will generate a non-interrupt routine.

1113 - "Program counters do not agree"
The program counter for a label generated by Pass 2 does not agree with the program counter for that label in Pass 3. Please report the error to Hewlett-Packard as soon as possible. This error is detected in Pass 3.

1200 - "Long range error; turn off OPTIMIZE for this line"
The option $OPTIMIZE$ causes the code generator to use 2-byte branch instructions for forward branches. This error occurs when the label is too far away. Turning $OPTIMIZE OFF$ for this line of code will produce a long jump which will always work.

Chapter 3

Run-time Library Specifications

General

This chapter describes the run-time library routines needed to execute Pascal programs compiled by the Pascal/64000 compiler for the 6809 microprocessor. Each routine description includes the purpose, input requirements, and output results.

The library is logically divided into two groups of routines. One group contains the standard library procedures and functions. The second group supplies the elementary routines that supplement the standard 6809 instruction set. Tables 3-1 and 3-2 list the standard and supplemental routines for the 6809 microprocessor.

Table 3-1. Pascal Library Routines (Standard)

Name	Purpose	Ref Page
ARRAY_	Compute address of array element	3-6
PARAM_	Pass parameters to procedures	3-14
RENTRY_	Recursive procedure entry	3-11
INITHEAP	Declares block of memory as memory pool	3-10
NEW	Dynamic memory allocation	3-10
DISPOSE	Dynamic memory deallocation	3-10
MARK	Save current status of dynamic memory heap	3-10
RELEASE	Restore prior status of dynamic memory heap	3-10

Table 3-2. Pascal Library Routines (for 6809)

8-bit Arithmetic Group

Name	Purpose	Ref Page
Zbyteabs	Byte absolute value	3-16
Zbyteneg	Byte negation	3-16
Zbyteadd	Byte addition	3-17
Zubyteadd	Unsigned byte addition	3-17
Zbytesub	Byte subtraction	3-17
Zubytesub	Unsigned byte subtraction	3-17
Zbytemul	Byte multiplication	3-17
Zubytemul	Unsigned byte multiplication	3-17
Zbytediv	Byte division	3-17
Zubytediv	Unsigned byte division	3-17

16-Bit Arithmetic Group

Name	Purpose	Ref Page
Zintabs	Integer absolute value	3-18
zintneg	Integer negation	3-18
Zintadd	Integer addition	3-19
Zuintadd	Unsigned integer addition	3-20
Zintsub	Integer subtraction	3-20
Zuintsub	Unsigned integer subtraction	3-20
Zintmul	Integer multiplication	3-20
Zuintmul	Unsigned integer multiplication	3-20
Zintdiv	Integer division	3-19
Zuintdiv	Unsigned integer division	3-20

Table 3-2. Pascal Library Routines (for 6809)(Cont'd)

Byte and Word Shifts

Name	Purpose	Ref Page
Zbshift	Byte shift logical with zero fill	3-22
Zbrotate	Byte shift circular	3-22
Zwshift	Word shift logical with zero fill	3-23
Zwrotate	Word shift circular	3-23

Byte and Word Set Operations

Name	Purpose	Ref Page
Zbinset8	Byte in 8-bit set	3-24
Zbinset16	Byte in 16-bit set	3-25
Zbtoset8	Byte to 8-bit set	3-25
Zbtoset16	Byte to 16-bit set	3-26
Zwinset16	Word in 16-bit set	3-27
Zwtoset16	Word to 16-bit set	3-27

Multi-byte Operations

Name	Purpose	Ref Page
MBmove	Multi-byte assignment	3-28
MBequ	Multi-byte equality test	3-29
MBneq	Multi-byte inequality test	3-29
MBgeq	Multi-byte greater than or equal test	3-29
MBgtr	Multi-byte greater than test	3-29
MBleq	Multi-byte less than or equal test	3-29
MBles	Multi-byte less than test	3-29

Table 3-2. Pascal Library Routines (6809)(Cont'd)

Multi-byte Set Operations

Name	Purpose	Ref Page
INSETmb	Multi-byte set inclusion	3-31
TOSETmb	Multi-byte set formation	3-32
SETmbINT	Multi-byte set intersection	3-32
SETmbUNI	Multi-byte set union	3-32
SETmbDIF	Multi-byte set difference or equal	3-32
SETmbLEQ	Multi-byte set less than or equal	3-33

Byte and Integer Comparison and Bounds Checking Routines

Name	Purpose	Ref Page
Zcc	Carry cleared test	3-35
Zequ	Byte and integer equality test	3-35
Zneq	Byte and integer inequality test	3-35
Zgeq	Byte and integer greater than or equal test	3-35
Zgtr	Byte and integer greater than test	3-35
Zleq	Byte and integer less than or equal test	3-35
Zles	Byte and integer less than test	3-35
Zugeq	Unsigned byte and integer greater than or equal test	3-35
Zugtr	Unsigned byte and integer greater than test	3-35
Zuleq	Unsigned byte and integer less than or equal test	3-35
Zules	Unsigned byte and integer less than test	3-35
Zbbounds	Byte bounds checking	3-36
Zubbounds	Unsigned byte bounds checking	3-36
Zwbounds	Integer bounds checking	3-37
Zuwbounds	Unsigned integer bounds checking	3-37

Table 3-2. Pascal Library Routines (for 6809)(Cont'd)

String Operations

Name	Purpose	Ref Page
STmove	String assignment	3-40
STequ	String equality test	3-38
STneq	String inequality test	3-38
STgeq	String greater than or equal test	3-38
STgtr	String greater than test	3-38
STleq	String less than or equal test	3-38
STles	String less than test	3-38
CHequ	String-char equality test	3-38
CHneq	String-char inequality test	3-38
CHgeq	String-char greater than or equal test	3-38
CHgtr	String-char greater than test	3-38
CHleq	String-char less than or equal test	3-38
CHles	String-char less than test	3-38

Miscellaneous

Name	Purpose
END_DATA_	Label at the end of the library that can be used to allocate the HEAP area.
Z_END_PROGRAM	Label called at the end of the main program.
EMPTY_SET_	The largest possible empty set for the 6809.
STACK_	Label for stack.
CASE_ERROR	Label for case error.

Array Reference Routines

The Pascal/64000 compiler supports generalized array references with up to 10 indices. The array reference routines are called with the parameters:

DOPE_VECTOR	-	address of a record describing the array.
BASE_ADDRESS	-	address of the first element of the array. (May be indirected like a VAR parameter.)
Index_list	-	addresses of the actual index expressions (one for each formal index expression).

The array reference routines return the computed memory address to the X register.

ARRAY_ .

The ARRAY_ routine returns the memory address of an n-dimensional array reference expression. The array reference call for the 3-index array variable expression:

```
A(I,J,7)
```

would be:

```
LDU  BASE_ADDRESS         ; base address of array A
LDY  I
LDX  J
PSHS X,U,Y
LDU  #0007H
PSHS U
LDA  #3                   ; number of indices passed
LDX  DOPE_VECTOR_ADDRESS  ; for array A
LBSR ARRAY_ .
```

To illustrate the use of indirection required for the base address, consider variable B defined as a pointer to an array of the same type as A in the above example. A reference to an element of B^ with the variable array expression:

R^ (6+T,J,7)

would generate a call to ARRAY_ in the form:

```
    LDD  T
    ADDD #0006H
    LDU  [R]                     ; base address of array A
    TFR  D,Y
    LDX  J
    PSHS X,U,Y
    LDU  #0007H
    PSHS U
    LDA  #3                      ; number of indices passed
    LDX  DOPE_VECTOR_ADDRESS     ; for array A
    LBSR ARRAY_ .
```

Pascal defines the ARRAY type recursively as a single dimensioned array of any declarable Pascal type. Thus multi-dimensioned arrays are simply defined as array of arrays. An array may be referred to in its entirety (a so-called ENTIRE variable) by referring to the array by its name using no parameters. A variable expression allows the user to refer to an INDEXED element type by allowing from 1 to N index expressions to be used in an array reference. Pascal arrays are stored such that the rightmost subscript changes faster.

The ARRAY_ call for a two-indexed array variable expression with a 3-dimensional array A is as follows:

A(I,J)

For example:

```
          LDU  BASE_ADDRESS        ; base address of array A
          LDY  I
          LDX  J
          PSHS U,Y,X
          LDA  #2                  ; number of indices passed
          LDA  DOPE_VECTOR_ADDRESS ; for array A
          LBSR ARRAY_ .
```

The formulae for computing array element addresses are as follows:

a. The formula used to compute the array element address is:

```
ADDRESS: BASE_ADDRESS + (-OFFSET_CONSTANT) +
         (I1 * PROD_1) + (I2 * PROD_2) +...+
         (IN * PROD_N)
```

b. The (-OFFSET_CONSTANT) term is the product of the index lower bounds and the row widths, i.e.,

```
(I1L * PROD_1) + (I2L * PROD_2) +...+
(INL * PROD_N)
```

c. The expression used to compute the array row reference using N-1 index expression is:

```
row_address := BASE_ADDRESS + (-OFFSET_CONSTANT)
        + (I1 * PROD_1) +...+
        (InMINUS_1 * InMINUS_1) + ROWnMINUS_1
```

NOTE

The addition of ROWnMINUS_1 takes you to ROWn.

Generalized Array DOPE__VECTOR

The form of the general array reference dope vector is equivalent to:

```
       DOPE_VECTOR        FDB N                   ;number of
                                                  ; dimensions
                          FDB (-OFFSET_CONSTANT) ;negative of
                                                  ; constant
                          FDB PROD_1
                          FDB PROD_2
                          ...
                          FDB PROD_N
                          FDB ROW1
                          FDB ROW2
                          ...
                          FDB ROWnMINUS_1
```

About the Routine:

At termination, this routine returns the stack pointer to the location it held at the beginning of the program.

ARRAY_ jumps to an entrance point in subroutine ARR_ where the address of those formal indices which are given is computed and then it returns to ARRAY_ to add in the ROWn value(s).

The array reference routines return the computed memory address in the X register.

NOTE

Users who write assembly language programs that define and use multi-dimension arrays to be used with the ARRAY_ routine need to ensure that their use is consistent with the Pascal compiler. In order to accomplish this, it is recommended that the user write a simple Pascal program defining and using the arrays. The user can then use the expanded listing file or the ASM_FILE option to determine how the Pascal compiler accesses these arrays and defines the array dope vectors. It is important that the user's array dope vector be identical to that produced by the compiler.

Dynamic Memory Allocation

Pascal/64000 supports dynamic allocation and deallocation of storage space through the procedures NEW, DISPOSE, MARK, RELEASE, and INITHEAP.

INITHEAP

The user declares a block of memory to be used as the memory pool or heap by calling: INITHEAP (Start_address, Length_in_bytes : INTEGER). The procedure, INITHEAP, must be declared EXTERNAL in the declaration block of a program. The resultant heap will be six bytes smaller than length_in_bytes.

NEW

The procedure NEW (Pointer : Pointer_to_type) is used to allocate space. The procedure, NEW, searches for available space in a free-list of ascending size blocks. When a block is found that is the proper size or larger, it is allocated and any space left over is returned to the free-list in a new place corresponding to the size of the leftover block. If the referenced block is four or less bytes in size, four bytes will be allocated.

DISPOSE

The procedure DISPOSE is exactly the reverse of the procedure NEW. It indicates that storage occuppied by the indicated variable is no longer required.

MARK

This procedure marks the state of the heap in the designated variable that may be of any pointer type. The variable must not be subsequently altered by assignment.

RELEASE

The procedure RELEASE restores the state of the heap to the value in the indicated variable. This will have the effect of disposing all heap objects created by the NEW procedure since the variable was marked. The variable must contain a value returned by a previous call to MARK; this value may not have been passed previously as a parameter to RELEASE.

Recursive Entry

Pascal/64000 supports recursive and reentrant calling sequences for procedures compiled for the 6809 with the $RECURSIVE ON$ option by additional run-time entry code. This code causes the local data area of a procedure to be allocated onto the stack before entry to the procedure and to be deallocated from the stack upon exit from the procedure. These functions are performed by the procedure RENTRY_.

RENTRY_ .

RENTRY_ is called at the entry point of a procedure or function which has been declared with the option $RECURSIVE ON$. RENTRY_ will copy the parameters, set the static links, and allocate the variable size area.

RENTRY_ is called upon entry to a recursive Pascal procedure or function. The calling sequence is:

```
LDU   var_area_size
LDA   #level                      ;always <=16
LDX   par_area_size               ;could be zero
LDB   #register_par_flag          ; true (1) or
                                  ; false (0)
```

The stack format at entry to RENTRY_:

NOTE

The stack entry at entry to RENTRY_ will vary slightly according to how the parameters are passed.

```
 _________________________
|                         |
|        Garbage          |
|_________________________|
|                         |
|        Par. #1          |
|_________________________|
|           .             |
|           :             |
|           .             |
|_________________________|
|                         |
|        Par. #n          |
|_________________________|
|                         |
|  RA (calling routine)   |
|_________________________|
|     RA from RENTRY_     |
|_________________________|  <---------- <S>
```

Procedure:

a. Copy the parameters from register S to (Y).

b. Copy the calling routine's RA to "Garbage".

c. Allocate var_size_area.

The stack format at exit from RENTRY_ is as follows:

```
 _________________________
|                         |
|  RA (calling routine)   |
|_________________________|
|                         |
|                  ^      |
|       Par.'s     |      |
|_________________________|
|                         |
|                  ^      |
|       Var.'s     |      |
|_________________________|
|                         |
|      Static Links       |
|_________________________|  <------------(S)
```

All registers but CC are modified.

Parameter Passing

General

The procedure head is extended by a parameter list in which the formal parameters are declared. The VAR in the parameter list indicates that the values of the parameters may be changed within the body of the procedure. An example of a program used for parameter passing is as follows:

```
PROCEDURE PROCA (VAR I: BYTE; VALUEP: BYTE);
  BEGIN
    I := I+VALUEP;
  END;
```

The 6809 compiler knows whether it has to pass the address or the value of the parameters; thus, no dope vector is necessary. It is recommended that, unless otherwise necessary, pass by reference parameters are used for parameters of a structured type. For example:

```
PROGRAM TEST;

TYPE ARR1TO10: ARRAY[1..10] OF INTEGER;
VAR  AA:  ARR1TO10

    PROCEDURE ONE(VAR A: ARR1TO10);  {pass by refer-}
                                     {ence parameter}
     BEGIN
        .
        .
     END;

    PROCEDURE TWO(A: ARR1TO10)       {pass by value }
                                     {parameter     }
     BEGIN
        .
        .
     END;

BEGIN
 ONE(AA);    {this call will pass two bytes of       }
             {parameter data, i.e., the address of AA}
 TWO(AA);    {this call will pass 20 bytes of        }
             {parameter data, i.e., the 10 values of }
             {AA                                     }
END.
```

Parameter Passer (PARAM_)

The parameter passer is called from a static routine receiving parameters where the parameters are not passed in registers. PARAM_ transfers the parameters from the stack to the called routine's static data area.

If the total par-area-size in a static routine is less or equal to eight bytes and there are not more than two parameters of size equal to one byte, then the parameters will be passed in registers (instead of on the stack), else all the parameters will be pushed on the stack.

A receiving routine where parameters are passed in the registers has to store or push (for recursive routines) every parameter in the receiving routine's data area.

Calling sequence (from the routine passing the parameters, to the routine receiving the parameters) for parameters not passed in registers:

a. Calling a recursive routine:

```
LDr1   par#1
 .
 .
 .
LDrX   par#x
PSHS   r1,...,rx,PC
LDr1   par#x+1
 .
 .
 .
LDrx   par#n
PSHS   r1,...,rx
LBSR   receiving_routine
```

b. Calling a static routine:

```
LDr1   par#1
 .
 .
 .
LDrx   par#x
PSHS   r1,...,rx
LDr1   par#x+1
 .
 .
 .
LDrx   par#n
PSHS   r1,...,rx
LBSR   receiving_routine
```

Calling sequence (from a static routine receiving the parameters, to PARAM_):

```
        LDX     to_address
        LDD     par_area_size
        LBSR    PARAM_ .
```

Stack format at entry to PARAM_:

```
 _________________________
|                         |
|         Par. #1         |
|_________________________|
|            .            |
|            .            |
|            .            |
|_________________________|
|                         |
|         Par. #n         |
|_________________________|
|                         |
|  RA (calling routine)   |
|_________________________|
|                         |
| RA (receiving routine)  |
|_________________________| <-------(S)
```

Stack format at exit from PARAM_:

```
 __________________________
|                          |
|  RA (calling routine)    |
|__________________________| <---------(S)
```

NOTE

If you write assembly language programs that define and use procedures and functions, particularly with parameters, be sure their use is consistent with the Pascal compiler. HP recommends writing a simple Pascal program defining the procedure or function with the desired parameter list and an empty BEGIN END block for code. Then use the expanded listing file or ASM_FILE option to determine how the Pascal compiler enters and exits the equivalent do-nothing procedure and how the parameters are passed. Your assembly language routines must follow the same entry, parameter passing, and exit code produced by the compiler. It is important that recursive or static mode declarations (and use) be consistent.

Standard Byte Routines

For standard byte routines, parameter values are passed using specific registers. The operands are 8-bit signed or unsigned bytes. There are two groups of byte operations: the unary byte operation, and the binary byte operation. These operations are discussed in the following paragraphs.

Unary Byte Operations

Zbyteabs	Byte absolute value
Zbyteneg	Byte negation

The unary byte operation is of the form:

```
RESULT := op B1
```

where:

B1 is loaded in register B

The library routine is called after loading B1 into the B register. The byte RESULT is returned in the B register.

```
Register Allocation Summary :: Unary byte operations

Input:  B contains value to be operated on

Output: B contains byte RESULT

Registers:
    Modified:  B
    Unchanged: A,X,Y,U,S,CC
```

Binary Byte Operations

Zbyteadd	Byte addition
Zubyteadd	Unsigned byte addition
Zbytesub	Byte subtraction
Zubytesub	Unsigned byte subtraction
Zbytemul	Byte multiplication
Zubytemul	Unsigned byte multiplication
Zbytediv	Byte division
Zubytediv	Unsigned byte division

a. **Zbyteadd** performs the addition of two bytes.

b. **Zbytediv** performs the division of two bytes using the following algorithm:

   (1) Shift divisor left to its highest possible value.

   (2) Subtract divisor from dividend.

   (3) If result is positive, put 1 in rightmost digit of quotient. If negative, add divisor back into dividend and put 0 in quotient.

   (4) Shift divisor right and repeat steps 2 and 3 until divisor returns to its original value. The result of the division is available in register B upon completion. The remainder is also available in register A (used for MODULUS).

c. **Zbytemul** performs the multiplication of two bytes. The actual multiplication works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result to obtain the correct result. If both operands are positive or negative, the positive dummy result is the actual result. The eight LSB of the result of the multiplication are available in register B upon completion of the routine and the eight MSB are in register A.

d. **Zbytesub** performs the subtraction of two bytes.

e. **Zubyteadd** performs the addition of two unsigned bytes.

f. **Zubytediv** performs the division of two unsigned bytes. The binary division algorithm is as that of b above. The result of the division is available in register B upon completion. The remainder is also available in register A (used for MODULUS).

g. **Zubytemul** performs the multiplication of two unsigned bytes.

h. **Zubytesub** performs the subtraction of two unsigned bytes.

```
Register Allocation Summary :: Binary 8BIT ops.

  Input:  B contains the first operand
          A contains the second operand

  Output: B contains the result
          A contains the MSB of result - MUL
            contains the remainder     - DIV

   Registers:
      Modified:  A,B
      Unchanged: X,Y,U,S,CC
```

Standard Integer Routines

The integer operations require 16-bit operands. The two 8-bit accumulators are normally used as a 16-bit register (called D) for these routines. As a register pair, the high-order byte is always stored in register A and the low-order byte is stored in register B. Register X is a 16-bit register and is used for binary operations and for returning some results. There are two groups of integer operations: the unary integer operation and the binary integer operation. These operations are discussed in the following paragraphs.

Unary Integer Operations

Zintabs	Integer absolute value
Zintneg	Integer negation

The unary integer operation is of the form:

RESULT := op I1

where:

I1 is loaded in register pair D.

The library routine is called after loading I1 into register D. The integer RESULT is returned in register D.

```
 ______________________________________________________________
|                                                              |
|  Register Allocation Summary :: Unary integer operations     |
|______________________________________________________________|
|                                                              |
|  Input:  D contains integer value to be operated on          |
|                                                              |
|  Output: D contains integer RESULT                           |
|                                                              |
|  Registers:                                                  |
|      Modified:  D                                            |
|      Unchanged: X,Y,U,S,CC                                    |
|______________________________________________________________|
```

Binary Integer Operations

Zintadd	Integer addition
Zuintadd	Unsigned integer addition
Zintsub	Integer subtraction
Zuintsub	Unsigned integer subtraction
Zintmul	Integer multiplication
Zuintmul	Unsigned integer multiplication
Zintdiv	Integer division
Zuintdiv	Unsigned integer division

a. **Zintadd** performs the addition of two integers.

b. **Zintdiv** performs the division of two integers. The actual division works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result. If both operands are positive or negative, the dummy result is the correct answer. The division algorithm is as follows:

   (1) Shift divisor left to its highest possible value.

   (2) Subtract divisor from dividend.

   (3) If result is positive, put 1 in rightmost digit of quotient. If negative, recover divisor before subtraction and put 0 in quotient.

   (4) Shift divisor right and repeat steps b and c until divisor returns to its original value.

The result of the division is available in register D upon completion. The remainder is available in register X (used for MODULUS).

c. **Zintmul** performs the multiplication of two integers. The actual multiplication works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result. If both operands are positive or negative, the positive dummy result is the correct result. The multiplication occurs as follows:

```
(A:B) * (C:D)  =           BDH : BDL
               +       BCH: BCL
               +       ADH: ADL
               +  ACH: ACL
```

The lower 16 bits of the result are placed into the D register and the 16 most significant bits of the result are placed in register X upon completion of the library routine.

d. **Zintsub** performs the subtraction of two integers.

e. **Zuintadd** performs the addition of two unsigned integers.

f. **Zuintdiv** performs the division of two unsigned integers. The binary division algorithm is as that in b above. The result of the division is available in register D upon completion. The remainder is available in register X (for MOD).

g. **Zuintmul** performs the multiplication of two unsigned integers. The actual multiplication occurs as explained in c above. The lower 16 bits of the result are placed into register D and the 16 most significant bits of the result are placed in register X upon completion of the library routine.

h. **Zuintsub** performs the subtraction of two unsigned integers.

```
 ____________________________________________________
|                                                    |
|   Register Allocation Summary :: Binary 16BIT ops. |
|____________________________________________________|
|       Input:  X contains the first operand         |
|               D contains the second operand        |
|       Output: D contains the result                |
|               X contains the MSW of result - MUL   |
|                 contains the remainder     - DIV   |
|                                                    |
|        Registers:                                  |
|           Modified:  D,X                           |
|           Unchanged: Y,U,S,CC                      |
|____________________________________________________|
```

Byte and Word Shifts

Pascal/64000 supports logical and circular shifting of both byte (8-bit) and word (16-bit) quantities using the predefined functions SHIFT and ROTATE. These functions are available when compiling with the $EXTENSIONS ON$ option of the compiler. The DIV operator using powers of 2 may be used to accomplish an arithmetic right shift (i.e., with sign extension). For example, X DIV 2 is equivalent to a one bit right shift with sign extension.

SHIFT

Logical shifting with zero fill will shift the quantity left or right placing a zero in the most (right shift) or least (left shift) significant bit for each shift. The function is called with two parameters: the quantity to be shifted and the number of bit positions to shift. The function call in Pascal is of the form:

```
variable := SHIFT(expression,n);
```

where:

expression	is any expression, variable or constant
n	is the number of bits to be shifted

where:

n>0	results in a left shift
n<0	results in a right shift

ROTATE

Circular shifting rotates the quantity left or right and fills the vacated position with the bit shifted out of the other end. The function is called with two parameters: the quantity to be shifted and the number of bit positions to shift. The function call in Pascal is of the form:

```
variable := ROTATE(expression,n);
```

where:

expression	is any expression, variable or constant
n	is the number of bits to be shifted

where:

n>0	results in a left circular shift
n<0	results in a right circular shift

Pascal/64000 determines the size (1 or 2 bytes) of the data being shifted by the type of the first parameter expression. The type of result returned by the function SHIFT or ROTATE is the same type as the type of the first parameter expression.

Byte Shifts

Zbshift	Byte shift logical with zero fill
Zbrotate	Byte shift circular

The byte shift operations compute the byte result of shift expressions of the form:

```
        RESULT := SHIFT(B1,B2);

      or

        RESULT := ROTATE(B1,B2);
```

where:

B1 is loaded in register A
B2 is loaded in register B

The library routine is called after loading B1 into register A and B2 into register B. The byte RESULT is returned in register B.

```
 ______________________________________________________________
|                                                              |
|  Register Allocation Summary :: Byte shift operations        |
|______________________________________________________________|
|                                                              |
|   Input:  A contains byte to be shifted, B1                  |
|           B contains number of positions to shift, B2        |
|                                                              |
|   Output: B contains byte RESULT                             |
|                                                              |
|   Registers:                                                 |
|       Modified:  B,CC                                        |
|       Unchanged: A,X,Y,U,S,DP                                |
|______________________________________________________________|
```

Word Shifts

Zwshift	Word shift logical with zero fill
Zwrotate	Word shift circular

The word shift operations compute the word result of shift expressions of the form:

```
RESULT := SHIFT(I1,I2);
```

or

```
RESULT := ROTATE(I1,I2);
```

where:

I1 is loaded in register X
I2 is loaded in register B

The library routine is called after loading I1 into register X and I2 into register B. The word RESULT is returned in register D.

```
 Register Allocation Summary :: Integer shift operations

Input:  X contains word to be shifted, I1
        B contains the number of positions to shift, I2

Output: D contains word RESULT

Registers:
    Modified:  D
    Unchanged: X,Y,U,S,DP
```

Byte and Word Set Operations

Byte Set Operations

Zbinset8	Byte in 8-bit set
Zbtoset8	Byte to 8-bit set

Zbinset8. This routine is used to test the set membership of a byte value in a specified byte set. For example, the Pascal/64000 expression:

```
R  IN  SET8
```

is a Boolean expression whose value is TRUE if bit R of SET8 is set and FALSE if bit R of SET8 is reset.

```
Register Allocation Summary :: Zbinset8

  Input:  B contains the byte set being compared
          A contains byte value to be tested

  Output: B set to 0, Z flag set if value not in set
          B set to 1, Z flag reset if value in set

  Registers:
      Modified:  B,CC
      Unchanged: A,X,Y,U,S,DP

At termination, register A will contain the byteset
 which was compared.
```

Zbtoset8. This routine converts a byte into an 8-bit set. The only valid input values are 0 through 7. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results. The Pascal statements:

```
   Q := BIT_0;
SET8 := [Q]
```

will assign to SET8 a byte with the least significant bit set and all the others reset.

```
 ________________________________________________
|                                                |
|    Register Allocation Summary :: Zbtoset8     |
|________________________________________________|
|                                                |
|   Input:  B contains byte value to be converted |
|                                                |
|   Output: B contains the byteset result        |
|                                                |
|    Registers:                                  |
|       Modified:  B                             |
|       Unchanged: A,X,Y,U,S,CC                  |
|________________________________________________|
```

Word Set Operations

Zbinset16	Byte in 16-bit set
Zbtoset16	Byte to 16-bit set
Zwinset16	Word in 16-bit set
Zwtoset16	Word to 16-bit set

Zbinset16. This routine is used to test the set membership of a byte value in a specified word set. For example, the Pascal/64000 expression:

```
V IN SET16
```

is a Boolean expression whose value is TRUE if bit V of SET16 is set and FALSE if bit V of SET16 is reset.

```
     Register Allocation Summary :: Zbinset16

    Input:  B contains byte value to be tested
            X contains the word set being compared

    Output: B set to 0, Z flag set if value not in set
            B set to 1, Z flag reset if value in set

    Registers:
        Modified:  B,CC
        Unchanged: A,X,Y,U,S,DP

 At termination, register X will contain the word
  set compared.
```

Zbtoset16. This routine converts a byte into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results. The Pascal statements:

```
    V := BIT_15;
SET16 := [V];
```

will assign to SET16 a word with the most significant bit of the second byte of SET16 set and all the others reset.

```
   Register Allocation Summary  ::  Zbtoset16

     Input:  B contains byte value to be converted

     Output: D contains the wordset result

       Registers:
           Modified:  D
           Unchanged: X,Y,U,S,CC
```

Zwinset16. This routine is used to test the set membership of a word value in a specified word set. For example, the Pascal/64000 expression:

W IN SET16

is a Boolean expression whose value is TRUE if bit W of SET16 is set and FALSE if bit W of SET16 is reset.

```
 ________________________________________________________
|                                                        |
|     Register Allocation Summary :: Zwinset16           |
|________________________________________________________|
|                                                        |
|    Input:  D contains word value to be tested          |
|            X contains the word set being compared      |
|                                                        |
|    Output: B set to 0, Z flag set if value not in set  |
|            B set to 1, Z flag reset if value in set    |
|                                                        |
|       Registers:                                       |
|           Modified:  D,CC                              |
|           Unchanged: X,Y,U,S,DP                        |
|________________________________________________________|
```

Zwtoset16. This routine converts a word into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results.

```
 ________________________________________________________
|                                                        |
|     Register Allocation Summary  ::  Zwtoset16         |
|________________________________________________________|
|                                                        |
|       Input:  D contains word value to be converted    |
|       Output: D contains the wordset result            |
|                                                        |
|         Registers:                                     |
|             Modified:  D                               |
|             Unchanged: X,Y,U,S,CC                      |
|________________________________________________________|
```

Multi-byte Operations

The multi-byte routines are used by the compiler to operate on multi-byte records (or arrays) of the same type.

MBmove

The routine MBmove is used for moving multi_byte records such as in an assignment of a complete record type or an array type to another of the same type. The Pascal statements:

```
     VAR A1,A2:ARRAY[0..LENGTH] OF BYTE;
     BEGIN
        .
        .
        .
        A1:=A2;
```

Will cause a call to MBmove.

```
 ________________________________________________
|                                                |
|   Register Allocation Summary :: MBmove        |
|________________________________________________|
|                                                |
|  Input: X contains the first record's addr.    |
|         U contains the second records's addr.  |
|         D contains the number of bytes in      |
|           the records.                         |
|                                                |
|  Registers:                                    |
|      Modified : D                              |
|      Unchanged: X,Y,U,S,CC                     |
|________________________________________________|
```

Multi-byte Comparisons

MBequ	Multi-byte equality test
MBneq	Multi-byte inequality test
MBgeq	Multi-byte greater than or equal test
MBgtr	Multi-byte greater than test
MBleq	Multi-byte less than or equal test
MBles	Multi-byte less than test

MBequ. This routine is used by the compiler to test multi-byte records of the same type for equality.

MBneq. This routine is used by the compiler to test multi-byte records of the same type for inequality.

MBgeq. This routine is used by the compiler to test if one set of records is greater than or equal to another set of records of the same type. The test is unsigned.

MBgtr. This routine is used by the compiler to test if one set of multi-byte records is greater than another set of the same type. The test is unsigned.

MBleq. This routine is used by the compiler to test if one set of records is less than or equal to another set of records of the same type. The test is unsigned.

MBles. This routine is used by the compiler to test if one set of records is less than another set of records of the same type. The test is unsigned.

The records might be defined in the PASCAL source program by:

```
TYPE
  PERSON=RECORD
           NAME : ARRAY[1..LENGTH] OF CHAR
           ADDRESS : ARRAY[1..LENGTH] OF CHAR
         END;
VAR
  SALESPERSON,TOP_SALESPERSON:PERSON;
```

In use of the variables, one might question equality such as: In use of the variables, one might question equality such as:

```
(MBequ)       IF SALESPERSON  = TOP_SALESPERSON THEN...
(MBneq)       IF SALESPERSON <> TOP_SALESPERSON THEN...
```

A compare routine is called to compare the bytes and upon re-entry to this program, a branch is taken to either a "true" or "false" routine.

PASCAL/64000 does not accept <=, <, >= or > comparisons for arrays or records, therefore the 6809 code generator will never generate calls to MBleq, MBles, MBgeq and MBgtr. These routines have been included in the library for consistency and are available to the user.

```
 ________________________________________________
|                                                |
|   Register Allocation Summary :: Multi-bytes   |
|________________________________________________|
|                                                |
|  Input: X contains the first record's addr.    |
|         U contains the second records's addr.  |
|         D contains the number of bytes in      |
|           the records.                         |
|                                                |
|  Registers:                                    |
|      Modified : D                              |
|      Unchanged: X,Y,U,S,CC                     |
|________________________________________________|
```

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Additionally, register A will contain the byte within the first set of bytes which caused the equality comparison to fail or was the last equal byte to be compared.

Multi-byte Set Operations

Pascal/64000 supports 8-bit and 16-bit sets as well as larger sets with up to 256 elements. These larger sets, requiring three or more bytes, are referred to as multi-byte sets.

Multi-byte Set Routines

INSETmb	Multi-byte set inclusion
TOSETmb	Multi-byte set formation
SETmbINT	Multi-byte set intersection
SETmbUNI	Multi-byte set union
SETmbDIF	Multi-byte set difference
SETmbLEQ	Multi-byte subset inclusion

INSETmb. This routine is used to test the set membership of an integer value in a multi-byte set. For example, the Pascal/64000 expression:

```
V IN SETmb
```

is a boolean expression whose value is TRUE if bit V of SETmb is set and FALSE if bit V of SETmb is reset.

```
 ___________________________________________________
|                                                   |
|    Register Allocation Summary  ::  INSETmb       |
|___________________________________________________|
|                                                   |
|     Input : X contains address of the multi-byte  |
|               set                                 |
|             D contains the integer value V        |
|                                                   |
|     Output: IF V is contained in set              |
|              THEN                                 |
|                B set to 1 (TRUE), Z flag reset    |
|              ELSE                                 |
|                B set to 0 (FALSE), Z flag set     |
|                                                   |
|     Registers:                                    |
|         Modified : D,CC                           |
|         Unchanged: X,Y,U,S,DP                     |
|___________________________________________________|
```

TOSETmb. This routine is used to convert a value into a multi-byte set. For example, the Pascal/64000 statements:

```
    Q := BIT_0;
SETmb := [Q];
```

will assign to SETmb a set with the least significant bit set and all others off.

```
 ______________________________________________
|                                              |
|  Register Allocation Summary :: TOSETmb      |
|______________________________________________|
|                                              |
| Input:  X contains byte value to be converted|
|         U contains the addr. of the result set|
|         D contains the number of bytes in the|
|           set                                |
|                                              |
|  Registers:                                  |
|     Modified:  D                             |
|     Unchanged: X,Y,U,S,CC                    |
|______________________________________________|
```

SETmbINT. This routine is used to compute the set intersection of two multi-byte sets.

SETmbUNI. This routine is used to compute the set union of two multi-byte sets.

SETmbDIF. This routine is used to compute the set difference of two multi-byte sets. The set difference is a set containing all the elements of the multi-byte set in (X) which are not contained in the multi-byte set in (U).

```
Register Allocation Summary :: Big sets

  Input : X contains the first set's adr.
          U contains the second set's adr.
          Y contains the result set's adr.
          D contains the number of bytes

  Registers:
      Modified : D
      Unchanged: X,Y,U,S,CC
```

SETmbLEQ. This routine is used to compute the set inclusion of two multi-byte sets. For example, the Pascal/64000 expression:

```
LARGE_SET[0,7,63] <= S1
```

is a boolean expression whose value is TRUE if bits 0, 7, and 63 of S1 are all set; otherwise the value is FALSE. This is equivalent to asking if the set with bits 0, 7, and 63 set is a subset of S1.

For expressions of the form:

```
S1 >= S2
```

the boolean result indicates whether S2 is a proper subset of S1.

```
Register Allocation Summary  ::  SETmbLEQ

  Input : X contains address of S1
          U contains address of S2
          D contains the number of bytes

  Output: IF S2 is a subset of S1
           THEN
             B set to 1 (TRUE), Z flag reset
           ELSE
             B set to 0 (FALSE), Z flag set

  Registers:
      Modified : D,CC
      Unchanged: X,Y,U,S
```

The Pascal/64000 expression:

```
S1 >= LARGE_SET[0,7,63]
```

is a boolean expression whose value is TRUE if bits 0, 7, 63 of S1 are all set; otherwise, the value is FALSE. This is equivalent to asking if the set with bits 0,7,63 set is a subset of S1.

For expressions of the form: S1 >= S2

the boolean result indicates whether S2 is a proper subset of S1.

To accomplish this operation, the 6809 compiler will invert the operands and proceed to call SETmbLEQ.

Byte and Integer Comparison and Bounds Checking Routines

The comparison (=,<>,>=,>,<=,<) of byte and integer variables produces a Boolean result (FALSE or TRUE) based on the signed or unsigned sequences of byte or word scalar types. In many cases where the comparison is being used as the condition for an IF, REPEAT, or WHILE statement, a branch is taken based on the result of the comparison. However, if the Boolean result is being assigned to a variable or if the expression has multiple comparisons (using AND and OR) an actual Boolean result is required. The byte and word comparison subroutines are used specifically in these situations were the Boolean result is necessary for further computations.

When the $RANGE ON$ option is enabled, all assignment statements and parameter passing of byte and word variables are checked to assure that they are within the bounds of the declared type. The range checking routines for byte and word variables are also described in this section.

Byte and Word Comparisons

Zcc	Carry clear test
Zequ	Byte and integer equality test
Zneq	Byte and integer inequality test
Zgeq	Byte and integer greater than or equal test
Zgtr	Byte and integer greater than test
Zleq	Byte and integer less than or equal test
Zles	Byte and integer less than test
Zugeq	Unsigned byte and integer greater than or equal test
Zugtr	Unsigned byte and integer greater than test
Zuleq	Unsigned byte and integer less than or equal test
Zules	Unsigned byte and integer less than test

Library subroutines are called when the Boolean result is required of a comparison expression of the form:

I1 .op. I2

Zcc is called to test if the carry bit is cleared.

Zequ is called to test for equality between I1 and I2 by testing if the Z flag of the condition codes is set.

Zneq is called to test for inequality between I1 and I2 by testing if the Z bit of the condition codes is set.

Zgeq is called to test if I1 is greater than or equal to I2 by testing if either, but not both, of the N or V bits of the condition codes is set.

Zgtr is called to test if I1 is greater than I2 by testing if the "EXCLUSIVE OR " of the N and V bits is 1 or Z=1.

Zleq is called to test if I1 is less than or equal to I2 by testing if the "EXCLUSIVE OR" of the N and V bits is 1 or Z=1.

Zles is called to test if either, but not both, of the N or V bits is set.

Zugeq is called to test if I1 is less than I2 by testing if the C flag of the condition codes is set.

Zugtr is called to test if I1 is greater than I2 by testing if the previous operation caused either a carry or a zero result.

Zuleq is called to test if I1 is less than or equal to I2 by testing if the previous operation caused either a carry or a zero result.

Zules is called to test if the C bit is set or not.

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Byte Bounds Checking

Zbbounds	Byte bounds checking
Zubbounds	Unsigned byte bounds checking

The bounds checking for signed and unsigned byte variables use the same calling sequence and return the same results. The value being checked is loaded into register X. The upper limit is loaded into register A and the lower limit is loaded into register B. Upon return, register B contains the Boolean result (FALSE or TRUE) of the bounds check and the Z flag will be set according to the Boolean value in B. If B=FALSE (0) then Z is set. If B=TRUE (1) then Z is reset.

The logic of the routine is:

```
IF  UL <= V <= LL  THEN true_result
                   ELSE false_result
```

```
 ________________________________________________
|                                                |
|  Register Allocation Summary :: Byte bounds check  |
|________________________________________________|
|                                                |
| Input:  B          contains the value V        |
|         MSB of X   contains the lower limit (LL) |
|         LSB of X   contains the upper limit (UL) |
|                                                |
| Output: B set to 0, Z flag set if value not in range|
|         B set to 1, Z flag reset if value in range |
|                                                |
| Registers:                                     |
|    Modified:  B,CC                             |
|    Unchanged: X,Y,U,S                          |
|________________________________________________|
```

Word Bounds Checking

Zwbounds	Integer bounds checking
Zuwbounds	Unsigned integer bounds checking

The bounds checking for signed and unsigned word variables use the same calling sequence and return the same results. The value being checked is loaded into register X. The upper limit is loaded into register D and the lower limit is loaded into register U. Upon return, register B contains the Boolean results (FALSE or TRUE) of the bounds check and the Z flag will be set according to the Boolean value of register B. If B=FALSE then Z is set. If B=TRUE than Z is reset.

The logic of the routine is:

```
IF  UL <= V <= LL    THEN true_result
                     ELSE false_result
```

```
 ______________________________________________________
|                                                      |
|   Register Allocation Summary :: Word bounds check   |
|______________________________________________________|
|                                                      |
| Input:  D contains the upper limit (UL)              |
|         U contains the lower limit (LL)              |
|         X contains the value V                       |
|                                                      |
| Output: B set to 0, Z flag set if value not in range |
|         B set to 1, Z flag reset if value in range   |
|                                                      |
| Registers:                                           |
|     Modified:  B,CC                                  |
|     Unchanged: X,Y,U,S                               |
|______________________________________________________|
```

Strings and Characters

The routines **STequ, STneq, STgeq, STgtr, STleq, STles** are used by the compiler to test strings equality or inequality.

The routines **CHequ, CHneq, CHgeq, CHgtr, CHleq, CHles** are used by the compiler to test strings .vs. characters equality or inequality. The character is always the first argument (the compiler will invert the relational operand if necessary, i.e. ST <= CH becomes CH > ST). This routines will set up their arguments and then call the string routines.

A compare routine is called to compare the bytes and upon re-entry to this program, a branch is taken to either a "true" or "false" routine.

String equality and inequality in the PASCAL compiler are determined by the following rules:

a. Two strings are equal IF their lengths are equal and they are equal character by character.

b. The inequality of two strings is determined by the first character by which they differ and if all characters are equal then the longest string is the largest.

```
 _____________________________________________________
|                                                     |
|   Register Allocation Summary: String routines      |
|_____________________________________________________|
|                                                     |
|    Input : X contains the first string's addr.      |
|            U contains the second string's addr.     |
|            D contains the number of bytes in        |
|              the string's type                      |
|                                                     |
|    Registers:                                       |
|        Modified : D                                 |
|        Unchanged: X,Y,U,S,CC                        |
|_____________________________________________________|
```

```
 _________________________________________________________
|                                                         |
|   Register Allocation Summary: Character routines       |
|_________________________________________________________|
|    Input : B contains the character (first operand)     |
|            U contains the string's address              |
|            D contains the number of bytes in the        |
|              string's type                              |
|                                                         |
|    Registers:                                           |
|        Modified : D                                     |
|        Unchanged: X,Y,U,S,CC                            |
|_________________________________________________________|
```

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Additionally, register A will contain the byte within the first set of bytes which caused the equality comparison to fail or was the last equal byte to be compared.

STmove

The routine STmove is used to copy a string from one location to another. The Pascal statements:

```
VAR ST1: PACKED ARRAY[1..n1] OF CHAR;
      { ST1[0] contains the run-time length of ST1 }
    ST2: PACKED ARRAY[1..n2] OF CHAR;
      { ST2[0] contains the run-time length of ST2 }
BEGIN
    .
    .
    .
   ST1:=ST2;
```

Will cause a call to STmove.

The string assignment is determined by the following rules:

```
if  ST2[0] >  n1  then  run-time-error;

if  ST2[0] <= n1  then  assign ST2[0]+1 bytes into ST1,
                         starting at ST2[0];
```

```
 ______________________________________________________
|                                                      |
|       Register Allocation Summary :: STmove          |
|______________________________________________________|
|                                                      |
|      Input: X contains the first string's addr.      |
|             U contains the second string's addr.     |
|             D contains the number of bytes in the    |
|               first string's type (n1+1)             |
|                                                      |
|      Registers:                                      |
|          Modified : D                                |
|          Unchanged: X,Y,U,S,CC                       |
|______________________________________________________|
```

Chapter 4

Real Number Library

Introduction

The Pascal/64000 implementation of the IEEE floating point standard for the 6809 microprocessor is supported by real library RealLIB:C6809 (for Pascal data types: LONGREAL and REAL).

The user interface to these libraries is similar to that described for "User Defined Operators" (see Chapter 2). Each library routine name is a global symbol composed of the symbol REAL or LONGREAL followed by the operation mnemonic (such as REAL_ADD or LONGREAL_MUL). where op is the mnemonic for one of the supported operations. Since the compiler performs some automatic type conversions, there are some additional operations to convert between INTEGER, REAL and LONGREAL data types. Each of the library routines is defined by the equivalent Pascal procedure heading for its declaration.

Table 4-1 summarizes the floating point routines supported by the Pascal/64000 real number libraries. The text describes in more detail the external calling sequence used by the 6809 code generator to invoke these routines. For each routine the Pascal procedure or function heading is given which describes the logical interface for passing parameters and receiving results.

Table 4-1. Pascal Real Number Library Routines

Name	Purpose
REAL_ADD	Real addition
REAL_SUB	Real subtraction
REAL_MUL	Real multiplication
REAL_DIV	Real division
REAL_ABS	Real absolute value
REAL_NEG	Real negation
REAL_SQRT	Real square root
REAL_EXP	Real exponentiation(e to the X)
REAL_LN	Real natural logarithm
REAL_SIN	Real sine
REAL_COS	Real cosine
REAL_ATAN	Real arctangent
REAL_EQU	Real equality test
REAL_NEQ	Real inequality test
REAL_LES	Real less than test
REAL_GTR	Real greater than test
REAL_LEQ	Real less than or equal test
REAL_GEQ	Real greater than or equal test
REAL_FLOAT	Integer to real conversion
REAL_ROUND	Real to integer conversion with rounding
REAL_TRUNC	Real to integer conversion with truncation
LONGREAL_ADD	Longreal addition
LONGREAL_SUB	Longreal subtraction
LONGREAL_MUL	Longreal multiplication
LONGREAL_DIV	Longreal division
LONGREAL_ABS	Longreal absolute value
LONGREAL_NEG	Longreal negation
LONGREAL_SQRT	Longreal square root
LONGREAL_EXP	Longreal exponentiation(e to the X)
LONGREAL_LN	Longreal natural logarithm
LONGREAL_SIN	Longreal sine
LONGREAL_COS	Longreal cosine
LONGREAL_ATAN	Longreal arctangent
LONGREAL_EQU	Longreal equality test
LONGREAL_NEQ	Longreal inequality test
LONGREAL_LES	Longreal less than test
LONGREAL_GTR	Longreal greater than test
LONGREAL_LEQ	Longreal less than or equal test
LONGREAL_GEQ	Longreal greater than or equal test
LONGREAL_FLOAT	Integer to longreal conversion
LONGREAL_ROUND	Longreal to integer conversion with rounding
LONGREAL_TRUNC	Longreal to intgr conversion with truncation
REAL_CONTRACT	Longreal to real conversion
REAL_EXTEND	Real to longreal conversion

Floating Point BINARY Operations

For binary floating point operations of the form:

```
          RESULT:= LEFT <op> RIGHT
```

the equivalent Pascal procedure heading is in the form:

```
     PROCEDURE REAL_<op> (VAR LEFT,RIGHT,RESULT:REAL)
or
     PROCEDURE LONGREAL_<op> (VAR LEFT,RIGHT,RESULT:LONGREAL).
```

Binary operations supported in RealLIB:C6809 are as follows:

```
     PROCEDURE REAL_ADD (VAR LEFT,RIGHT,RESULT:REAL)
     PROCEDURE REAL_SUB (VAR LEFT,RIGHT,RESULT:REAL)
     PROCEDURE REAL_MUL (VAR LEFT,RIGHT,RESULT:REAL)
     PROCEDURE REAL_DIV (VAR LEFT,RIGHT,RESULT:REAL)
     PROCEDURE LONGREAL_ADD (VAR LEFT,RIGHT,RESULT:LONGREAL)
     PROCEDURE LONGREAL_SUB (VAR LEFT,RIGHT,RESULT:LONGREAL)
     PROCEDURE LONGREAL_MUL (VAR LEFT,RIGHT,RESULT:LONGREAL)
     PROCEDURE LONGREAL_DIV (VAR LEFT,RIGHT,RESULT:LONGREAL)
```

Floating Point UNARY Operations

For unary floating point operations of the form:

```
          RESULT:=  <op> RIGHT
```

the equivalent Pascal procedure heading is in the form:

```
     PROCEDURE REAL_<op> (VAR RIGHT,RESULT:REAL)
or
     PROCEDURE LONGREAL_<op> (VAR RIGHT,RESULT:LONGREAL).
```

Unary operations supported in RealLIB:C6809 are as follows:

```
PROCEDURE REAL_ABS  (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_NEG  (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_SQRT (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_EXP  (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_LN   (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_SIN  (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_COS  (VAR RIGHT,RESULT:REAL)
PROCEDURE REAL_ATAN (VAR RIGHT,RESULT:REAL)
PROCEDURE LONGREAL_ABS  (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_NEG  (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_SQRT (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_EXP  (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_LN   (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_SIN  (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_COS  (VAR RIGHT,RESULT:LONGREAL)
PROCEDURE LONGREAL_ATAN (VAR RIGHT,RESULT:LONGREAL)
```

Floating Point Comparison Operations

For floating point comparison operations of the form:

```
BOOLEAN:= LEFT <op> RIGHT
```

the equivalent Pascal procedure heading is in the form:

```
FUNCTION REAL_<op> (VAR LEFT,RIGHT:REAL):BOOLEAN;
```
or
```
FUNCTION LONGREAL_<op> (VAR LEFT,RIGHT:LONGREAL):BOOLEAN;
```

Comparison operations supported in RealLIB:C6809 are as follows:

```
FUNCTION REAL_EQU (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION REAL_NEQ (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION REAL_LES (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION REAL_GTR (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION REAL_LEQ (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION REAL_GEQ (VAR LEFT,RIGHT:REAL):BOOLEAN
FUNCTION LONGREAL_EQU (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
FUNCTION LONGREAL_NEQ (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
FUNCTION LONGREAL_LES (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
FUNCTION LONGREAL_GTR (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
FUNCTION LONGREAL_LEQ (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
FUNCTION LONGREAL_GEQ (VAR LEFT,RIGHT:LONGREAL):BOOLEAN
```

Floating Point Conversion Operations

For floating point conversion operations of the form:

```
RESULT:= <op> RIGHT
```

the equivalent Pascal procedure heading is in the form:

```
PROCEDURE REAL_<op> (VAR RIGHT:RIGHTtype;VAR
RESULT:RESULTtype)
```

or

```
PROCEDURE LONGREAL_<op> (VAR RIGHT:RIGHTtype;VAR
RESULT:RESULTtype)
```

Conversion operations supported in RealLIB:C6809 are as follows:

```
PROCEDURE REAL_FLOAT  (VAR RIGHT:INTEGER;VAR RESULT:REAL)
PROCEDURE REAL_ROUND  (VAR RIGHT:REAL;VAR RESULT:INTEGER)
PROCEDURE REAL_TRUNC  (VAR RIGHT:REAL;VAR RESULT:INTEGER)
PROCEDURE LONGREAL_FLOAT (VAR RIGHT:INTEGER;VAR
RESULT:LONGREAL)
PROCEDURE LONGREAL_ROUND (VAR RIGHT:LONGREAL;VAR
RESULT:INTEGER)
PROCEDURE LONGREAL_TRUNC (VAR RIGHT:LONGREAL;VAR
RESULT:INTEGER)
PROCEDURE REAL_CONTRACT  (VAR RIGHT:LONGREAL;VAR
RESULT:REAL)
PROCEDURE REAL_EXTEND    (VAR RIGHT:REAL;VAR
RESULT:LONGREAL)
```

Floating Point Error Detection

The floating point libraries have two error conditions which, when detected, cause the execution of one of two global routines: OVERFLOW and INVALID. OVERFLOW is called when an operation would produce an invalid number. INVALID is called when an invalid floating point number is passed as a parameter to one of the floating point routines.

Users may replace either of these routines with an error recovery routine of their own. In particular, defining either of these routines as a simple return from subroutine instruction (RTS) will cause the program to continue with an invalid number returned as a result.

The routines provided in the library will write a message to the buffer ERROR_MESSAGE indicating the type of error and where it occurred. They will then return and continue normal operation.

You can get this error information by entering the emulation command:

```
display memory ERROR_MESSAGE blocked word
```

which will produce a memory display indicating the error condition.

If no error has occurred, the display will appear as follows:

```
 Memory    :words  :blocked :repetitively
 address       data             :hex                    :ascii
---------------------------------------------------------------

 9003-12    4E6F 2065 7272 6F72 2020 2020 2020 2020    No error
 9013-22    2020 2020 2020 2020 2020 2020 2020 2020
 9023-32    2020 2020 2020 2020 2020 2020 2020 2020
 9033-42    2020 2020 2020 2020 2020 2020 2020 2020
 9043-52    2020 2020 2020 2020 2020 2020 2020 2020
 9053-62    2020 2020 2020 2020 2020 2020 2020 2020
 9063-72    2020 2020 2020 2020 2020 2020 2020 2020
 9073-82    2020 2020 2020 2020 2020 2020 2020 2020
 9083-92    2020 2020 2020 2020 2020 2020 2020 2020
 9093-A2    2020 2020 2020 2020 2020 2020 2020 2020
 90A3-B2    2020 2020 2020 2020 2020 2020 2020 2020
 90B3-C2    2020 2020 2020 2020 2020 2020 2020 2020
 90C3-D2    2020 2020 2020 2020 2020 2020 2020 2020
 90D3-E2    2020 2020 2020 2020 2020 2020 2020 2020
 90E3-F2    2020 2020 2020 2020 2020 2020 2020 2020
 90F3-02    2020 2020 2020 2020 2020 2020 2020 2020
```

```
 Memory    :words  :blocked :repetitively
 address       data             :hex                    :ascii
---------------------------------------------------------------

 9003-12    5265 616C 2020 2020 6572 726F 7220 2020   Real error
 9013-22    494E 5641 4C49 4420 2020 2020 2020 2020    INVALID
 9023-32    5245 414C 5F41 4444 2020 2020 2020 2020    REAL_ADD
 9033-42    726F 7574 696E 6520 6361 6C6C 6564 2020    routine
 9043-52    6279 2020 2020 2020 7573 6572 2020 2020    called by
 9053-62    6672 6F6D 2020 2020 6164 6472 6573 7320    user from
 9063-72    3143 3137 482E 2020 2020 2020 2020 2020    address
 9073-82    2020 2020 2020 2020 2020 2020 2020 2020    1C17H.
 9083-92    2020 2020 2020 2020 2020 2020 2020 2020
 9093-A2    2020 2020 2020 2020 2020 2020 2020 2020
 90A3-B2    2020 2020 2020 2020 2020 2020 2020 2020
 90B3-C2    2020 2020 2020 2020 2020 2020 2020 2020
 90C3-D2    2020 2020 2020 2020 2020 2020 2020 2020
 90D3-E2    2020 2020 2020 2020 2020 2020 2020 2020
 90E3-F2    2020 2020 2020 2020 2020 2020 2020 2020
 90F3-02    2020 2020 2020 2020 2020 2020 2020 2020
 Memory    :words  :blocked :repetitively
 address       data               :hex                    :ascii
-----------------------------------------------------------------

 9003-12    5265 616C 2020 2020 6572 726F 7220 2020  Real error
 9013-22    4F56 4552 464C 4F57 2020 2020 2020 2020   OVERFLOW
 9023-32    4C4F 4E47 5245 414C 5F41 4444 2020 2020   LONGREAL
 9033-42    726F 7574 696E 6520 6361 6C6C 6564 2020   _ADD
 9043-52    6279 2020 2020 2020 7573 6572 2020 2020   routine
 9063-72    3144 3535 482E 2020 2020 2020 2020 2020   called by
 9073-82    2020 2020 2020 2020 2020 2020 2020 2020   user from
 9083-92    2020 2020 2020 2020 2020 2020 2020 2020   address
 9093-A2    2020 2020 2020 2020 2020 2020 2020 2020   1D55H
 90A3-B2    2020 2020 2020 2020 2020 2020 2020 2020
 90B3-C2    2020 2020 2020 2020 2020 2020 2020 2020
 90C3-D2    2020 2020 2020 2020 2020 2020 2020 2020
 90D3-E2    2020 2020 2020 2020 2020 2020 2020 2020
 90E3-F2    2020 2020 2020 2020 2020 2020 2020 2020
 90F3-02    2020 2020 2020 2020 2020 2020 2020 2020
```

Floating Point Number Internal Format

The floating point numbers use the IEEE standard for the two packed formats (single precision (REAL) and double precision (LONGREAL)). The two formats are described in the following paragraphs.

Single Precision Format. The single precision floating point number used for the type REAL is a 32-bit binary value packed as follows:

```
 ____________________________________________
|   |          |                             |
| s |    e     |              f              |
|___|__________|_____________________________|
  0            8                             31
```

where:

s is the sign bit.

e is the exponent.

f is the 23-bit fraction.

The value (v) of a single precision floating point number (x) can be computed as follows:

(a) If $e=255$ and $f \neq 0$, then $v=$not a number.
(b) If $e=255$ and $f=0$, then $v=(-1)^s\infty$.
(c) If $0<e<255$, then $v=(-1)^s 2^{e-127}(1.f)$.
(d) If $e=0$ and $f \neq 0$, then $v=(-1)^s 2^{-126}(0.f)$.
(e) If $e=0$ and $f=0$, then $v=(-1)^s 0$, (zero).

The range of REAL numbers is approximately +/- 10^{38}.

Double Precision Format. A double precision floating point number used for the type LONGREAL is a 64-bit binary value packed as follows:

```
 _____________________________________________________________
|   |                   |                                     |
| s |         e         |                  f                  |
|___|___________________|_____________________________________|
  0                  11                                      63
```

where:

s is the sign bit.

e is the exponent.

f is the 52-bit fraction.

The value (v) of a double precision floating point number (x) can be computed as follows:

(a) If $e=2047$ and $f \neq 0$, then $v=$not a number.
(b) If $e=2047$ and $f=0$, then $v=(-1)^s\infty$.
(c) If $0<e<2047$, then $v=(-1)^s 2^{e-1023}(1.f)$.
(d) If $e=0$ and $f \neq 0$, then $v=(-)^s 2^{-1022}(0.f)$.
(e) If $e=0$ and $f=0$, then $v=(-1)^s 0$, (zero).

The range of LONGREAL numbers is approximately +/-10^{308}.

Chapter 5

Pascal File I/O Libraries

Introduction

The Pascal File I/O features are provided by the Pascal I/O support library: PIOLIB:C6809. The simulated I/O features of the emulation subsystem are provided by the support library: SIMLIB:C6809.

Chapter 6 of the Pascal/64000 Reference Manual contains a complete machine independent description of the routines in these libraries.

Both libraries are compiled with the options $SEPARATE ON,RECURSIVE OFF$. They will load subroutines in the PROG relocatable area and use the DATA relocatable area for local data and a message buffer for error detection.

File Error Detection

The Pascal I/O libraries support error detection as described in the Pascal/64000 Reference Manual.

If the file operations are compiled with the option,$IOCHECK OFF$, each file operation will set a global variable to indicate the result code. The user should follow each file operation with a call to the function IORESULT, defined by the Pascal I/O library, to obtain the result code of the most recent I/O operation. It is the user's responsibility to ensure the correct processing of any I/O error so that the program continues properly.

If the file operations are compiled with the option ,$IOCHECK ON$ (default case), any error detected by the I/O libraries will cause an error message to be written into 6809 memory at the location FILE_ERROR. The program will wait in the library module FMON_6809 in a loop executing the illegal opcode, 1FH. Since this mode of operation cannot assume the correct response to any arbitrary I/O error, it effectively stops the operation of the program so no further errors will occur.

When running programs compiled with option $IOCHECK ON$, in the emulation subsystem, it is recommended that the user answer the emulation configuration question "Stop processor on illegal opcodes?" in the affirmative. If a file error is then detected, the emulation status message will display:

```
   "ERROR:  6809--Reset in background   Illegal opcode 1FH at XXXXH"
```

The user can then see the file error number and the location where the file routine was called by entering the command:

    display memory FILE_ERROR blocked word

If no error has occurred, the display will appear as follows:

```
Memory    :words  :blocked
address       data            :hex                          :ascii
------------------------------------------------------------------------
  67C8-D7    4E6F 2065 7272 6F72 2020 2020 2020 2020    No error
  67D8-E7    2020 2020 2020 2020 2020 2020 2020 2020
  67E8-F7    2020 2020 2020 2020 2020 2020 2020 2020
  67F8-07    2020 2020 2020 2020 2020 2020 2020 2020
  6808-17    2020 2020 2020 2020 2020 2020 2020 2020
  6818-27    2020 2020 2020 2020 2020 2020 2020 2020
  6828-37    2020 2020 2020 2020 2020 2020 2020 2020
  6838-47    2020 2020 2020 2020 2020 2020 2020 2020
  6848-57    2020 2020 2020 2020 2020 2020 2020 2020
  6858-67    2020 2020 2020 2020 2020 2020 2020 2020
  6868-77    2020 2020 2020 2020 2020 2020 2020 2020
  6878-87    2020 2020 2020 2020 2020 2020 2020 2020
  6888-97    2020 2020 2020 2020 2020 2020 2020 2020
  6898-A7    2020 2020 2020 2020 2020 2020 2020 2020
  68A8-B7    2020 2020 2020 2020 2020 2020 2020 2020
  68B8-C7    2020 2020 2020 2020 2020 2020 2020 202E
```

If a file error has been detected the display will appear as follows:

```
Memory    :words  :blocked
address       data            :hex                          :ascii
------------------------------------------------------------------------
  67C8-D7    2020 492F 4F20 2020 2065 7272 6F72 2020      I/O      error
  67D8-E7    2020 2020 2020 2020 2020 2020 2020 3031                  01
  67E8-F7    4669 6C65 2049 4F20 726F 7574 696E 6520    File IO   routine
  67F8-07    2063 616C 6C65 6420 2020 2062 7920 2020     called      by
  6808-17    2020 7573 6572 2020 2020 6672 6F6D 2020      user      from
  6818-27    2061 6464 7265 7373 2020 3132 3536 4820     address    1256H
  6828-37    2020 2020 2020 2020 2020 2020 2020 2020
  6838-47    2020 2020 2020 2020 2020 2020 2020 2020
  6848-57    2020 2020 2020 2020 2020 2020 2020 2020
  6858-67    2020 2020 2020 2020 2020 2020 2020 2020
  6868-77    2020 2020 2020 2020 2020 2020 2020 2020
  6878-87    2020 2020 2020 2020 2020 2020 2020 2020
  6888-97    2020 2020 2020 2020 2020 2020 2020 2020
  6898-A7    2020 2020 2020 2020 2020 2020 2020 2020
  68A8-B7    2020 2020 2020 2020 2020 2020 2020 2020
  68B8-C7    2020 2020 2020 2020 2020 2020 2020 202E
```

With this display, the user can determine the number of the error which has occurred. The description of the function, IORESULT, in Chapter 6 of the Pascal/64000 reference manual contains the explanation for each error number.

If the error number is 1 and the simulated I/O library, SIMLIB:C6809, is being used, then the global variable, errno, will contain the simulated I/O error number. These errors are summarized in the Pascal/64000 Reference Manual, Chapter 6, in the section describing error reporting for the Simulated I/O library.

Appendix A

Run-time Error Descriptions

This appendix contains descriptions of run-time errors that may occur.

Error Utilities

Name	Purpose
Derrors	Debugging library error handler
Zerrors	Normal library error handler

Derrors

Derrors contains the run-time routines which store user information at the time an error occurs during debugging. The following errors may occur in the indicated library routines:

ERROR	ROUTINES
Underflow	Zbytemul,Zintmul,Zuintmul Zbyteadd, Zubyteadd, Zintadd, Zuintadd Zbytesub, Zubytesub, Zintsub, Zuintsub
Overflow	Zbytemul, Zintmul, Zuintmul Zbytediv, Zubytediv, Zintdiv, Zuintdiv Zbyteadd, Zubyteadd, Zintadd, Zuintadd Zbytesub, Zubytesub, Zintsub, Zuintsub Zbyteneg, Zintneg Zbyteabs, Zintabs
Div_by_Zero	Zbytediv, Zubytediv, Zintdiv, Zuintdiv
Case_error	User programs
Range_error	COMPB_ .

Heap_error	INITHEAP, NEW, DISPOSE, MARK, RELEASE
Set_conversion_error	Zbtoset8, Zwtoset8 Zbtoset16, Zwtoset16
String_error	MOVEST_ .

When an error is detected, a jump to Derrors is generated and valid register information is saved. The labels for the stored information are described below:

LABEL	DESCRIPTION
Z_CALLER_H Z_CALLER_L	Contain the high byte (CALLER_H) and the low byte (CALLER_L) of the address of the statement which called the routine where the actual error occurred.
Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U	Contain the contents of the registers at the time the error occurred. Only registers with information relevant to the error are saved - the indicated contents of the other registers is garbage.

NOTE: The CC register which is displayed is that which was present when the error occurred in the Debug Library routine. The CC register which was present when the Debug routine was called is not retrievable.

The following is a description of the errors that may occur and the information that is accessible when they do occur.

ERROR MSG.	DESCRIPTION	INFORMATION AVAILABLE
Z_ERR_OVERFLOW	Jump to error occurs when results of multiplication, addition, subtraction, negation, or the absolute value is too positive (i.e. INTEGERS: result > 32767 BYTES: result > 127)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U
Z_ERR_UNDERFLOW	Jump to error occurs if results of addition, subtraction, or multiplication were too negative (i.e. INTEGERS result < -32768 BYTES result < -128)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U
Z_ERR_DIV_BY_0	Jump to error occurs if division by zero is attempted by byte or integer division routines.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_REG_X Z_REG_U
Z_ERR_SET_CONV	Jump to error occurs if operand is not legal ordinal value for a set of the base type.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X

Z_ERR_RANGE	Jump to error occurs if a range declaration has been violated (i.e.: a variable does not fall within its assigned range)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_U
Z_ERR_HEAP	Jump to error occurs when some misuse of the dynamic allocation keywords NEW, DISPOSE,MARK, or RELEASE takes place.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS ?
Z_ERR_CASE	Jump to error occurs when the test variable of CASE statement is out of range and no OTHERWISE label exists.	Z_CALLER_H Z_CALLER_L Z_ACC_A Z_ACC_B
Z_ERR_STRING	Jump to error occurs on a string assingment, when the run-time size of the string being assigned is larger than that of which is it is being assigned to.	Z_CALLER_H Z_CALLER_L
Z_END_PROGRAM	Jump to message occurs when the program completes execution of the main body code.	

The illegal opcodes associated with the various errors are as follows:

OPCODE	ERROR
01	Overflow
02	Div_by_0
05	Case_error
14	Range_error
15	Recursive_error
18	Heap_error
38	Set_conversion_error
41	Underflow
42	String_size_assignment_error

Zerrors

Zerrors contains the run-time routines which store user information at the time an error occurs during execution in the non-debug library. The following errors may occur in the indicated library routines:

ERROR	ROUTINES
Case_error	User programs
Range_error	COMPB_ .
Heap_error	INITHEAP, NEW, DISPOSE, MARK, RELEASE
String_error	MOVEST_ .

When an error is detected, a jump to Zerrors is generated and valid register information is saved. The stored information, the routines and the illegal opcodes for this errors are as described in Derrors.

Z_END_PROGRAM is also called.

Index

The following index lists important terms and concepts of this manual, along with the location(s) in which they can be found. The numbers to the right of the listings indicate the following manual areas:

- Chapters - references to chapters appear as "Chapter X", where "X" represents the chapter number.
- Appendices - references to appendices appear as "Appendix Y", where "Y" represents the letter designator of the appendix.
- Figures - references to figures are represented by the capital letter "F" followed by the section figure number.
- Other entries in the Index - references to other entries in the Index are preceded by the word "See" followed by the reference entry. Otherwise, entries are referenced by page number.

64813-90903, FEBRUARY 1983
Replaces: 64813-90902, May 1981

HEWLETT PACKARD

PRINTED IN U.S.A.