HP64000 Logic Development System

Model 64819AF C/64000 Compiler Supplement 68000/68008/68010

HEWLETT PACKARD

CERTIFICATION

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other international Standards Organization members.

WARRANTY

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country.

HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

LIMITATION OF WARRANTY

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

EXCLUSIVE REMEDIES

THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES. HP SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

ASSISTANCE

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

C/64000
Compiler Supplement
68000/68008/68010

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. Vertical bars in a page margin indicate the location of reprint corrections.

First Printing .... March 1983 (P/N 64819-90901)
Second Edition .... May 1984 (P/N 64819-90902)

Table of Contents

Table of Contents (Cont'd)

List of Tables

Chapter 1

C/64000 COMPILER

INTRODUCTION

This compiler supplement is an extension of the C/64000 Compiler Reference Manual. It contains processor-dependent compiler information for use with 68000, 68008, and 68010 microprocessors.

NOTE

All references to the 68000 microprocessor in this manual are equally applicable to the 68010 microprocessor unless otherwise noted.

Descriptions of compiler features, options, and their uses are supplied. A discussion of run-time libraries required by the 68000 code generator is included. In addition, a brief discussion of the features, capabilities, and limitations of C program development using the emulator is provided.

C PROGRAM DESIGN

C 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. For example, 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. For the 68000, 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 C/64000 programs for the 68000 processor.

HOW TO IMPLEMENT A PROGRAM

The usual process of software generation is as follows:

a. Create source program files using the editor.
b. Compile source programs.
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 C/64000 compiler takes as input a program source file created with the editor. The basic form of a source file is:

```
"C"
"68000"
    .
    .
    .
    .
/*C PROGRAM*/
    .
    .
    .
    .
```

When source file editing is complete, the file is ready for compilation. Notice in the example form that the first line of the source program specifies the name of the language, and the second line specifies the name of the processor. These two lines must always be present.

There are five possible forms of compiler output: a relocatable file, a listing file (if specified), an assembly language source file (if specified), an assembly symbol file, and a compiler symbol file. These output files are described in the following paragraphs.

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, a listing file containing source lines with line numbers, program counter, level numbers, errors and expanded code (if specified) will be generated.
Assembly file:	If the ASM_FILE option is turned on anywhere in the source file, an assembly file (called ASM68000:source) will be created. This file will contain the C source as comments and the assembly language produced. This file may be assembled with the 68000 assembler. There is only one ASM68000 source file per userid. On multistation systems this means that when two 68000 compilations are done simultaneously in the same userid and both sources contain the ASM_FILE option, only one of the compilations will build the ASM68000 file.

Assembly symbol file:	If no errors were detected in the file (called FILENAME:source), an assembly symbol file (called FILENAME:asmb_sym) will be created. This file contains information about symbols that were defined in the source file and is used by the various HP 64000 analysis tools during the debugging stage. The :asmb_sym file may be suppressed by using the $ASMB_SYM OFF$ compiler directive.
Compiler symbol file:	If the compiler was executed using "options comp_sym", a compiler symbol file (called FILENAME:comp_sym) will be created. The file contains additional high-level information about symbols defined in the source file and is used by the HP 64000 high-level analysis tools during the debugging stage.

PRODUCING PROGRAMS FOR THE 68008 PROCESSOR

When compiling a program for the 68008 processor, the second line of the source file must contain the special compiler directive "68008" as shown below.

```
"C"
"68008"
     .
     .
     .
/* C PROGRAM */
     .
     .
     .
```

When linking for the 68008, all modules must be compiled or assembled using the "68008" directive or the linker will produce an error. In particular, the user must specify the 68008 run-time library file names, A5_LIB:L68008, etc., as opposed to the 68000 library files. Refer to "Linking" in the paragraphs below.

When compiling for the 68008 processor, the compiler produces exactly the same instructions as for the 68000 processor. The only difference is the relocatable files are identified as being for an 8-bit processor as opposed to a 16-bit processor. This attribute only affects the operation of the PROM Programmer.

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 input/output, signed and unsigned multiply and divide for 32-bit numbers, dynamic memory allocation, real number processing, etc. These routines must be linked with the program modules.

These routines are provided in three libraries for the 68000 and three other libraries for the 68008 processor. The names of the library relocatable files and a description of their functions is given in Table 1-1. The library files in userid L68000 should be linked with modules compiled for the 68000 processor while the library files in userid L68008 should be linked with modules compiled for the 68008 processor.

A5_LIB and ABS_LIB define the same routines and are identical in function. The differ only in the method used for accessing their own global variables. REAL_LIB does not access program level variables.

You may replace one or more of the routines in the libraries with your own version simply by specifying your own replacement routines at link time.

Table 1-1. Relocatable Library Files

Library File Name	Global Variable Addressing Mode	Functions Provided
A5_LIB:L68000 A5_LIB:L68008	A5 relative (i.e. $COMMON$)	Dynamic memory allocation, 32-bit arithmetic, string operations, range checking, program starting and ending.
ABS_LIB:L68000 ABS_LIB:L68008	absolute (i.e. FAR)	Same as A5_LIB except for global variable addressing mode.
REAL_LIB:L68000 REAL_LIB:L68008	N/A (no global variables used)	Real number operations.

LINKING EXAMPLE

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

```
link  ...

Object files: MODULE1,MODULE2,USER_LIB

Library files: A5_LIB:L68000

Load addresses: PROG,DATA,COMN,A5=0004000H,0002000H,0000000H,0000000H

    .
    .
    .

Absolute file name: PROGRAM
```

Any object files that are meant to replace files in library files A5_LIB:L68000, ABS_LIB:L68000, or REAL_LIB:L68000 should be contained in the library "USER_LIB" above.

The numbers specified for the PROG, DATA, COMN, and A5 values have the following meaning.

PROG - Specifies the starting location of the "program" relocatable memory area. Machine instructions and constant data are normally stored in the PROG area.

The value of PROG must be an even address. In the above, PROG is set to 4000H so the emulator will load the absolute file at 4000H. Note that, in general, the address specified in PROG is not the entry point where the emulator will begin execution of the absolute module.

DATA - Specifies the starting location of the DATA relocatable memory area. Program-level variables (i.e. any variable defined at the program level of a C program) are normally the DATA area.

The value of DATA must be an even address. In the example above, the linker will combine the DATA segments from each of the relocatable files above into one and load it into address 2000H.

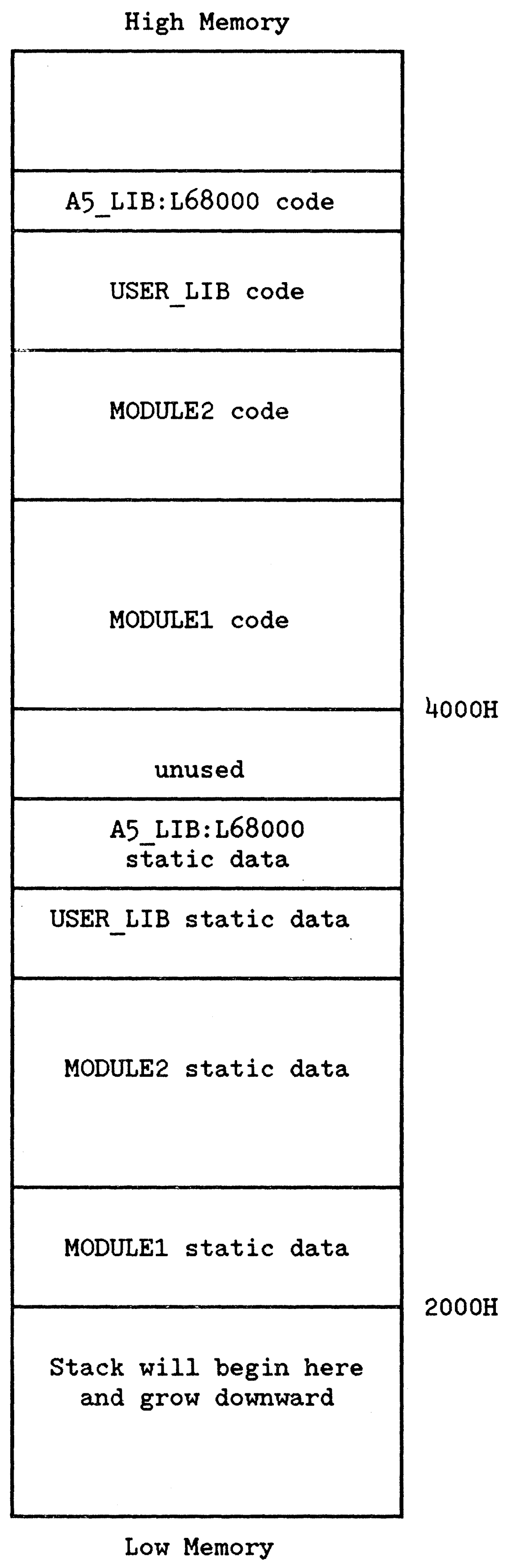
COMN - Specifies the starting location of the COMN relocatable memory area. The compiler never allocates data to the COMN area. The COMN area is seldom used. It is meant to provide for a FORTRAN-like sharable common area.

A5 - Specifies the value that will be contained in the 68000 address register A5. A5 is a register used by the compiler for accessing program level variables when the $COMMON$ compiler option is in effect. Refer to the section on program-level variable addressing which follows.

The value of A5 must be an even address. A5 should be set to the value that will actually be loaded into address register 5 at run time.

The following diagram shows how the absolute file would look in memory after linking and loading. It is assumed that the value 2000H will be loaded into register A7 to define the beginning address of the hardware-maintained user stack.

When library routines from the library A5_LIB:L68000 are required they will be linked at the end of the last user-relocatable PROG and/or DATA areas.


High Memory
A5_LIB:L68000 code
USER_LIB code
MODULE2 code
MODULE1 code
4000H
unused
A5_LIB:L68000 static data
USER_LIB static data
MODULE2 static data
MODULE1 static data
2000H
Stack will begin here and grow downward
Low Memory

PROGRAM-LEVEL VARIABLE ADDRESSING

The compiler can use any one of three addressing modes to access any particular program-level (i.e. not defined within a C routine) variable. The compiler options COMMON, BASE_PAGE, and FAR are used to select the program-level variable addressing mode in the following manner.

$COMMON$	specifies that affected variables are accessed using Address Register Indirect With Displacement mode. The address register used is A5. This is the default addressing mode.
$BASE_PAGE$	specifies that affected variables are accessed using Absolute Short Address mode.
FAR	specifies that affected variables are accessed using Absolute Long Address mode.

The addressing option, COMMON, BASE_PAGE, or FAR, that is in effect when the first C statement is encountered in a source file controls the addressing mode for all program-level variables defined within that program. After the first C statement, the options COMMON, BASE_PAGE, and FAR only affect the addressing of external variables. Consider the following example.

```
"C"
"68000"
int a;
$FAR$
extern int b;
int c;
main()
{
a=0; /* a accessed w/Address Register Indirect With Displacement mode */
b=0; /* b accessed w/Absolute Long Address mode                        */
c=0; /* c accessed w/Address Register Indirect With Displacement mode */
}
```

In the example above, the $COMMON$ option, by default, is in effect when the first C statement is encountered. Therefore all program-level variables defined in this program (i.e. a and c) will be accessed in the $COMMON$ mode in this program. Note that FAR, specified after the first C statement, affects only the externally defined variable b.

The $COMMON$ option causes affected variables to be accessed using the A5+d addressing mode. In this case, d is a 16-bit displacement which occupies two bytes of memory in the machine instruction. The displacement is first sign extended to 32 bits and then added to the contents of A5 to produce the effective address of the variable. The 16 bit displacement value allows a maximum of 64k bytes to be accessed in the DATA area. Because the 16-bit displacement is sign extended, the area of memory that can be accessed is +/- 32k bytes on either side of the address contained in A5.

The linker calculates the displacement value, d, of an A5+d variable reference in the following way. The A5 value, specified at link time, is subtracted from the actual address of the referenced variable. The result, truncated to 16 bits, becomes the displacement value. The linker will report an error if the calculated displacement is greater than 32767 or less than -32768.

The library routine Zstartprogram contained in A5_LIB or ABS_LIB sets the value of A5 to 0000000H. The linker initially sets its A5 value to 0000000H. By default then, the program may access memory locations in the range 0000000H through 0007FFFH for positive displacement values and 0FF8000H through 0FFFFFFH for negative displacement values assuming a 24-bit address width. If one desires to locate the DATA area outside of the above ranges, one must specify a new value for A5 at link time and also link to an assembly language routine which executes code to load A5 with the same new value.

The $BASE_PAGE$ option causes affected variables to be accessed using the Absolute Short Address mode. In this case, a 16-bit absolute address is contained in two bytes of memory in the machine instruction. The 16-bit address is sign extended to 32 bits to produce the effective address of the variable. The 16-bit address allows a maximum of 64k bytes to be accessed in the DATA area. Because the 16-bit address is sign extended, the accessible memory locations are in the range 0000000H through 0007FFFH for positive values and 0FF8000H through 0FFFFFFH for negative values assuming a 24-bit address width. The linker will report an error if the actual address of the referenced variable is outside these ranges.

The FAR options causes affected variables to be accessed using the Absolute Long Address mode. In this case, a 32-bit absolute address is contained in four bytes of memory in the machine instruction. The 32-bit address is the same as the effective address. This mode allows any location in memory to be accessed but at the cost of longer code and slower execution.

One may access a particular variable using different modes from different modules. For example, one module may define a global variable with the $COMMON$ option in effect and access that variable using A5+d mode within that module. Another module may declare the same variable external with the $BASE_PAGE$ or FAR option in effect and access the variable using Absolute Short or Absolute Long mode. Of course, when using $COMMON$ or $BASE_PAGE$, the actual address of the variable must always be within the range that is accessible using A5+d or absolute short addressing modes respectively.

POSITION INDEPENDENT CODE AND DATA

Several compiler options are available to the user to help him control the 68000 addressing modes used by the compiler when it generates code. The options are described in Chapter 2 under the heading Addressing Options. With these options it is possible for the user to postpone the determination of the load addresses for code and data until run time. This can be useful in a multiprogramming environment where dynamic memory mapping is required but the hardware to do it is not available.

If either the $CALL_PC_SHORT$ or the $CALL_PC_LONG$ option is used and either the LIB_PC_SHORT or the LIB_PC_LONG option is used, the executable code generated by the compiler will be position independent, and may be loaded anywhere in memory at run time. However, the linker requires that an address be specified for PROG, and the emulation loader will always load the absolute file generated by the linker at that address for emulation purposes.

If the $COMMON$ option is used before the first C statement, all references to program level variables (variables in the static data block) will be accessed using the Address Register Indirect Plus Displacement addressing mode. The address register used will be A5. The executable code generated by the compiler will be position independent and may be loaded anywhere in memory at run time. Also, the data relocatable area may be located anywhere in memory run time. However, the linker requires that addresses be specified for DATA and A5. The emulation loader will always load the data portion of the absolute file at the location specified for DATA.

If the user loads the code to a different location, then, for correct execution, the user must insure that difference, DATA - A5, specified at link time is equal to the difference of the actual run time location of the data area minus the actual run time value of A5.

The $COMMON$ option is ON by default at compiler initialization time. To cause the compiler to use absolute addressing for program level variables use the $BASE_PAGE$ option or the FAR option immediately after the "68000" line.

EMULATION OF C 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 should be initialized with the memory mapped as it will be used in the target system.

A program that is designed to be executed in read-only-memory (ROM) may be compiled with the 68000 C compiler. The $SEPARATE$ option described in the C/64000 Reference Manual is ignored by the 68000 C Compiler. C programs compiled for the 68000 are always compiled as if the $SEPARATE$ option is ON. Thus, RAM data will always be counted under the DATA counter, while code and constants are always counted under the PROG counter.

Each C absolute file must have an entry point. In C, the function "main" or "MAIN" is used as the entry point of a C program. Although many C programs may be linked to form one absolute file, only one C program should contain the function "main" or "MAIN". When the C compiler encounters the declaration of the function "main" or "MAIN", an external reference to the library routine "entry" or "ENTRY" will be generated. The routines "entry" and "ENTRY" are contained in both A5_LIB:L68000 and ABS_LIB:L68000 libraries. To execute a C absolute file in the emulator when the emulator is in the "running in monitor" state, issue the command:

```
     run from entry

          or

     run from ENTRY
```

depending on whether "main" or "MAIN" has been used as the name of the entry point function.

Program execution begins in the library routine ENTRY, where the run-time environment for the program may be initialized. The library routines "entry" and "ENTRY" initialize registers A5 and A6 to all zeros, and load A7 with the highest address of a 256-word stack. The user may write his own ENTRY routine to initialize the environment in some other way. This is most often required when the user needs more than 256 words for the stack. After the run-time environment is initialized, control is transferred to the function "main" or "MAIN" and the program begins execution. If the program runs to completion, control will return to the ENTRY routine and the message "End of Program" will be displayed on the status line of the CRT.

Alternatively, the user may turn the $ENTRY$ option OFF at the beginning of each C program. With the $ENTRY$ option OFF, no external reference to either library entry routine is generated. In this case, the user is responsible for run-time environment initialization. Normally, for emulating C programs, the $ENTRY$ option should be left ON to provide a fail-safe method for doing run-time environment initialization.

When executing code on the emulator, if a run time error occurs, a jump to the monitor will be generated and a message will be displayed on the status line indicating the error and, in many cases, the address where the error occurred.

NOTE

It is important to remember that during emulation of C/64000 programs, a C program may be debugged symbolically (using global symbols in the source program) or by source program line numbers of the form: #n, where n is the source line number of an executable C line.

Chapter 2

C/64000 PROGRAMMING

PROGRAMMING CONSIDERATIONS

INTRODUCTION

This chapter describes the run-time environment for 68000 C/64000 programs. Although some parts of the run time environment are not necessary for every C program, the programmer should become familiar with the information supplied in order to be able to use it when the structure of a 68000 program does require it. The specific areas to be discussed are stack pointer initialization, multiple module programs, heap initialization for use with the dynamic memory routines (NEW, DISPOSE, MARK, and RELEASE) and interrupt processing with C programs.

REGISTER ALLOCATION

The 68000 has eight data (D) registers and eight address (A) registers. The 68000 makes use of all 16 registers. Addresses are computed only in the A registers while data values are computed only in the D registers. Data registers 0 through 6 are used for expression evaluations. Data register 7 is the function value return register. In other words, if the value of a function is 4 bytes or less, the value will be returned in D7. Address registers 5, 6, and 7 are permanently allocated by the compiler for specific tasks. The user must never destroy the addresses contained in these registers.

A7 is the stack pointer; it always points to the last item on the stack. A6 is the local frame pointer. It always points to the highest address of the data area of the currently executing function.

The LINK and UNLK instructions are used by the compiler with A6 and A7 to maintain a linked list of local data areas for nested procedure calls.

A5 is the static data pointer. If the $COMMON$ option is used to access variables anywhere in the program, then A5 must be initialized to the same value that was specified for A5 at link time. If the $COMMON$ option is never used (remember that $COMMON$ is ON by default), then register A5 will not be used by the compiler. If the $COMMON$ is used for accessing program-level variables, use the library A5_LIB:L68000 when linking. If either $BASE_PAGE$ or $FAR$ is used for accessing global variables, use the library ABS_LIB:L68000 when linking.

STACK POINTER INITIALIZATION

The stack pointer (Address Register 7) is a hardware register maintained by the processor. Prior to use, however, it must be initialized to the highest address of the stack.

The libraries A5_LIB:L68000 and ABS_LIB:68000 contain routines for default runtine environment initialization called entry and "ENTRY". When a C program containing function "main" or "MAIN" is compiled with the $ENTRY$ option ON ($ENTRY$ is ON by default), an external reference will be made to "entry" or "ENTRY". When the compiled program is linked with library A5_LIB or ABS_LIB, the linker will select the appropriate routine, "entry" or "ENTRY", for default run-time environment initialization. These routines set registers A5 and A6 to all zeros, and load the address of a 256-word stack into register A7. If a larger stack is needed, or if other initialization is required, the user must provide his own entry routine. An example of a run-time environment initialization routine that replaces ENTRY in library A5_LIB or ABS_LIB is as follows:

```
"68000"
                GLOBAL ENTRY
ENTRY
                MOVEA.L         #0,A5
                MOVEA.L         #0,A6
                LEA             Zstack,A7
                JSR             MAIN[PC]
                JMP             Zendprogram[PC]
            EXTERNAL            MAIN
            EXTERNAL            Zendprogram
                DATA
                DS.W            2500
Zstack          DS.W            1
                END             ENTRY
```

When this program is assembled and listed as an object file during the linking process, it will replace the routine called ENTRY in library A5_LIB or ABS_LIB. Registers A5 and A6 are cleared to all zeros and register A7 is loaded with the highest address of a 2500-word stack (in this example). Zendprogram is a routine in libraries A5_LIB and ABS_LIB that can be called to display the message "End of Program" on the status line of the CRT when the program run is completed. Zendprogram returns control to the emulation monitor; therefore, there is no RTS instruction after it is called. Note that MAIN is spelled with capital letters as is ENTRY. If , in the C program declaring MAIN, lower-case letters are used, then in the example above, MAIN and ENTRY should also be spelled with lower-case letters. Note also that the stack must begin and end on an even address.

FUNCTIONS AND PARAMETERS PASSING

All parameters are passed on the stack. An array is passed by pushing its address on the stack. A value of 4 bytes or less is passed by pushing the value on the stack. Structures larger than four bytes have their addresses pushed on the stack like reference parameters. Then at the function entry, the large structures are copied into the local data area of the function. The local data of the function is also allocated on the stack at the function entry point with the link instruction.

If a parameter is a byte, it is placed in the most significant byte of the word on the stack with the least significant byte untouched. Parameters that are two words long are pushed one word at a time, with the word which is uppermost in memory pushed first.

Parameters are normally passed from right to left, i.e., the right-most parameter is pushed first. If the $FIXED PARAMETERS$ option is ON, the parameters are passed from left to right as in Pascal. When calling a Pascal procedure or function from a C program, be sure that the $FIXED_PARAMETERS$ option is ON when the procedure or function is declared as an external function in the C program.

After all the parameters are pushed, the function is called. After function call, the compiler generates code to pop the parameters off the stack.

VALUE RETURNING FUNCTIONS

If the value returned is larger than four bytes, the calling routine passes the address of a temporary local variable that will hold the function result. This address is passed after the last parameter and immeadiately before the function is called. When incrementing the stack pointer after the call, the pointer to the temporary variable is also removed.

Care must be taken when calling a function indirectly through a pointer when the function returns a value larger than four bytes. In this case, one must always use the function result in some way. If this is not done, the compiler will fail to pass the address of the temporary result area and the code will not work properly.

STATIC DATA AREA

The static data area is composed of the static data areas of all the separately compiled C modules. Usually, the static data area is a single continuous block of memory but this need not be the case. It is possible, when linking, to locate the DATA (or PROG) areas of the various modules in several separate areas of memory.

The $COMMON$, $BASE_PAGE$, and FAR compiler options control the addressing mode used by the 68000 to access the variables in the static data area. Usually, the programmer specifies that all variables in all modules be accessed using the same addressing mode but this need not be the case. It is possible for one module's variables to be accessed with one mode and a different module's variables to accessed with a different mode. It is also possible for a single variable to be accessed using different modes in different modules. See the section on Program-level variable addressing in Chapter 1 for more information.

DYNAMIC MEMORY ALLOCATION HEAP INITIALIZATION

Pascal defines several memory management routines that can also be used in C programs. However, the standard names are unknown to the C compiler, so they must be declared in any C program in which they are used. Since these routines are written in Pascal, they should be declared with the $FIXED_PARAMETERS$ option ON so that parameters will be passed left to right.

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

```
     INITHEAP(start_address,Length_in_bytes)
       long  *  start_address;
       long  Length_in_bytes;
```

The procedure INITHEAP must be used as declared above. Start_address should point to the smallest address of the memory block to be used, and it must be an even address. Length_in_bytes must be an even value.

For example, if the block to be used is located in memory from 4000H to 5FFFH, the initialization should appear as follows:

```
#define HEAPSIZE    0X2000

$ORG   0X4000$
long heapstart;
$END_ORG$
$FIXED_PARAMETERS ON$
   extern INITHEAP();
$FIXED_PARAMETERS OFF$
        .
        .
        .
INITHEAP(&heapstart,(long)HEAPSIZE);
        .
        .
        .
```

NEW is used to obtain a block of memory from heap and should be used as declared below:

```
NEW(ppblock,numbytes;)
type **ppblock;
unsigned long numbytes;
```

On return, *ppblock will contain a pointer to the block which has been allocated. The number of bytes, numbytes, must be even to ensure proper alignment.

DISPOSE will return a block to the heap that was previously obtained by a call to NEW and should be used as declared below:

```
DISPOSE(ppblock,numbytes)
type **ppblock;
unsigned long numbytes;
```

Only blocks which have been previously allocated by NEW should be DISPOSEd and the amount returned should be the same as the amount allocated.

Example:

```
     $FIXED_PARAMETERS ON$
        extern NEW();
        extern DISPOSE();
     $FIXED_PARAMETERS OFF$
        struct block   {
          long i1,i2;
          struct block *next;
        } *pblock;
              .
              .
              .
          NEW(&pblock,(long)sizeof(struct block));
              .
              .
              .
          DISPOSE(&block,(long)sizeof(struct block));
```

MARK will set its parameter to the address of a location on the heap and should be used as declared below:

```
     MARK(ppmark)
     long **ppmark;
```

When the pointer is later RELEASEd:

```
     RELEASE(ppmark)
     long **ppmark;
```

everything which has been allocated by calls to NEW since MARK will be DISPOSEd. The user may have any number of MARKs.

Example:

```
     $FIXED_PARAMETERS ON$
        extern MARK();
        extern RELEASE();
        extern NEW();
     $FIXED_PARAMETERS OFF$
        long *pmark;
            .
            .
            .
        MARK(&pmark);
            .
            .
            .
        NEW(...);
            .
            .
            .
        RELEASE(&pmark);
```

Twelve bytes of heap space are used when the heap space is initialized. Twelve bytes of heap space are also used each time the heap is marked. When NEW is called, the minimum memory allocated is 8 bytes, even if fewer bytes are required.

Items must be allocated in an even number of bytes. The user must ensure that NEW and DISPOSE always pass even sizes.

INTERRUPT HANDLING

Interrupt handling routines may be written in C using the INTERRUPT option. Additionally, code produced by the compiler is safely interruptable as long as the interrupt driven process saves and restores the registers and returns with a return from exception (RTE) instruction. The compiler does not automatically generate the interrupt vectors for procedures defined as interrupt procedures.

SPECIAL OPTIONS FOR THE 68000 COMPILER

The following options have special functions for the 68000 compiler.

INTERRUPT
Default OFF

All functions defined while the $INTERRUPT$ option is ON will be suitable for use as interrupt routines. On entry to the function, all registers will be saved on the stack. On exit, the registers will be restored and RTE is used to return instead of the normal RTS. $INTERRUPT$ functions may not be called and they may not have parameters. The value of $INTERRUPT$ applies at the function heading. It is the user's responsibility to set up the interrupt vectors to point to the appropriate $INTERRUPT$ functions.

OPTIMIZE
Default OFF

When $OPTIMIZE$ is OFF, storing indirectly or into an external will cause the contents of remembered registers to be forgotten because there is a possibility that their contents will be incorrect. If $OPTIMIZE$ is ON, these registers will not be forgotten. It is possible that the code generated will be incorrect, although in most cases it will be correct. An example which produces incorrect code is shown on the following page.

```
"C"
"68000"
$BASE_PAGE$
 long i,j,k,l;
    long * ip;
 p()
  {
  p
      LINK         A6,#0
      ip = &i;
         LEA          static00_D,A0
         MOVE.L       A0,static00_D+00010H
      j = 3;
         MOVE.L       #3,static00_D+00004H
      k = 2;
         MOVE.L       #2,static00_D+00008H
   clear:
      i = j + k;
     clear01
          MOVE.L      static00_D+00004H,D0
          ADD.L       static00_D+00008H,D0
          MOVE.L      D0,static00_D
       *ip = k;
          MOVEA.L     static00_D+00010H,A1
          MOVE.L      static00_D+00008H,[A1]
      l = i;
          MOVE.L      D0,static00_D+0000CH
    }
          UNLK        A6
          RTS
          DATA
     static00_D
          DC.L        0
          DC.L        0
          DC.L        0
          DC.L        0
          DC.L        0
          PROG
     Ep                  EQU $-1
          GLOBAL      Ep
                             GLOBAL   i
                             GLOBAL   j
                             GLOBAL   k
                             GLOBAL   l
                             GLOBAL   ip
                             GLOBAL   p

End of compilation, number of errors=     0
```

If $OPTIMIZE$ is ON, L will be assigned 5 instead of 2. Errors may also occur if two externals and/or absolutes refer to the same address.

Another optimization occurs when $OPTIMIZE$ is ON; forward jumps will be assumed to be within 128 bytes. This saves two bytes for each forward jump. If the label is out of range, an error (number 1200) will be given in pass 3. If this occurs, turn $OPTIMIZE OFF$ around the line that caused the error. Normally, programs may be compiled with $OPTIMIZE ON$ without fear of generating incorrect code.

SEPARATE
Default OFF
SEPARATE has no effect for the 68000. PROG and DATA are always separate.

TRAP
Default -1
When a function heading is encountered in the source, the value of TRAP will be checked. If it is a value from 0 to 15, the function will be declared as a TRAP function. For the option $TRAP=5$ a function name will be assigned the trap value 5. When any calls to this function are encountered in the source, the compiler will generate a TRAP 5 instruction rather than a JSR instruction. Trap functions may have parameters like any other function. After the value of TRAP has been assigned to a function name, the value of TRAP is set to -1. The trap number used where the function is declared global must be the same one used where the function is declared external. The compiler generates a call to Zentertrap at the entry point of a trap function. Trap functions return via the RTE instruction. It is the user's responsibility to set up the trap vectors to point to the appropriate TRAP functions. TRAP functions may not return a value.

ADDRESSING OPTIONS

The following three options allow the user to control the addressing mode for any variables defined as extern as well as the entire program level data block. One and only one of these options is always ON. The option that is ON when the first C statement is encountered by the compiler shall control the addressing mode of all the program-level variables defined in that program. After the first C statement, the options only affect extern variables.

If the $COMMON$ option is used ($COMMON$ is ON by default), link with library A5_LIB:L68000. Otherwise, link with the library ABS_LIB:L68000.

BASE_PAGE
Default OFF
All external variables defined while $BASE_PAGE$ is ON will be accessed by the absolute short addressing mode. The linker will report an error if a base page variable is linked to an address outside the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH. Insert $BASE_PAGE$ before the first C statement if absolute short addressing is desired for locally defined program level variables.

FAR
Default OFF
All external variables defined while FAR is ON will be accessed by the absolute long addressing mode. Insert FAR before the first C statement if absolute long addressing is desired for locally defined program level variables.

COMMON
Default ON
All external variables defined when $COMMON$ is ON will be accessed with the address register indirect plus displacement mode. The address register will be A5. Insert $COMMON$ (or nothing since $COMMON$ is ON by default) before the first C statement if A5+d addressing is desired for locally defined program level variables.

The linker will report an error if the calculated displacement is greater than 32767 or less than -32768. The displacement is equal to the actual address of the referenced variable minus the A5 value that was specified to the linker.

The following four options allow the user to control the addressing modes for calling functions. One and only one of these options is always on. The addressing mode used to call a function is determined by the setting of the options when the function heading is encountered.

CALL_ABS_LONG
Default OFF
Functions defined while $CALL_ABS_LONG$ is ON will be called with the absolute long addressing mode.

CALL_ABS_SHORT
Default OFF
Functions defined while $CALL_ABS_SHORT$ is ON will be called with the absolute short addressing mode. The linker will report an error if an attempt is made to call a function with the absolute short addressing mode if the function is not assigned an address within the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH.

CALL_PC_SHORT
Default ON
Functions defined while $CALL_PC_SHORT$ is ON will be called using the program counter plus displacement addressing mode. The linker will report an error if an attempt is made to call such a function whose displacement from the current program counter is greater than +- 32K bytes.

CALL_PC_LONG
Default OFF
Functions defined while $CALL_PC_LONG$ is ON will be called using the program counter plus displacement + index addressing mode. This option should be used only when necessary because loading the index portion for each call is inefficient.

The following four options allow the user to control the addressing modes used for calling predefined functions. They are similar to the previous four options, but they are applied to each function call individually. One and only one of these options is always ON.

LIB_ABS_LONG
Default OFF
Calls to predefined functions encountered while LIB_ABS_LONG is ON will use the absolute long addressing mode.

LIB_ABS_SHORT
Default OFF
Calls to predefined functions encountered while LIB_ABS_SHORT is ON will use the absolute short addressing mode. The linker will report an error if an attempt is made to call a predefined function with the short absolute mode and the address of the function is not in the range 0000000H through 0007FFFH or 0FF8000H through 0FFFFFFH.

LIB_PC_SHORT
Default ON
Calls to predefined functions encountered while LIB_PC_SHORT is ON will use the program counter plus displacement addressing mode. The linker will report an error if an attempt is made to call such a function and the displacement from the call is greater than +-32K bytes.

LIB_PC_LONG
Default OFF
Calls to predefined functions encountered while LIB_PC_LONG is ON will use the program counter plus displacement plus index addressing mode. This option should be used only when necessary, because loading the index for each call is inefficient.

USER-DEFINED OPERATORS

GENERAL

C/64000 allows the user to redefine the meaning of certain operators. User defined operators are created by using the option: $USER_DEFINED$ during the declaration of a user type. The option, when used, applies to the next type definition encountered.

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 THAT MAY BE REDEFINED

The following is a list of operators that can be redefined associated with the routine that the compiler will create for the operation.

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 68000 compiler will use a multi-byte move loop for types larger than four bytes, or a regular move for types smaller than four bytes.

PARAMETERS FOR USER DEFINED OPERATIONS

For the 68000, the parameters are passed on the stack as follows:

1) The address of the first operand is pushed on the stack.

2) The address of the second operand is pushed on the stack.

3) The address of the result is pushed on the stack if the result is larger than four bytes. Otherwise, the compiler expects the result to be returned in data register 7.

Negate has only one operand and a result.

Relational operations will not pass an address for the result. Instead, a Boolean value should be returned in data register 7 as follows:

True:	D7 set to 1
False:	D7 set to 0

User routines may be written in C. For example:

```
USER_MUL (OPERAND2,OPERAND1)
  USER *OPERAND1, *OPERAND2;

Short USER_LES (OPERAND2,OPERAND1)
  USER *OPERAND1, *OPERAND2;
```

NOTE

All parameters are passed by reference (VAR parameters). Functions with result values smaller than five bytes return the result value in D7. The parameter order above may be reversed by using the $FIXED_PARAMETER$ option.

The following example is an expanded listing demonstrating use of a user type.

```
"C"
       EXTERNAL mat1
       EXTERNAL mat2
       EXTERNAL mat3
       EXTERNAL flag
       EXTERNAL entry
"68000"
$BASE_PAGE$
#define MAXSIZE 5
$USER_DEFINED$
 struct  {
           int mat[MAXSIZE] [MAXSIZE];
           int mat[5 ] [ 5];
           short nrows,ncolumns;
        } typedef MATRIX;
 extern MATRIX mat1,mat2,mat3;
 extern int flag;
main()
{
 main
     LINK      A6,#-52
 mat1 = mat2 + mat3 * mat1;
     PEA       mat3
     PEA       mat1
     PEA       -52[A6]
     JSR       MATRIX_MUL[PC]
     ADDA.L    #12,A7
     PEA       mat2
     PEA       -52[A6]
     PEA       mat1
     JSR       MATRIX_ADD[PC]
     ADDA.L    #12,A7
 flag = mat  != mat2;
     PEA       mat1
     PEA       mat2
     JSR       MATRIX_NEQ[PC]
     ADDQ.L    #8,A7
     MOVE.W    D7,flag
}
     UNLK      A6
     RTS
 Emain            EQU $-1
     GLOBAL    Emain
               EXTERNAL MATRIX_ADD
               EXTERNAL MATRIX_MUL
               EXTERNAL MATRIX_NEQ
               GLOBAL   main

End of compilation, number of errors=  0
```

PASS 2 ERRORS

Pass 2 errors will be displayed on the screen with the message:

```
LINE # <line number>--PASS2 ERROR # <Pass2 error number>
```

In addition, if a listing file has been indicated for the compilation, the compiler will indicate pass 2 errors where they occurred in the listing. It will also list the meaning of each error.

Pass 2 error numbers will always be >=1000. Errors with numbers between 1000 and 1099 are fatal errors. Errors with numbers >=1100 are non-fatal errors.

Pass 2 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. Pass 2 Errors

1000 - "Out of memory"
The 68000 code generator has run out of memory, break up your program and recompile. This error can also occur if there is a bug in the compiler itself. If you feel that you have not used up the entire symbol table, contact Hewlett-Packard.

1001 - "This error can't possibly occur #1"
Contact Hewlett-Packard.

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

1003 - "This error can't possibly occur #2"
Contact Hewlett-Packard.

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

1005 - "Unimplemented feature"
You have used a feature of C that has not been implemented in the 68000 code generator.

1006 - "Compiler error. Contact Hewlett-Packard"
This error should never occur. Please report this error to Hewlett-Packard as soon as possible.

Table 2-1. Pass 2 Errors (Cont'd)

1008 - "All data registers are active"
Even though the 68000 has 8 data registers, it is still possible to write an expression that will require more. Break up the expression and recompile.

1009 - "All address registers are active"
The compiler computes addresses in address registers A0 - A4. You have succeeded in writing an expression that requires computation of more than 5 addresses. Break up the expression and recompile.

1100 - "Bounds error"
An attempt was made to store a value into a result which was too small; for example assigning 300 to a byte. This error will occur if the $RANGE$ option is on.

1103 - "Interrupt procedure must not have parameters"
An interrupt procedure cannot have parameters.

1104 - "Interrupt procedure call not allowed"
An interrupt routine can only be accessed through an interrupt vector, since it will return with an RTE instead of an RTS.

1105 - "Data size too large"
More than 32K bytes of data have been allocated for this procedure.

1106 - "Trap or interrupt routine may not be a function"
Only procedures may be trap or interrupt routines.

1107 - "Data counter overflow"
The DATA section has become larger than 32K bytes.

1108 - "Trap number must be 0 to 15"
The 68000 has 16 trap vectors numbered 0 to 15.

1113 - "Program counters do not agree"
If this error occurs before any other error then it means there is a bug in the compiler - contact Hewlett-Packard. If this error occurs with some other error, ignore it.

1200 - "Long range error; turn off OPTIMIZE for this line"
The compiler has tried to generate an 8-bit jump where a 16-bit jump is required. Turn off the OPTIMIZE option around the source line where the error is reported, and recompile.

Chapter 3

RUN-TIME LIBRARY SPECIFICATIONS

INTRODUCTION

This chapter describes the 68000 run-time libraries A5_LIB:L68000, A5_LIB:L68008, ABS_LIB:L68000, ABS_LIB:L68008, REAL_LIB:L68000, and REAL_LIB:L68008. When linking, use the library files in userid L68000 if you have compiled your programs with "68000" in the source file. Use the the library files in userid L68008 if "68008" is in the source file.

A5_LIB and ABS_LIB define the same routines and are identical in function. The difference is that A5_LIB uses the A5+d (i.e. $COMMON$) addressing mode to access its own program-level variables while ABS_LIB uses the absolute long (i.e. $FAR$) addressing mode.

Usually, programmers use one access method, COMMON, BASE_PAGE, or FAR in all separately compiled modules of a program. In this case, if you use $COMMON$ ($COMMON$ is ON by default) for accessing program-level variables, link to A5_LIB. Otherwise, if your program uses $BASE_PAGE$ or $FAR$, use ABS_LIB. If you mix accessing modes, you can use either library. If you use A5_LIB, you must insure that the run-time value of A5 is equal to the value of A5 specified at link time.

The following paragraphs describe library routines that are called when operations encountered by the C compiler for which there is no 68000 instruction. Descriptions of some of the more useful predefined Pascal routines are also included.

DYNAMIC MEMORY ALLOCATION

The dynamic memory allocation routines that are required for Pascal may also be used in C programs. However, they are unknown to the C compiler and must be declared external.

INITHEAP

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Initialize a block of memory as a memory pool or "heap".
Declaration:	extern short HEAP[4000]; /*4000 byte heap*/ $FIXED_PARAMETERS ON$ extern INITHEAP(); $FIXED_PARAMETERS OFF$
Calling Sequence:	INITHEAP(&HEAP,(long) sizeof (HEAP);
Parameters:	The first parameter is the address of a block of memory. The second parameter, which must be passed as a long integer, is the length of the heap in bytes.
Return Value:	none.
Remarks:	12 bytes of the heap will be used to initialize the heap. INITHEAP is written in Pascal.

NEW

Specifications for this routine are as follows:

Library: A5_LIB and ABS_LIB Purpose: Allocate a block of memory from the heap.

Declaration:
```
$FIXED_PARAMETERS ON$
extern NEW();
$FIXED_PARAMETERS OFF$
```

Calling Sequence: `NEW(&POINTER,(long) sizeof (*POINTER);`

Parameters: The first parameter must be the address of a pointer. The second parameter is the number of bytes of memory required, and it must be passed as a long integer.

Return Value: There is no return value but the first parameter will contain the address of the allocated block of memory.

Remarks: A minimum of 8 bytes is allocated even if fewer than 8 bytes are required. The number of bytes of memory being allocated must be even.

DISPOSE

Specifications for this routine are as follows:

Library: A5_LIB and ABS_LIB

Purpose: Deallocate a block of memory.

Declaration:
```
$FIXED_PARAMETERS ON$
extern DISPOSE()
$FIXED_PARAMETERS OFF$
```

Calling Sequence: `DISPOSE(&POINTER,(long) sizeof (*POINTER);`

Parameters: The first parameter must be the address of a pointer. The pointer contains the address of the block of memory that is to be disposed. The second parameter is the number of bytes of memory being disposed. It must be passed as a long integer.

Return Value: none.

Remarks: The number of bytes of memory being disposed must be even.

MARK

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Mark the state of the heap in a pointer variable.
Declaration:	extern MARK();
Calling Sequence:	MARK(&POINTER);
Parameters:	The parameter is the address of the pointer variable where the state of the heap is to be stored.
Return Value:	There is no return value but the contents of the parameter will be set to the address representing the state of the heap.
Remarks:	POINTER should not be modified until RELEASE(&POINTER); has been executed.

RELEASE

Specifications for the routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Restore heap to the state contained in the pointer variable parameter.
Declaration:	extern RELEASE();
Calling Sequence:	RELEASE(&POINTER);
Parameters:	The parameter is the address of the pointer variable where the state of the heap was previously marked.
Return Value:	none.
Remarks:	POINTER must have been set by a previous call to MARK.

MEMERR

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	When an error occurs, MEMRR is called to display an appropriate message on the status line of the display. The possible errors are:

0: **Heap length too small.** INITHEAP was called incorrectly or the value passed as the heap length was too small.

1: **Heap has not been initialized.** INITHEAP was not called before the first call to NEW, DISPOSE, MARK, or RELEASE.

2: **No free space in current mark.** Space may exist in previous marks but is not available to NEW until those marks are released.

3: **No block large enough to allocate.** Smaller fragmented blocks may exist.

4: **Pointer variable points outside of heap.** DISPOSE or RELEASE was called with garbage for an address.

5: **No free space in heap.** All the memory is used.

6: **Unable to mark, no block large enough.** Each mark requires 12 bytes of the heap. There was no block of 12 or more bytes available.

7: **Attempt to release mark that does not exist.** RELEASE was called with a pointer containing an address that was not the result of a call to MARK.

Declaration:	MEMERR(ERROR_NUMBER);
	short ERROR_NUMBER;

Calling Sequence:	not applicable.
Parameters:	The parameter is a byte representing the number of the error.

Return Value: none. Control is not returned to user program.

Remarks: The supplied version of MEMERR is shown below. It transfers control to the emulator monitor by calling Zerror after storing the address of a message in MONITOR_MESSAGE. The user must supply his own version of MEMERR when his software is executed without the emulator.

```
"68000"
          GLOBAL MEMERR
          EXTERNAL MONITOR_MESSAGE,Zerror
*MONITOR_MESSAGE is a global variable in the 68000 emulator
*monitor. It is used to store the address of an ASCII
*message that the monitor is supposed to write on the status
*line of the display.
*Zerror is called to transfer control to the monitor for the
*purpose of displaying an error message on the status line
*of the display. Zerror also converts a binary address into
*ASCII and stores it in the error message.

MEMERR

         MOVE.B    4[A7],D0            ;Fetch the error number.
         MOVEA.L   A6,A1
         CMPI.B    #4,D0               ;Is the error number 4?
         BNE.S     NOT_4               ;Branch if error is not 4.
         MOVEA.L   [A1],A1             ;Error 4 means address
                                       ;where error occurred is
                                       ;in a different place.
NOT_4
         ADDI.B    #030H,D0            ;Make error number into
                                       ;ASCII char.
         MOVE.B    D0,ERROR            ;Store char in message.
         LEA       MESSAGE,A0          ;Get addr of error msg.
         MOVE.L    A0,MONITORE_MESSAGE ;Store it for monitor.
                                       ;Store it for monitor.
         MOVE.L    4[A1],D0            ;Addr where error occurred.
         JMP       Zerror[PC]          ;Convert addr and display
                                       ;message.
         JMP       $[PC]               ;Endless loop if monitor
                                       ;returns.
         DATA
MESSAGE  DC.B  END_MESSAGE-MESSAGE-1
         ASC  "Memory error #"
ERROR    ASC  " at      "              ;Converted to "n at xxxx"
END_MESSAGE
```

32-bit Arithmetic

The 68000 does not have instructions for doing 32-bit multiplies and divides. When the compiler encounters a 32-bit multiply, divide, or modulus operation one of the following routines is called.

Zunsmult - Unsigned 32-bit multiply

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Multiply two 32-bit unsigned numbers yielding a 32-bit unsigned result.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	Two 32-bit unsigned values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit unsigned value in data register D7.
Remarks:	If this routine is to be replaced, note that the replacement routine must pop the parameters off the stack before returning. When the $DEBUG$ option is ON the compiler will generate a call to Dunsmult instead of Zunsmult.

Zmult - Signed 32-bit multiply

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Multiply two 32-bit signed numbers yielding a 32-bit signed result.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	Two 32-bit signed values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit signed value in data register D7.
Remarks:	If this routine is to be replaced, note that the replacement routine must pop the parameters off the stack before returning. When the $DEBUG$ option is ON the compiler will generate a call to Dmult instead of Zmult.

Zunsdiv - Unsigned 32-bit divide

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Divide one 32-bit unsigned number by another 32-bit unsigned number yielding a 32-bit unsigned result.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR / RIGHT_FACTOR;
Parameters:	Two 32-bit unsigned values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit unsigned value in data register D7.
Remarks:	If this routine is to be replaced, note that the replacement routine must pop the parameters off the stack before returning. If division by zero is attempted a zero division exception is initiated.

Zdiv - Signed 32-bit divide

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Divide one 32-bit signed number by another 32-bit signed number yielding a 32-bit signed result.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR / RIGHT_FACTOR;
Parameters:	Two 32-bit signed values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit signed value in data register D7.
Remarks:	If this routine is to be replaced, note that the replacement routine must pop the paremeters off the stack before returning. If division by zero is attempted a zero division exception is initiated.

Zmodu - Unsigned 32-bit modulus

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Divide one 32-bit unsigned number by another 32-bit unsigned number yielding a 32-bit unsigned modulus.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR % RIGHT_FACTOR;
Parameters:	Two 32-bit unsigned values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit unsigned value in data register D7.
Remarks:	If this routine is to be replaced note that the replacement routine must pop the parameters off the stack before returning. If division by zero is attempted a zero division exception is executed.

Zmods - Signed 32-bit modulus

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Divide one 32-bit signed number by another 32-bit signed number yielding a 32-bit signed modulus.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR % RIGHT_FACTOR;
Parameters:	Two 32-bit signed values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit signed value in data register D7.
Remarks:	If this routine is to be replaced note that the replacement routine must pop the parameters off the stack before returning. If division by zero is attempted a zero division exception is executed.

Dunsmult - Unsigned 32-bit multiply with overflow check

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Multiply two 32-bit unsigned numbers yielding a 32-bit unsigned result. If overflow occurs an overflow exception is initiated by executing the TRAPV instruction.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	Two 32-bit unsigned values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit unsigned value in data register D7.
Remarks:	If this routine is to be replaced note that the replacement routine must pop the parameters off the stack before returning. When the $DEBUG$ option is ON the compiler will generate a call to Dunsmult instead of Zunsmult.

Dmult - Signed 32-bit multiply with check for overflow

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Multiply two 32-bit signed number yielding a 32-bit signed result. If overflow occurs an overflow exception is initiated by executing the TRAPV instruction.
Declaration:	not applicable.
Calling Sequence:	RESULT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	Two 32-bit signed values are pushed onto the stack. The left factor is pushed first followed by the right factor. The parameters are popped off the stack before returning.
Return Value:	32-bit signed value in data register D7.
Remarks:	If this routine is to be replaced note that the replacement routine must pop the parameters off the stack before returning. When the $DEBUG$ option is ON the compiler will generate a call to Dmult instead of Zmult.

Error Routines

Zerror - error message formatter

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Converts an address in D0 to ASCII, stores the ASCII address into the message pointed to by MONITOR_MESSAGE and transfers control to the emulator monitor to display the message on the status line of the display.
Declaration:	not applicable.
Calling Sequence:	not applicable; called by MEMERR only.
Parameters:	D0 contains the address to be converted. MONITOR_MESSAGE contains the address of an error message.
Return Value:	none. Control is not returned to the calling program.
Remarks:	The supplied version of Zerror is shown below. It transfers control to the emulator monitor by calling the monitor entry point JSR_ENTRY. Zerror must be modified or replaced for reporting errors when executing without the emulator.

```
"68000"
         GLOBAL Zerror
         EXTERNAL MONITOR_MESSAGE,JSR_ENTRY
*JSR_ENTRY is the entry point of the emulator monitor.
*MONITOR_MESSAGE contains the address of the message.
*
Zerror
         MOVEA.L   MONITOR_MESSAGE,A0   ;address of message
         CLR.L     D1
         MOVE.B    [A0],D1         ;message length
         ADDQ.B    #1,D1           ;for predecrement mode
         ADDA.L    D1,A0           ;1 byte after end of message
         MOVE.B    #6,D2           ;# of addr characters
LOOP
         MOVE.B    #15,D1          ;mask 4 lowest bits
         AND.B     D0,D1
         ADDI.B    #30H,D1         ;30H is zero ASCII
         CMPI.B    #39H,D1         ;bigger than 9?
         BLE.S     DIGIT          ;no
         ADDQ.B    #7,D1          ;yes, must be A-F
DIGIT
         MOVE.B    D1,-[A0]       ;store char in message
         LSR.L     #4,D0          ;shift to next char
         SUBQ.B    #1,D2          ;done everything?
         BNE.S     LOOP           ;no
         JSR       JSR_ENTRY      ;yes, display message
         JMP $[PC]
         END
```

ENTRY - Upper Case Entry Point

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Allows the linker to define the label "ENTRY" as the absolute file entry point. A default run-time environment is initialized. A5 and A6 are cleared to all zeros, and A7 is set to the highest address of a 256-word stack.
Declaration:	not applicable.
Calling Sequence:	not applicable.
Parameters:	none.
Return Value:	none.
Remarks:	When executing the absolute file in the emulator the command "run from ENTRY" will initialize the run-time environment to default values and start the program executing at function "MAIN". If a different run-time environment is required then the user must provide his own version of "ENTRY". Upon return from function "MAIN" the message "end of program" will be displayed on the Status line of the display.

entry - Lower Case Entry Point

Specifications for this routine are as follows:

Library:	A5_LIB and ABS_LIB
Purpose:	Allows the linker to define the label "entry" as the absolute file entry point. A default run-time environment is initialized. A5 and A6 are cleared to all zeros, and A7 is set to the highest address of a 256-word stack.
Declaration:	not applicable.
Calling Sequence:	not applicable.
Parameters:	none.
Return Value:	none.
Remarks:	When executing the absolute file in the emulator the command "run from entry" will initialize the run-time environment to default values and start the program executing at function "main". If a different run-time environment is required then the user must provide his own version of "entry". Upon return from function "main" the message "end of program" will be displayed on the status line of the display.

Floating Point Operations

Floating point numbers, type float and type double, are implemented as IEEE floating point numbers. The library REAL_LIB:L68000 contains functions for manipulating IEEE floating point numbers. The C compiler handles operations on items of type float or type double by generating calls to value returning functions in this library.

Calls to the following functions are generated automatically by the C compiler whenever a floating point operation is encountered in the source. When the $SHORT_ARITH$ option is ON, the compiler will generate a call to the function expecting 32-bit IEEE format parameters. When the $SHORT_ARITH$ option is OFF, the compiler will generate a call to the function expecting 64-bit IEEE format parameters. Note that the functions expecting 32-bit IEEE format parameters cannot be replaced with functions written in the C language. C functions with parameters of type float expect to receive those parameters as type double, but the compiler treats the functions in REAL_LIB specially, allowing float type parameters to be passed without converting them to type double.

Zlongreal__add - double addition

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Add two 64-bit IEEE floating point numbers returning a 64-bit IEEE floating point sum.
Calling Sequence:	SUM = LEFT_TERM + RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	The address of the 64-bit sum is pushed onto the stack immediately after the second parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal_add - float addition

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Add two 32-bit IEEE floating point numbers returning a 32-bit IEEE floating point sum.
Calling Sequence:	SUM = LEFT_TERM + RIGHT_TERM;
Parameters:	Two 32-bit floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	The 32-bit floating point sum is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language, because parameters of type float are passed by the C compiler as type double.

Zlongreal_sub - double subtraction

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Subtract one 64-bit IEEE floating point number from another returning a 64-bit IEEE floating point difference.
Calling Sequence:	DIFFERENCE = LEFT_TERM - RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	The address of the 64-bit difference is pushed onto the stack immediately after the second parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal__sub - float subtraction

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Subtract one 32-bit IEEE floating point number from another returning a 32-bit IEEE floating point difference.
Calling Sequence:	DIFFERENCE = LEFT_TERM - RIGHT_TERM;
Parameters:	Two 32-bit floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	The 32-bit floating point difference is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__mul - double multiplication

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Multiply two 64-bit IEEE floating point numbers returning a 64-bit IEEE floating point product.
Calling Sequence:	PRODUCT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left factor is pushed first followed by the address of the right factor.
Return Value:	The address of the 64-bit product is pushed onto the stack immediately after the second parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal_mul - float multiplication

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Multiply two 32-bit IEEE floating point numbers returning a 32-bit IEEE floating point product.
Calling Sequence:	PRODUCT = LEFT_FACTOR * RIGHT_FACTOR;
Parameters:	Two 32-bit floating point values are pushed onto the stack. The left factor is pushed first followed by the right factor.
Return Value:	The 32-bit floating point product is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal_div - double division

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Divide one 64-bit IEEE floating point number by another returning a 64-bit IEEE floating point quotient.
Calling Sequence:	QUOTIENT = LEFT_FACTOR / RIGHT_FACTOR;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left factor is pushed first followed by the address of the right factor.
Return Value:	The address of the 64-bit quotient is pushed onto the stack immediately after the second parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal__div - float division

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Divide one 32-bit IEEE floating point number by another returning a 32-bit IEEE floating point quotient.
Calling Sequence:	QUOTIENT = LEFT_FACTOR / RIGHT_FACTOR;
Parameters:	Two 32-bit floating point values are pushed onto the stack. The left factor is pushed first followed by the right factor.
Return Value:	The 32-bit floating point quotient is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__neg - double negate

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Negate a 64-bit IEEE floating point number.
Calling Sequence:	RESULT = -TERM;
Parameters:	The address of the 64-bit value to be negated is pushed onto the stack.
Return Value:	The address of the 64-bit result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal__neg - float negate

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Negate a 32-bit IEEE floating point number.
Calling Sequence:	RESULT = -TERM;
Parameters:	The 32-bit value to be negated is pushed onto the stack.
Return Value:	The 32-bit result is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__equ - Compare items of type double for equality

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 64-bit IEEE floating point numbers for equality.
Calling Sequence:	LEFT_TERM == RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__equ - Compare items of type float for equality

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for equality.
Calling Sequence:	LEFT_TERM == RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__neq - Compare items of type double for inequality

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 64-bit IEEE floating point numbers for inequality.
Calling Sequence:	LEFT_TERM != RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__neq - Compare items of type float for inequality

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for inequality.
Calling Sequence:	LEFT_TERM != RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlong real__les - Compare items of type double for less than

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 64-bit IEEE floating point numbers for less than.
Calling Sequence:	LEFT_TERM < RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__les - Compare items of type float for less than

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for less than.
Calling Sequence:	LEFT_TERM < RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__gtr - Compare items of type double for greater than

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 64-bit IEEE floating point numbers for greater than.
Calling Sequence:	LEFT_TERM > RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to 0FFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__gtr - Compare items of type float for greater than

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for greater than.
Calling Sequence:	LEFT_TERM > RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to OFFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__leq - Compare items of type double for less than or equal to

specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	compare two 64-bit IEEE floating point numbers for less than or equal to.
Calling Sequence:	LEFT_TERM <= RIGHT_TERM
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to OFFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__leq - Compare items of type float for less than or equal to

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for less than or equal to.
Calling Sequence:	LEFT_TERM <= RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to OFFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__geq - Compare items of type double for greater than or equal to

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 64-bit IEEE floating point numbers for greater than or equal to.
Calling Sequence:	LEFT_TERM >= RIGHT_TERM;
Parameters:	The addresses of two 64-bit IEEE floating point values are pushed onto the stack. The address of the left term is pushed first followed by the address of the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 is set to OFFH.
Remarks:	The calling program will pop the parameters off the stack.

Zreal__geq - Compare items of type float for greater than or equal to

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compare two 32-bit IEEE floating point numbers for greater than or equal to.
Calling Sequence:	LEFT_TERM >= RIGHT_TERM;
Parameters:	Two 32-bit IEEE floating point values are pushed onto the stack. The left term is pushed first followed by the right term.
Return Value:	If the condition is false, the low order byte of data register D7 is set to zero. Otherwise, the low order byte of D7 ios set to OFFH.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zlongreal__float - Convert long to double

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a long integer to a 64-bit IEEE floating point number.
Parameters:	The 32-bit integer value to be converted is pushed onto the stack.
Return Value:	The address of the 64-bit IEEE floating point result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal__float - Convert long to float

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a long integer to a 32-bit IEEE floating point number.
Parameters:	The 32-bit integer value to be converted is pushed onto the stack.
Return Value:	The 32-bit IEEE floating point result is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal__trunc - Convert double to long

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a 64-bit IEEE floating point number to a 32-bit integer by truncation.
Parameters:	Th to be converted is pushed onto the stack.
Return Value:	The 32-bit integer result is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program.

Zreal__trunc - Convert float to long

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a 32-bit IEEE floating point number to a 32-bit integer by truncation.
Parameters:	The 32-bit floating point value to be converted is pushed onto the stack.
Return Value:	The 32-bit integer result is returned in data register D7.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zreal__extend - Convert float to double

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a 32-bit IEEE floating point number to a 64-bit IEEE floating point number.
Parameters:	The 32-bit floating point value to be converted is pushed onto the stack.
Return Value:	The address of the 64-bit IEEE floating point result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program. This function cannot be replaced with a function written in the C language because parameters of type float are passed by the C compiler as type double.

Zreal__contract - Convert double to float

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Convert a 64-bit IEEE floating point number to a 32-bit IEEE floating point number.
Parameters:	The address of the 64-bit IEEE floating point number to be converted is pushed onto the stack.
Return Value:	The 32-bit IEEE floating point result is returned in data register D7.
Remarks:	The parameters are popped off the s calling program.

The following functions are callable but must be declared as external functions.

Zlongreal__abs - Absolute Value

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Obtain the absolute value of a 64-bit IEEE floating point number.
Declaration:	extern double Zlongreal_abs();
Calling Sequence:	X = Zlongreal_abs (Y);
Parameters:	The address of a 64-bit IEEE floating point value whose absolute value is to be taken is pushed onto the stack.
Return Value:	The address of the 64-bit IEEE f result value is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal_atan - Arctangent

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the arctangent of the 64-bit IEEE floating point parameter.
Declaration:	extern double Zlongrea-
Calling Sequence:	X = Zlongreal_atan (Y);
Parameters:	The address of a 64-bit IEEE floating point arctangent result is pushed onto the stack.
Return Value:	The address for the 64-bit IEEE floating point arctangent result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal_cos - Cosine

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the cosine of a 64-bit IEEE floating point value measured in radians.
Declaration:	extern double Zlongreal_cos ();
Calling Sequency:	X = Zlongreal_cos (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack. The units of the parameter are radians.
Return Value:	The address for the 64-bit Ieee floating point cosine result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal__exp - Exponential

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the exponential of a 64-bit IEEE floating point value.
Declaration:	extern double Zlongreal_exp ();
Calling Sequence:	X = Zlongreal_exp (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack.
Return Value:	The address for the 64-bit IEEE floating point exponential result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal__ln - Natural logarithm

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the natural logarithm of a 64-bit IEEE floating point value.
Declaration:	extern double Zlongreal_ln ();
Calling Sequence:	X = Zlongreal_ln (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack.
Return Value:	The address for the 64-bit IEEE floating point immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal__sin - Sine

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the sine of a 64-bit IEEE floating point value measured in radians.
Declaration:	extern double Zlongreal_sin ();
Calling Sequence:	X = Zlongreal_sin (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack.
Return Value:	The address of the 64-bit IEEE floating point sine result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling program.

Zlongreal__round - Round a floating point number

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Round a double floating point value to a long integer.
Declaration:	extern long Zlongreal_round ();
Calling Sequence:	I = Zlongreal_round (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack.
Return Value:	A 32-bit integer representing the rounded value is returned in data register D7.
Remarks:	The parameter is popped off the stack by the calling program.

Zlongreal_sqrt - Square root

Specifications for this function are as follows:

Library:	REAL_LIB
Purpose:	Compute the square root of the parameter using Newton's method.
Declaration:	extern double Zlongreal_sqrt ();
Calling Sequence:	X = Zlongreal_sqrt (Y);
Parameters:	The address of a 64-bit IEEE floating point value is pushed onto the stack.
Return Value:	The address for the 64-bit IEEE floating point square root result is pushed onto the stack immediately after the parameter.
Remarks:	The parameters are popped off the stack by the calling routine.

Appendix A

Run-time Error Descriptions

When emulating with the monitor linked in, run-time errors will be displayed on the status line. The program will jump to the monitor and the message:

```
    68000---running . . . in monitor eeeeeeeeeeeeeeeeeeeee at XXXXXX
```

where: XXXXXX represents the next address after the call to the error routine, and eeeeeeeeeeeeeeeeeeeeeeis an explanation of the error. Following is a list of possible run-time errors.

Overflow

This error occurs when DEBUG is ON. An arithmetic operation caused an overflow.

Divide by zero

A division by zero was detected.

Illegal Instruction

The processor attempted to execute an illegal opcode.

Memory error #X

An error occurred during dynamic memory allocation (NEW, DISPOSE, MARK, and RELEASE). X indicates errors as follows:

0: Heap length too small (call INITHEAP with larger number for size of heap).

1: Heap has not been initialized (call INIHEAP before first use of NEW or MARK.

2: No free space in current mark. Space may exist in previous marks but is not available to the keyword NEW.

3: No block large enough to allocate but smaller blocks may exist.

4: Pointer variable points outside of heap.

5: No free space in heap.

6: Unable to mark, no block large enough.

7: Attempted to release mark that does not exist.

The address shown with a memory error gives the call to the memory routine, not the call to the error routine. The error number X is passed as a byte parameter to MEMERR.

<u>End of program</u>

Not actually an error; this message is displayed when the program terminates. No address is given. Zendprogram is jumped to instead of being called.

Index

The following index lists important terms and concepts of this manual along with the location(s) where 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.

* Appendixes - References to appendixes appear as "Appendix Y" where "Y" represents the letter designator of appendix.

* Figures/Tables - References to figures or tables are represented by the capital letter "F" or "T" followed by the appropriate number.

* Other entries in the index - Other entries in the index have their location indicated by page number.

64819-90902, MAY 1984
Replaces: 64819-90901, March 1983

HEWLETT PACKARD

PRINTED IN U.S.A.