64000

HP64000 Logic Development System

Model 64810A Pascal/64000 Compiler Supplement 8085

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 options, 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 environmental 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.

CW&A 2/81

FOLD HERE

HEWLETT PACKARD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

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

POSTAGE WILL BE PAID BY ADDRESSEE

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

FOLD HERE

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

READER COMMENT SHEET

Part Number: 64810-90905

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

1. The information in this book is complete:

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

2. The information in this book is accurate:

Too many errors	1 2 3 4 5	Exactly right

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

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

4. The Index and Table of Contents are useful:

Helpful	1 2 3 4 5	Missing or inadequate

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

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

6. What about the writing style:

Confusing	1 2 3 4 5	Clear

7. What about organization of the book:

Poor order	1 2 3 4 5	Good order

8. What about the size of the book:

too big/small	1 2 3 4 5	Right size

Comments: _____

_____

_____

_____

Particular pages with errors?

_____

Name (optional): _____

Job title: _____

Company: _____

Address: _____

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

Pascal/64000
Compiler Supplement
8085

LOGIC SYSTEMS DIVISION

COLORADO SPRINGS, COLORADO, U.S.A.

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 ..... July 1980 (Part Number 64810-90902)
Reprinted ..... December 1980
Second Edition ..... October 1983 (Part Number 64810-90905)

Table of Contents

Table of Contents (Cont'd)

List of Illustrations

List of Tables

Chapter 1

Pascal/64000 Compiler

8085

Introduction

This supplement contains a description of the processor dependent compiler features for the 8085 microprocessor. Description of various features and their use are also supplied. In addition, a discussion covering emulation of 8085 Pascal programs is provided.

This supplement is intended to be an extension of the Pascal/64000 Compiler Reference Manual.

NOTE

It is extremely important that the user reads and becomes familiar with the contents of Chapter 2, especially those paragraphs covering programming considerations.

Pascal Program Design

Pascal programs should be designed to be as processor and implementation independent as possible, yet certain concessions must be made when the processor has unique characteristics. Programs written to run on a large mainframe computer with megabytes of virtual memory may not run on an 8085 with a maximum of 64k-bytes of random access memory. Most large mainframe computer implementations have enough memory to allocate a stack area and a heap for dynamic memory allocation with no prompting by the user. In a limited memory system these factors must be communicated to the compiler in some manner. For the 8085, the user must specify the location of the stack and, if needed, the location of a memory pool for the dynamic allocation routines. The following sections describe subjects related to programming and compiling Pascal/64000 for the 8085 processor.

How to Implement a Program

The usual process of software generation is as follows:

a. Create a source program file using the editor.

b. Compile the source program.

c. Link the relocatable files.

d. Emulate the absolute file.

e. Debug as necessary.

This chapter will provide insight into each of these processes.

The Source File

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

```
"8085"
PROGRAM Name;
  .              {comments}
  .
CONST
    ...;
    ...;
TYPE
    ...;
    ...;
VAR
    ...;
    ...;
PROCEDURE Procedure_name(Parameter1 : Type);
    .
    .
    BEGIN
      .
      .
      .
    END;
 BEGIN
  .
  .              {main program code}
  .
 END.
```

After completing the source program, it is ready for compilation. Notice in the example form that the first line of the source program specifies the 8085 processor. This first line must be the special compiler directive indicating the processor for which the program was written. The compiler only recognizes upper-case keywords, but identifiers may be upper or lower case.

A sample compiler command would appear as follows:

```
compile <FILE_name> listfile <FILE_list> options xref
```

The compiler may produce up to four files as output: a relocatable file, a listing file (if specified), an assembly file (if specified), and an assembler symbol file (if specified). Descriptions of these files are as follows:

Assembly file:	If an assembly file is specified by the option ASM_FILE anywhere in the source file, an assembly file, ASM8085, will be created in the current userid. This file will contain the assembly language source equivalent of the Pascal program being compiled with the Pascal language source intermixed as comments. This file may be assembled by the assembler.
Assembler symbol file:	If no errors were detected in the source file, an assembler symbol file (called <FILE_name>:asmb_sym) will be created. This file contains the symbolic information useful for program debugging during emulation.
Relocatable file:	If no errors were detected in the source file (called <FILE_name>:source), a relocatable file (called <FILE_name>:reloc) will be created. This file will be used by the linker to create an executable absolute file.
Listing file:	If a listfile is specified when the compiler is evoked, a file <FILE_list> containing source lines with line numbers, program counter, level numbers, errors and expanded code (if specified) will be generated.

Linking

After program modules have been compiled (assembled), the modules may be linked to form an executable absolute file. The compiler generates calls to a set of library routines for commonly used operations such as multiply, divide, comparisons, array referencing, etc. These routines must be linked with the program modules. There are two libraries which may be linked. The first is a debug library file called DLIB8085:HP.

This library of relocatables contains some extra code to detect errors such as division by 0, or underflow and overflow on multiplication.

The second library is called LIB8085:HP. This library, which has only a limited set of error-detection code, should execute slightly faster and take up less space in memory. This library may be linked in place of the debug library after reasonable assurance that the code is error free.

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

link ...

Object files: SETSTACK_,MODULE1,MODULE2

Library files: DLIB8085:HP

.
.
.

Absolute file name: PROGRAM

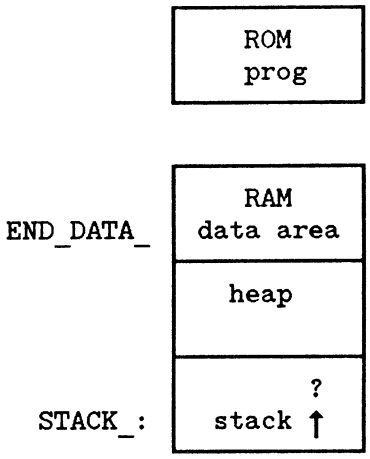
In the link listfile, the library routines that are referenced by the compiled code are linked at the end of the last user relocatable PROG and/or DATA areas. This fact must be considered for the proper choice of the stack pointer location, and PROG and DATA link addresses.

Only the necessary routines are linked from the library. Careful programming may sometimes eliminate the necessity of loading some of the library routines.

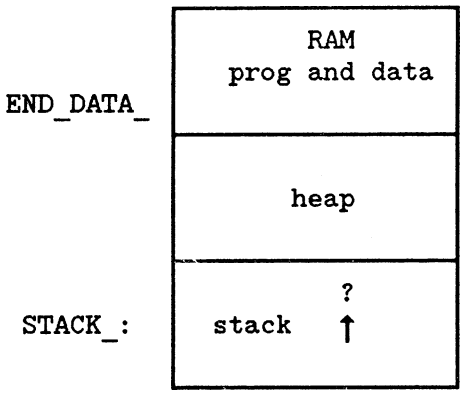
Emulation of Pascal Programs

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

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



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



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

```
run from NAME
```

Debugging with DLIB8085:HP Library

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

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

```
display local_symbols_in Derrors:HP
```

The list will roll off the screen; do not stop it with the reset key since the information which rolls off is not important. When the list has stopped, scan the upper portion of the list for the address at which the illegal opcode occurred. The error type will be listed at the left of this address. A detailed description of the run time errors is given in Appendix A of this manual. When using the library LIB8085:HP, the same list will be obtained by entering the command:

```
display local_symbols_in Zerrors:HP
```

The display will appear as follows:

NOTE

The addresses will change, depending on the link.

Label	Address	Data	
Z_END_PROGRAM	1242H	C3H	
Z_ERR_1FUTURE	1270H	22H	Scan this portion for the address where the illegal opcode occurred. The data field in this portion is not significant.
Z_ERR_CASE	1258H	08H	
Z_ERR_DIV_BY_0	124AH	08H	
Z_ERR_HEAP	1268H	08H	
Z_ERR_OVERFLOW	123CH	08H	
Z_ERR_SET_CONV	1251H	08H	
Z_ERR_UNDERFLOW	1243H	08H	
Z_PSW_FLAGS	1296H	89H	
Z_REG_A	1297H	8FH	
Z_REG_B	1299H	F6H	
Z_REG_C	1298H	F5H	The data field in this portion may contain useful information. The addresses in this portion are not significant.
Z_REG_D	129BH	F0H	
Z_REG_E	129AH	55H	
Z_REG_H	129DH	EDH	
Z_REG_L	129CH	E1H	
Z_ZCALLER_H	1295H	69H	
Z_ZCALLER_L	1294H	A0H	

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

NOTE

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

Chapter 2

Pascal/64000 Programming

Programming Considerations

Introduction

This chapter covers some important requirements of the run time environment for 8085 Pascal/64000 programs. Although some requirements may not be necessary for every program, the programmer should become familiar with the information supplied in order to use it when the structure of a 8085 program requires it. The specific areas to be discussed are stack pointer initialization, multiple module programs, heap initialization for use with dynamic memory routines (NEW, DISPOSE, MARK, and RELEASE), interrupt processing with Pascal programs, routine internal structure, and optional code generation.

Stack Pointer Initialization

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

```
EXT STACK_
LXI SP, STACK_
```

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

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

a. If the DLIB8085:HP library is linked, the stack will be assigned 128 bytes in the program (PROG) area of the linked modules.

b. If the LIB8085:HP library is linked, the stack will be assigned 128 bytes in the data (DATA) area of the linked modules.

NOTE

Whenever the LIB8085:HP library is linked, a DATA area location must be specified.

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

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

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


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

The compiler will generate relocatable code which sets the stack pointer to the address of STACK_ (4000H in this example), and equate an area of 128 bytes (3F80H..3FFFH) for the stack.

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

```
"8085"
            GLB STACK-
STACK_      EQU 2000H     ;puts initial stack
            .             ; pointer at 2000H
            .
            .
```

or:

```
"8085"
            GLB STACK-
            DATA
STACKBOT    DS  <stacksize>  ;puts stack
                             ; storage in the
STACK_:                      ; DATA area of
            .                ; the program
            .
            .
```

This file may then be linked with the other program modules generated by the compiler as follows:

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

Multiple Module Programs

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

Example:

```
(file MODULE1:source)

    PROGRAM MODULE1; {this is the main module}

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

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

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

NOTE

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

```
(file MODULE2:source)

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

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

Dynamic Allocation Heap Initialization

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

```
INITHEAP(Start_address,Length_in_bytes : INTEGER)
```

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

```
PROGRAM Test;

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

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

Example:

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

This example would reserve 40 bytes for the stack and the remainder of the memory from the end of the DATA area to the bottom of the stack (SP-40) for the dynamic allocation routines. This implies that the stack is in a contiguous block with the DATA area.

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

Interrupt Vector Handling

An interrupt to the 8085 by way of the INTR, TRAP, RST5.5, RST6.5, or RST7.5 line may direct the processor to begin execution of Pascal procedures designed to handle the interrupts. A typical example would be to use the RST 7 instruction on interrupt. Placing a call to the global PROCEDURE Interrupt, in address location 38H, would handle the interrupt and then return to location 3BH which could be a RET to the Pascal code that was interrupted. The following is an example of this type of operating environment:

Assembly code file:

```
"8085"
    EXT Interrupt   ;interrupt vector to PROCEDURE
                    ;  interrupt

    ORG 38H         ;address of RST 7
    CALL Interrupt  ;handle interrupt
    RET             ;go back to what you were doing
```

Register Save onto Stack. The following routine will put the current register information onto the stack to be retrieved later by RES_REG. Only A,B,C,D,E,H, and L are saved (all that needs to be saved if the interrupt handler is a Pascal procedure).

```
            GLB  SAVE_REG
SAVE_REG    XTHL             ;put HL on stack for return to caller
            PUSH  D          ;save DE
            PUSH  B          ;save BC
            PUSH  PSW        ;save status and accumulator
            PCHL             ;return to caller
```

Register Restoration from Stack. The following routine gets the register information off the stack that was current at the time of call to SAVE_REG:

```
            GLB RES_REG
RES_REG     POP H            ;get return to caller to HL
            POP PSW          ;restore status and accumulator
            POP B            ;restore BC
            POP D            ;restore DE
            XTHL             ;put return to caller on stack, restore HL
            RET              ;return to caller
```

The Pascal module might look like the following:

```
PROGRAM INTERRUPT_HANDLER;


PROCEDURE SAVE_REG;EXTERNAL;
PROCEDURE RES_REG;EXTERNAL;

$GLOBPROC ON$
  PROCEDURE Interrupt;
  BEGIN
    SAVE_REG;          {save registers}
    .
    .                  {code to handle interrupt goes here}
    .
    RES_REG;           {restore registers}
  END;                 {go back to what was happening before
                        interrupt}

$GLOBPROC OFF$
```

Parameter Passing Restrictions

Parameter passing for the Pascal/64000 run time environment in the 8085 processor is done by address reference. Thus the calling routine indicates the location of the actual parameters as a list of addresses following the procedure or function call. In order to permit certain special cases (such as VAR parameters of one procedure being passed as

parameters to another routine and certain variable address expressions) where the address in the parameter list represents a pointer to the actual parameter, the parameter passing routines (PARAM_ and RPARAM_) interpret the address 0000H as being an indirection indicator and will look at the next parameter address as a pointer to the actual parameter address. As a result of this interpretation for the address 0000H, it is not possible to pass a variable located at address 0000H as an actual parameter in a procedure or function call. The user is warned not to allocate any variable at absolute address 0000H (all other addresses present no problem). Programs which attempt to pass a parameter variable located at absolute 0000H will not work correctly. Further information about the details of parameter passing and the procedures PARAM_ and RPARAM_ is provided in the Run-time Library discussion in Chapter 3.

Routine Internal Structure

Programs, procedures, and functions are the basic blocks of Pascal program structure. Each of these routine types has a similar structure in the 8085 code generator. A routine is generally composed of a code area (including the entry point, code and an exit point), a data area and two constant areas. The 8085 compiler allocates each of these areas as relocatable blocks of data normally assigned to the PROG relocation area. If the $SEPARATE$ option is in effect, the data area is assigned to the DATA relocation area and the code and constant blocks are assigned to the PROG relocation area.

The code area contains the entry point defined by a local or global label, followed by the code required to perform the routine's function. In Pascal a routine can have only one entry point and it will always return from one exit point.

The data area is the memory block where the routine's local variables and parameters are allocated. A function also needs to allocate room in the data area for the temporary copy of the function result. Finally the data area contains space for temporary values needed by the code generator to evaluate expressions which can not be computed in registers alone.

The two constant areas are memory blocks where constants unique to a routine are specified. The F constant area contains the dope vectors required for routines with parameters or compiled with the $RECURSIVE ON$ option and creating calls to the run time routines: PARAM_, RPARAM_, RENTRY_ and REXIT_.

The C constant area, labeled Cproc, will contain the dope vectors for any array references requiring the run time library routines ARRAY_ and ARRAYN_. Constants being passed as value parameters will be defined in the Cprog area.

Compiler Internal Label Conventions

The construction of internal labels within the compiler generated code is discussed in Appendix C of the Pascal/64000 Compiler Manual. For the 8085 code generator, every procedure has associated with it each of the labels described in the preceding paragraphs (i.e. the entry label, return label, data area label, and an end label). In addition, a 8085 procedure can have the two constant area labels marking the areas needed for local constants and dope vectors.

In summary, for a procedure named "test", the 8085 compiler would create the labels: test, Rtest, Ctest, Ftest, Dtest, and Etest. For the sample program listed in figure 2-1, the compiler generated labels are summarized as follows:

Compiler Generated Label	Program Counter	Label Description
Assign	0000H	Procedure entry
RAssign	0033H	Return label
CAssign	0034H	Constant area C
FAssign	0036H	Constant area F
DAssign	0044H	Data area
EAssign	0056H	End of procedure
SAME_function	0057H	Procedure entry
RSAME_function	0098H	Return label
CSAME_function	0099H	Constant area C
FSAME_function	0099H	Constant area F
DSAME_function	00A7H	Data area
ESAME_function	00BAH	End of procedure
PF_sample	00BBH	Program entry
PF_samp00_1	00DFH	Compiler generated label
RPF_sample	00DFH	Return label
CPF_sample	00E2H	Constant area C
FPF_sample	00ECH	Constant area F
DPF_sample	00ECH	Data area
EPF_sample	00F6H	End of procedure

Figure 2-1 shows a source listing for a simple program. Figure 2-2 shows the expanded source listing for this program, indicating the use of internal compiler labels.

```
FILE: PF_sample:T8085   HP 64000 - Pascal     "8085" Code Generator


   1 0000  1  "8085"
   2 0000  1  PROGRAM PF_sample;
   3 0000  1  $EXTENSIONS$
   4 0000  1  TYPE BIG_type= ARRAY [0..1,0..3] OF BYTE;
   5 0000  1  VAR
   6 0000  1    Byte    :BYTE;
   7 0001  1    Integer :INTEGER;
   8 0003  1    Big_one :BIG_type;
   9 0000  1
  10 0000  1  PROCEDURE EXTproc(I:INTEGER); EXTERNAL;
  11 0000  1
  12 0000  1  PROCEDURE Assign(B1:BYTE;     VAR B2:BYTE;
  13 0003  2                   I1:INTEGER;  VAR I2:INTEGER;
  14 0007  2                   X1:BIG_type; VAR X2:BIG_type);
  15 0011  2   VAR DUMMY_local_var:INTEGER;
  16 0009  2   BEGIN
  17 0009  2    DUMMY_local_var:=0;
  18 000F  2    EXTproc(1234H);
  19 0014  2    B2:= B1;
  20 001B  2    I2:= I1;
  21 0026  2    X2:= X1;
  22 0033  2   END;
  23 0000  1
  24 0000  1  FUNCTION SAME_function (B1:BYTE;     VAR B2:BYTE;
  25 0003  2                          I1:INTEGER;  VAR I2:INTEGER;
  26 0007  2                          X1:BIG_type; VAR X2:BIG_type) :BOOLEAN;
  27 0012  2   VAR DUMMY_local_var :INTEGER;
  28 0060  2   BEGIN
  29 0060  2    DUMMY_local_var:=1;
  30 0066  2    SAME_function:= (B2=B1) AND (I2=I1) AND (X2=X1);
  31 0097  2   END;
  32 00BB  1
  33 00BB  1  BEGIN   {Main program: PF_sample}
  34 00BE  1   IF NOT SAME_function (Byte,Byte,Integer,Integer,Big_one,Big_one)
  35 00D0  1     THEN  Assign(Byte,Byte,Integer,Integer,Big_one,Big_one);
  36 00DF  1  END.


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

Figure 2-1. Internal Structure Source Listing

```
FILE: PF_sample:T8085   HP 64000 - Pascal     Expanded 8085 listing

  1 0000  1  "8085"
  2 0000  1  PROGRAM PF_sample;
               0000                  NAME  "PF_sample Pascal"

  3 0000  1  $EXTENSIONS$
  4 0000  1  TYPE BIG_type= ARRAY [0..1,0..3] OF BYTE;
  5 0000  1  VAR
  6 0000  1    Byte    :BYTE;
  7 0001  1    Integer :INTEGER;
  8 0003  1    Big_one :BIG_type;
  9 0000  1
 10 0000  1  PROCEDURE EXTproc(I:INTEGER); EXTERNAL;
               0000                  EXT   EXTproc

 11 0000  1
 12 0000  1  PROCEDURE Assign(B1:BYTE;     VAR B2:BYTE;

 13 0003  2                  I1:INTEGER;  VAR I2:INTEGER;
 14 0007  2                  X1:BIG_type; VAR X2:BIG_type);
 15 0011  2   VAR DUMMY_local_var:INTEGER;
               0000            Assign:
               0000  01  ????        LXI   B,FAssign
               0003  11  ????        LXI   D,DAssign
               0006  CD  ????        CALL  PARAM_

 16 0009  2   BEGIN
 17 0009  2    DUMMY_local_var:=0;
               0009  21  0000        LXI   H,0
               000C  22  ????        SHLD  DAssign+17

 18 000F  2    EXTproc(1234H);
               000F  CD  ????        CALL  EXTproc
               0012      ????        DW    CAssign

 19 0014  2    B2:= B1;
               0014  3A  ????        LDA   DAssign
               0017  2A  ????        LHLD  DAssign+1
               001A  77              MOV   M,A

 20 001B  2    I2:= I1;
               001B  2A  ????        LHLD  DAssign+5
               001E  EB              XCHG
               001F  2A  ????        LHLD  DAssign+3
               0022  EB              XCHG
               0023  73              MOV   M,E
               0024  23              INX   H
               0025  72              MOV   M,D
```

Figure 2-2. Internal Structure Expanded Listing

```
FILE: PF_sample:T8085   HP 64000 - Pascal     Expanded 8085 listing

 21 0026  2    X2:= X1;
               0026  2A  ????        LHLD  DAssign+15
               0029  11  ????        LXI   D,DAssign+7
               002C  EB              XCHG
               002D  01  0800        LXI   B,8
               0030  CD  ????        CALL  MOVEB_

 22 0033  2   END;
               0033            RAssign:
               0033  C9              RET
               0034            CAssign:
               0034      3412        DW    4660
               0036            FAssign:
               0036      0600        DW    6
               0038      0100        DW    1
               003A      FEFF        DW    -2
               003C      0200        DW    2
               003E      FEFF        DW    -2
               0040      0800        DW    8
               0042      FEFF        DW    -2
               0044            DAssign:
               0044                  DS    19
                               EAssign EQU $-1

 23 0000  1
 24 0000  1  FUNCTION SAME_function (B1:BYTE;     VAR B2:BYTE;

 25 0003  2                          I1:INTEGER;  VAR I2:INTEGER;
 26 0007  2                          X1:BIG_type; VAR X2:BIG_type) :BOOLEAN;
 27 0012  2   VAR DUMMY_local_var :INTEGER;
               0057            SAME_function:
               0057  01  ????        LXI   B,FSAME_function
               005A  11  ????        LXI   D,DSAME_function
               005D  CD  ????        CALL  PARAM_

 28 0060  2   BEGIN
 29 0060  2    DUMMY_local_var:=1;
               0060  21  0100        LXI   H,1
               0063  22  ????        SHLD  DSAME_function+18

 30 0066  2    SAME_function:= (B2=B1) AND (I2=I1) AND (X2=X1);
               0066  2A  ????        LHLD  DSAME_function+5
               0069  5E              MOV   E,M
               006A  23              INX   H
               006B  56              MOV   D,M
```

Figure 2-2. Internal Structure Expanded Listing (Cont'd)

```
FILE: PF_sample:T8085   HP 64000 - Pascal     Expanded 8085 listing

              006C  2A  ????        LHLD  DSAME_function+3
              006F  CD  ????        CALL  Zintequ
              0072  2A  ????        LHLD  DSAME_function+1
              0075  47              MOV   B,A
              0076  3A  ????        LDA   DSAME_function
              0079  BE              CMP   M
              007A  C2  8200        JNZ   $+8
              007D  AF              XRA   A
              007E  3C              INR   A
              007F  C3  8300        JMP   $+4
              0082  AF              XRA   A
              0083  A0              ANA   B
              0084  2A  ????        LHLD  DSAME_function+15
              0087  11  ????        LXI   D,DSAME_function+7
              008A  EB              XCHG
              008B  F5              PUSH  PSW
              008C  01  0800        LXI   B,8
              008F  CD  ????        CALL  EQUMB_
              0092  E1              POP   H
              0093  A4              ANA   H
              0094  32  ????        STA   DSAME_function+17

 31 0097  2   END;
              0097  B7              ORA   A
              0098            RSAME_function:
              0098  C9              RET
              0099            CSAME_function:
              0099            FSAME_function:
              0099      0600        DW    6
              009B      0100        DW    1
              009D      FEFF        DW    -2
              009F      0200        DW    2
              00A1      FEFF        DW    -2
              00A3      0800        DW    8
              00A5      FEFF        DW    -2
              00A7            DSAME_function:
              00A7                DS    20
                              ESAME_function EQU $-1
              00BB            PF_sample:

 32 00BB  1
 33 00BB  1  BEGIN   {Main program: PF_sample}
              00BB  31  ????        LXI   SP,STACK_
```

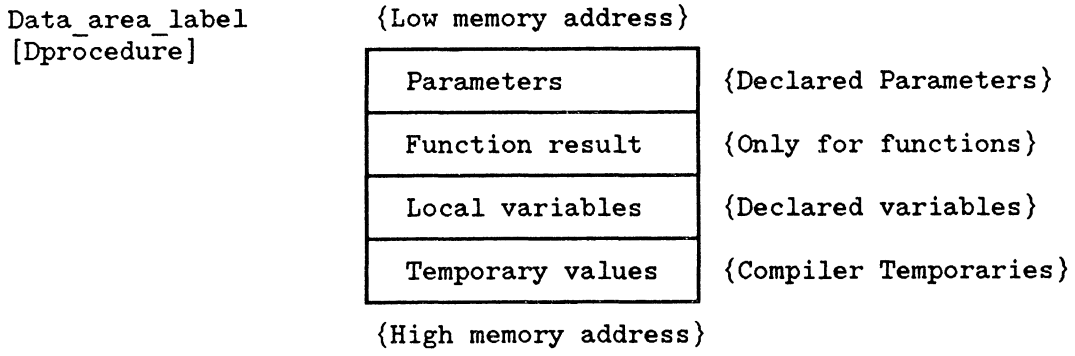
Figure 2-2. Internal Structure Expanded Listing (Cont'd)

```
FILE: PF_sample:T8085   HP 64000 - Pascal     Expanded 8085 listing

 34 00BE  1   IF NOT SAME_function (Byte,Byte,Integer,Integer,Big_one,Big_one)
              00BE  CD  ????        CALL  SAME_function
              00C1      ????        DW    DPF_sample
              00C3      ????        DW    DPF_sample
              00C5      ????        DW    DPF_sample+1
              00C7      ????        DW    DPF_sample+1
              00C9      ????        DW    DPF_sample+3
              00CB      ????        DW    DPF_sample+3
              00CD  C2  ????        JNZ   PF_samp00_1

 35 00D0  1    THEN  Assign(Byte,Byte,Integer,Integer,Big_one,Big_one);
              00D0  CD  ????        CALL  Assign
              00D3      ????        DW    DPF_sample
              00D5      ????        DW    DPF_sample
              00D7      ????        DW    DPF_sample+1
              00D9      ????        DW    DPF_sample+1
              00DB      ????        DW    DPF_sample+3
              00DD      ????        DW    DPF_sample+3
              00DF            PF_samp00_1:

 36 00DF  1  END.
              00DF            RPF_sample:
              00DF  C3  ????        JMP   Z_END_PROGRAM
              00E2            CPF_sample:
              00E2      0200        DW    2
              00E4      0000        DW    0
              00E6      0400        DW    4
              00E8      0100        DW    1
              00EA      0000        DW    0
              00EC            FPF_sample:
              00EC            DPF_sample:
              00EC                  DS    11
                              EPF_sample EQU $-1
              00F7                  GLB   PF_sample
              00F7                  GLB   RPF_sample
              00F7                  GLB   EPF_sample
              00F7                  EXT   Zintequ
              00F7                  EXT   PARAM_
              00F7                  EXT   MOVEB_
              00F7                  EXT   STACK_
              00F7                  EXT   EQUMB_
              00F7                  EXT   Z_END_PROGRAM
              00F7                  END   PF_sample


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

Figure 2-2. Internal Structure Expanded Listing (Cont'd)

Data Variable Allocation

The allocation of variables to the data area of a routine is always in the order: parameters followed by function result (if required), followed by local variables, followed by temporary storage.



Procedures and functions pass parameters in the same way using the generalized parameter passing method using dope vectors described in detail in Chapter 3.

The expanded compiler listing in figure 2-2 is intended to show the memory allocation of data areas and the parameter passing method for procedures and functions. A descriptive summary of the data area for PROCEDURE Assign, FUNCTION SAME_function and main PROGRAM PF_sample is provided to help interpret the listing.

Table 2-1. PROCEDURE Assign Data Area Description Summary

DAssign		DS 19	; 19 bytes	
Program Counter {HEX}	Data_area Offset {HEX}	Size {Bytes}	Name Identifier	Description
0044	0000	1	B1	Byte parameter
0045	0001	2	B2	VAR byte parameter
0047	0003	2	I1	Integer parameter
0049	0005	2	I2	VAR integer parameter
004B	0007	8	X1	BIG_type parameter
0053	000F	2	X2	VAR BIG_type parameter
0055	0011	2	DUMMY_local_var	Integer variable

Table 2-2. FUNCTION SAME_function Data Area Description Summary

DSAME_function	DS 20	; 20 bytes		
Program Counter (HEX)	Data_area Offset (HEX)	Size (Bytes)	Name Identifier	Description
00A7	0000	1	B1	Byte parameter
00A8	0001	2	B2	VAR byte parameter
00AA	0003	2	I1	Integer parameter
00AC	0005	2	I2	VAR integer parameter
00AE	0007	8	X1	BIG_type parameter
00B6	000F	2	X2	VAR BIG_type parameter
00B8	0011	1	SAME_function	Boolean function return value
00B9	0012	2	DUMMY_local_var	Integer variable

Table 2-3. PROGRAM PF_sample Data Area Description Summary

DPF_sample	DS 11	; 11 bytes		
Program Counter (HEX)	Data_area Offset (HEX)	Size (Bytes)	Name Identifier	Description
00EC	0000	1	Byte	Byte variable
00ED	0001	2	Integer	Integer variable
00EF	0003	8	Big_one	BIG_type variable

Optional Code Generation

ASM_FILE

The compiler option ASM_FILE will produce a source file of the 8085 assembler code equivalent to the original program. This assembler source file will be created in filename:

```
ASM8085 [:current userid]
```

The remainder of this section briefly describes the effects of three compiler options on the code produced by the compiler. None of the other compiler options result in a significantly different code sequence. Table 2-4 summarizes the illustrations, example names, and the options for each compilation example.

Table 2-4. Optional Code Generation Illustration Summary

Figure	Name	Option		
Figure 2-3	OPT1	RECURSIVE ON	OPTIMIZE OFF	DEBUG OFF
Figure 2-4	OPT2	RECURSIVE ON	OPTIMIZE ON	DEBUG OFF
Figure 2-5	OPT3	RECURSIVE OFF	OPTIMIZE OFF	DEBUG OFF
Figure 2-6	OPT4	RECURSIVE OFF	OPTIMIZE OFF	DEBUG ON

Option 1 - $RECURSIVE ON, OPTIMIZE OFF, DEBUG OFF$

The $RECURSIVE$ option permits the programmer to specify recursive or static procedures and functions. Since local variables in procedures and functions are always allocated into a fixed relocatable memory area, special entry and exit codes are required to permit recursive calls. Figure 2-3 shows the code produced for a simple procedure with two parameters. Because of the additional time required to save the existing local variables and parameters, the compiler generates extra code to test for the first call to PROCA. If it is the first call there is no need to save the current (undefined) data in PROCA.

The example listing for OPT1 shows the expanded assembly listing of PROCA with $RECURSIVE ON, OPTIMIZE OFF$. Note the additional code at the entry and exit of PROCA to check for the first calling of the procedure. Since the total time required for the routine RENTRY_ and REXIT_ is approximately equivalent to that for the parameter passing, the program will execute somewhat faster and use less stack area when compiled in this manner.

```
FILE: OPT1:T8085        HP 64000 - Pascal     RECURSIVE ON,OPTIMIZE OFF,DEBUG OFF

  7 0000  1  {Initialized options} $RECURSIVE ON,OPTIMIZE OFF,DEBUG OFF$
  8 0000  1
  9 0000  1   PROCEDURE PROCA(VAR I: BYTE; VALUEP: BYTE);
              0000            PROCA:
              0000  2A  ????        LHLD  DPROCA+3          *CALL_COUNT test code:
              0003  7D              MOV   A,L               * Test for first call
              0004  B4              ORA   H                 *  IF CALL_COUNT<>0 THEN
              0005  CA  1100        JZ    $+12              *    optional call to RENTRY_
              0008  01  ????        LXI   B,CPROCA          #
              000B  11  ????        LXI   D,DPROCA          #  RENTRY_call
              000E  CD  ????        CALL  RENTRY_           #
              0011  2A  ????        LHLD  DPROCA+3          *
              0014  23              INX   H                 *Increment CALL_COUNT
              0015  22  ????        SHLD  DPROCA+3          *
              0018  01  ????        LXI   B,FPROCA          &
              001B  11  ????        LXI   D,DPROCA          & RPARAM_ call
              001E  21  0300        LXI   H,3               &  for parameter passing
              0021  CD  ????        CALL  RPARAM_           &

 10 0024  2    BEGIN
 11 0024  2     I:= VALUEP+VALUEP;
              0024  3A  ????        LDA   DPROCA+2
              0027  87              ADD   A
              0028  2A  ????        LHLD  DPROCA
              002B  77              MOV   M,A

 12 002C  2    END;
              002C  2A  ????        LHLD  DPROCA+3          *
              002F  2B              DCX   H                 *Decrement CALL_COUNT
              0030  22  ????        SHLD  DPROCA+3          *
              0033  7D              MOV   A,L               * Test for first call
              0034  B4              ORA   H                 *  If CALL_COUNT<>0 then
              0035  CA  4100        JZ    $+12              *    optional call to REXIT_
              0038  01  ????        LXI   B,CPROCA          %
              003B  11  ????        LXI   D,DPROCA          % REXIT_ call
              003E  CD  ????        CALL  REXIT_            %
              0041            RPROCA:
              0041  C9              RET
              0042            CPROCA:
              0042      0500        DW    5
              0044            FPROCA:
              0044      0200        DW    2
              0046      FEFF        DW    -2
              0048      0100        DW    1
              004A            DPROCA:
              004A                  DS    3
              004D      0000        DW    0
                              EPROCA EQU $-1
```

Figure 2-3. OPT1 Listing Example

Option 2 - $RECURSIVE ON, OPTIMIZE ON, DEBUG OFF$

Figure 2-4 shows the same procedure used in Figure 2-3 but compiled with options $RECURSIVE ON, OPTIMIZE ON$. In this mode the compiler does not generate the CALL_COUNT code for the detection of the first procedure call; however, it does generate the calls to RENTRY_ and REXIT_ in all cases. This saves 27 bytes of code within the recursive procedure PROCA, but adds the additional execution time to copy and restore the local variables in each call and will also utilize more stack memory than the example in Figure 2-3.

```
FILE: OPT2:T8085        HP 64000 - Pascal     RECURSIVE ON,OPTIMIZE ON ,DEBUG OFF

  7 0000  1                      $RECURSIVE ON,OPTIMIZE ON ,DEBUG OFF$
  8 0000  1
  9 0000  1   PROCEDURE PROCA(VAR I: BYTE; VALUEP: BYTE);
              0000            PROCA:
              0000  01  ????        LXI   B,CPROCA
              0003  11  ????        LXI   D,DPROCA
              0006  CD  ????        CALL  RENTRY_
              0009  01  ????        LXI   B,FPROCA
              000C  11  ????        LXI   D,DPROCA
              000F  21  0300        LXI   H,3
              0012  CD  ????        CALL  RPARAM_

 10 0015  2    BEGIN
 11 0015  2     I:= VALUEP+VALUEP;
              0015  3A  ????        LDA   DPROCA+2
              0018  87              ADD   A
              0019  2A  ????        LHLD  DPROCA
              001C  77              MOV   M,A

 12 001D  2    END;
              001D  01  ????        LXI   B,CPROCA
              0020  11  ????        LXI   D,DPROCA
              0023  CD  ????        CALL  REXIT_
              0026            RPROCA:
              0026  C9              RET
              0027            CPROCA:
              0027      0300        DW    3
              0029            FPROCA:
              0029      0200        DW    2
              002B      FEFF        DW    -2
              002D      0100        DW    1
              002F            DPROCA:
              002F                  DS    3
                              EPROCA EQU $-1
```

Figure 2-4. OPT2 Listing Example

Option 3 - $RECURSIVE OFF, OPTIMIZE OFF, DEBUG OFF$

Figure 2-5 shows the same procedure PROCA compiled with $RECURSIVE OFF$ option. This procedure is 23 bytes shorter than the example in Figure 2-4 and 50 bytes shorter than the example in Figure 2-3. Since the routines RENTRY_ and REXIT_ are not called and the static parameter passer PARAM_ (rather than RPARAM_) is called, the execution time overhead for a call to PROCA in Figure 2-5 is about one-half that of Figure 2-3 and about one-quarter of Figure 2-4.

```
FILE: OPT3:T8085        HP 64000 - Pascal     RECURSIVE OFF,OPTIMIZE OFF,DEBUG OFF


  7 0000  1                      $RECURSIVE OFF,OPTIMIZE OFF,DEBUG OFF$
  8 0000  1
  9 0000  1   PROCEDURE PROCA(VAR I: BYTE; VALUEP: BYTE);
              0000            PROCA:
              0000  01  ????        LXI   B,FPROCA
              0003  11  ????        LXI   D,DPROCA
              0006  CD  ????        CALL  PARAM_

 10 0009  2    BEGIN
 11 0009  2     I:= VALUEP+VALUEP;
              0009  3A  ????        LDA   DPROCA+2
              000C  87              ADD   A
              000D  2A  ????        LHLD  DPROCA
              0010  77              MOV   M,A

 12 0011  2    END;
              0011            RPROCA:
              0011  C9              RET
              0012            CPROCA:
              0012            FPROCA:
              0012      0200        DW    2
              0014      FEFF        DW    -2
              0016      0100        DW    1
              0018            DPROCA:
              0018                  DS    3
                              EPROCA EQU &-1
```

Figure 2-5. OPT3 Listing Example

Option 4 - $RECURSIVE OFF, OPTIMIZE OFF, DEBUG ON$

Figure 2-6 illustrates the additional library call for the byte add operation which is generated with option $DEBUG ON$. This option, used with the debug library DLIB8085:HP, will allow the programmer to detect add operations which cause overflow.

```
FILE: OPT4:T8085        HP 64000 - Pascal     RECURSIVE OFF,OPTIMIZE OFF,DEBUG ON

  7 0000  1                      $RECURSIVE OFF,OPTIMIZE OFF,DEBUG ON $
  8 0000  1
  9 0000  1   PROCEDURE PROCA(VAR I: BYTE; VALUEP: BYTE);
              0000            PROCA:
              0000  01  ????        LXI   B,FPROCA
              0003  11  ????        LXI   D,DPROCA
              0006  CD  ????        CALL  PARAM_

 10 0009  2    BEGIN
 11 0009  2     I:= VALUEP+VALUEP;
              0009  3A  ????        LDA   DPROCA+2
              000C  6F              MOV   L,A
              000D  CD  ????        CALL  Zbyteadd
              0010  2A  ????        LHLD  DPROCA
              0013  77              MOV   M,A

 12 0014  2    END;
              0014            RPROCA:
              0014  C9              RET
              0015            CPROCA:
              0015            FPROCA:
              0015      0200        DW    2
              0017      FEFF        DW    -2
              0019      0100        DW    1
              001B            DPROCA:
              001B                  DS    3
                              EPROCA EQU $-1
```

Figure 2-6. OPT4 Listing Example

Chapter 3

Run-time Library Specifications

General

This chapter describes the run-time library needed to execute Pascal programs compiled by the Pascal/64000 compiler for the 8085 microprocessor.

The library is logically divided into two groups of routines. One group contains the standard library procedures and functions available to all Pascal/64000 programs, independent of the actual processor. The second group supplies the elementary routines that supplement the standard 8085 instruction set. Tables 3-1 and 3-2 list the standard and supplemental routines for the 8085 microprocessor.

Table 3-1. Pascal Library Routines (Standard)

Name	Purpose	Ref Page
ARRAY_	Compute address of array element	3-3
ARRAYN_	Compute address of array vector	3-4
COMPAREST	Compare string variables	3-7
EQUMB_	Compare multi-byte records for equality	3-8
EQUMW_	Compare multi-word records for equality	3-8
NEQMB_	Compare multi-byte records for inequality	3-8
NEQMW_	Compare multi-word records for inequality	3-8
MOVEB_	Move multi-byte record	3-9
MOVEW_	Move multi-word record	3-9
INITHEAP	Declares block of memory as memory pool	3-9
NEW	Dynamic memory allocation	3-9
DISPOSE	Dynamic memory deallocation	3-9
MARK	Save current status of dynamic memory heap	3-10
RELEASE	Restore prior status of dynamic memory heap	3-10
RENTRY_	Recursive procedure entry	3-10
REXIT_	Recursive procedure exit	3-10
PARAM_	Pass parameters to procedures	3-13
RPARAM_	Pass parameters to recursive routines	3-13

Table 3-2. Pascal Library Routines (for 8085)

Name	Purpose	Ref Page
Zbyteabs	Byte absolute value	3-18
Zbyteneg	Byte negation	3-18
Zbytesqr	Byte square	3-18
Zbyteadd	Byte addition	3-18
Zbytesub	Byte subtraction	3-18
Zbytemul	Byte multiplication	3-18
Zbytediv	Byte division	3-18
Zbytemod	Byte modulus (remainder of byte division)	3-18
Zbyteleq	Byte less than or equal	3-19
Zbyteles	Byte less than	3-19
Zbytegeq	Byte greater than or equal	3-19
Zbytegtr	Byte greater than	3-19
Zbshift	Byte shift logical with zero fill	3-21
Zbshiftc	Byte shift circular	3-21
Zintabs	Integer absolute value	3-15
Zintneg	Integer negation	3-15
Zintsqr	Integer square	3-15
Zintadd	Integer addition	3-15
Zintsub	Integer subtraction	3-15
Zintmul	Integer multiplication	3-15
Zintdiv	Integer division	3-15
Zintmod	Integer modulus (remainder of integer division)	3-15
Zintequ	Integer equal	3-16
Zintneq	Integer not equal	3-16
Zintleq	Integer less than or equal	3-16
Zintles	Integer less than	3-16
Zintgeq	Integer greater than or equal	3-16
Zintgtr	Integer greater than	3-16
Zwshift	Logical word shift with zero fill	3-21
Zwshiftc	Circular word shift	3-21
Zbtoint	Byte to integer	3-20
Zbinset8	Byte in 8-bit set	3-23
Zbinset16	Byte in 16-bit set	3-24
Zbtoset8	Byte to 8-bit set	3-22
Zbtoset16	Byte to 16-bit set	3-23
Zwinset8	Word in 8-bit set	3-25
Zwinset16	Word in 16-bit set	3-25
Zwtoset8	Word to 8-bit set	3-24
Zwtoset16	Word to 16-bit set	3-25
Zset16int	Intersection of 16-bit set	3-28
Zset16uni	Union of 16-bit set	3-27
Zset16geq	16-bit set greater than or equal	3-26
Zset16leq	16-bit set less than or equal	3-26
Zset16dif	Set difference of 16-bit set	3-28

Array Reference Routines

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

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

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

ARRAY_

The ARRAY_ routine returns the memory address of an n-dimensional array reference expression. An alternate form of array reference, ARRAYN_, is used if the array reference variable expression specifies less than the defined number of indices. In this case, the call is similar, but the number of actual index parameters is passed in register A of the 8085.

The array reference call for the 3-index array variable expression:

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

would be:

```
CALL    ARRAY_
DW      DOPE_VECTOR     ;for array A
DW      BASE_ADDRESS    ;of array A
DW      I               ;address of first index expression
DW      J               ;address of second index expression
DW      ADDR_CONST_7    ;address of the third index expression
```

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

```
B^(6+Y,J,7)
```

would generate a call to ARRAY_ in the form:

```
CALL    ARRAY_
DW      DOPE_VECTOR     ;for array of same type as A
DW      0               ;Indirect address indicator
DW      B               ;address of B which points to array
DW      D_TEMP          ;address where temp value (6+Y) stored
DW      J               ;address of second index expression
DW      ADDR_CONST_7    ;address of the third index expression
```

NOTE

The first parameter is the address of the array DOPE_VECTOR and will always be interpreted as the dope_vector address even if it is equal to zero. The BASE_ADDRESS parameter may be the actual base address, or if it is zero, it implies that the next word is a pointer to the actual base address. All of the index parameters will be the address of a 16-bit word containing a signed integer VALUE of the index expression.

ARRAYN_

The routine ARRAYN_ is used to compute the address of an array "row" which has been referenced by an array variable expression with less than the defined number of formal indices. The actual number of indices is passed in register A of the 8085.

An ARRAYN_ call for a two-index array variable expression for the array A (used in the previous example), but with the expression:

A(I,J)

would be:

```
MVI   A,2
CALL   ARRAYN_
DW      DOPE_VECTOR       ;for array A
DW      BASE_ADDRESS      ;of array A
DW      I                 ;address of first index expression
DW      J                 ;address of second index expression
```

Generalized Array Dope Vectors

For the general Pascal array defined by the declaration:

[1]

A : ARRAY [I1L..I1H,I2L..I2H,..., InMINUS1L..InMINUS1H,InL..InH] of any_type;

the address of the array element defined by the variable expression A[I1,I2,...,InMINUS1,In] is computed by the expression:

```
element_address := base_address                              [2]
                  +(I1-I1L)*(D2*D3 ... *Dn*BPE)
                  +(I2-I2L)*(D3* ... *Dn*BPE)
                   .
                   .
                   .
                  +(InMINUS1-InMINUS1L)*(Dn*BPE)
                  +(In-InL)*(BPE)
```

where:

```
In  -  Actual index expression for index n
InL -  Formal index lower bound for index n
InH -  Formal index upper bound for index n
Dn  -  "Row" size := (InH-InL+1) for index n
BPE -  Bytes Per Element of array element
         (size of data type any_type)
```

The constant terms representing the product of the index lower bounds(InL) and the "row" widths of the form:

```
PRODi := (DiPLUS1* ... *DnMINUS1*Dn*BPE)
```

may be combined into one constant called the OFFSET_CONSTANT. This constant is defined as:

```
OFFSET_CONSTANT := I1L*PROD1                                 [3]
                 + I2L*PROD2
                    .
                    .
                    .
                 + InMINUS1L*PRODnMINUS1
                 + InL*PRODn
```

The resulting combined formula can now be written as:

```
ADDRESS := BASE_ADDRESS + (OFFSET_CONSTANT)                  [4]
         + I1*PROD1+I2*PROD2 + ... + In*PRODn
```

Pascal defines the ARRAY type recursively as a single-dimensioned array of any declarable Pascal type. Thus, multi-dimensioned arrays are simply defined as ARRAYS of arrays. An array may be referred to in its entirety, a so called ENTIRE variable, by referring to the array by its name using no parameters. An array reference expression allows the user to refer to any ARRAY of N dimensions with from 1 to N index expressions.

For the array defined as in [1], an array variable expression with N-1 expressions will access one element of the type:

```
ARRAY[InL..InH] OF any_type.
```

An individual element of this ROW type may be accessed by the address expression:

[5]

```
row_address_nMINUS1 := BASE_ADDRESS
                       +(I1-I1L)*(D2*D3*...*Dn*BPE)
                       +(I2-I2L)*(D3*...*Dn*BPE)
                         .
                         .
                         .
                       +(InMINUS_1-InMINUS_1L)
                           *(DnMINUS_1*Dn*BPE)
```

When we compute a row address, we will compute the address as defined in [4], but omitting the unnecessary product terms. However, this expression has already incorporated the term:

```
ROWnMINUS_1_OFFSET := InL*PRODn
```

in the OFFSET_CONSTANT, so it must now be added back to the ADDRESS as defined in [5]. Thus, the computational expression used to compute the array row reference, using N-1 index expressions, is:

[6]

```
row_address:=BASE_ADDRESS+(-OFFSET_CONSTANT)+I1*PROD1+I2*PROD_2
                 +InMINUS_1*PRODnMINUS_1+ROWnMINUS_1_OFFSET
```

For each additional missing index expression there is one less multiplication but one more addition for the OFFSET_CONSTANT correction.

The form of the general array reference DOPE_VECTOR is equivalent to:

```
DOPE_VECTOR     DW    N                     ;number of dimensions
                DW    -(OFFSET_CONSTANT)    ;negative of constant
                DW    PROD_1
                DW    PROD_2
                 .
                 .
                 .
                DW    PROD_N
                DW    ROW1_OFFSET
                DW    ROW2_OFFSET
                 .
                 .
                 .
                DW    ROWnMINUS_1_OFFSET
```

NOTE

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

COMPAREST

The routine COMPAREST is used to compare two-byte strings. The byte strings are defined by the Pascal run time system to be of the type:

```
BYTESTRING = RECORD
               LENGTH : byte;
               STRING : ARRAY[1..LENGTH] OF BYTE;
             END;
```

The LENGTH is of the special type:

```
byte = 0..255
```

rather than the standard type:

```
BYTE = -128..127.
```

A variable of the type BYTESTRING can have a length of zero. In this case, no elements of the record subfield, STRING, will be accessed by any standard subroutine.

This routine will evaluate the dictionary style collating sequence based on the ASCII character codes.

The COMPAREST routine computes the boolean result of the integer comparison:

X op Y

where:

the address of string X is loaded in the DE register pair of the 8085, and the address of string Y is loaded in the HL register pair.

Registers B and C are not destroyed by this routine. In addition, the status bits reflect the result of the string comparison according to the following table:

Result	Z Flag	CY Flag
X = Y	SET	RESET
X < Y	RESET	SET
X > Y	RESET	RESET

Multi-byte Record Compares and Moves

EQUMB_ and EQUMW_

The routine EQUMB_ is used by the compiler to test multi-byte records of the same type for equality. The records might be defined in the source program by:

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

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

```
IF SALESPERSON = TOP_SALESPERSON THEN ....
```

The routine EQUMB_ or EQUMW_ would be given the address of one record in the HL register pair, the other record in the DE register pair, and the length of the record type in the BC register pair.

The result of the comparison will be available in register A as a boolean value and the Z flag will be set accordingly.

Results	Register A	Z Flag
Equality true	1	Reset
Equality false	0	Set

The B,C,D,E,H,L registers are restored to the input parameter values.

The routines NEQMB_ and NEQMW_ may be used to check for inequality. The result of the comparison will be available in register A as a boolean value and the Z flag will be set accordingly.

Results	Register A	Z Flag
Inequality true	1	Reset
Inequality false	0	Set

MOVEB_ and MOVEW_

The routines MOVEB_ and MOVEW_ are used for moving multi-byte records, such as in an assignment of a complete record type or array type to another of the same type.

The Pascal statements:

```
VAR X,Y : ARRAY[0..LENGTH] OF BYTE;
BEGIN
   .
   .
   X := Y;
```

would place the address of array X into register pair DE; the address of array Y into register pair HL; the length of the array (in bytes) into register pair BC, and subsequently call MOVEB_ for byte and char arrays or mixed type records or MOVEW_ for homogeneous integer records and integer arrays. All input registers will be altered.

Dynamic Memory Allocation

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

INITHEAP (Start_address, Length_in_bytes : INTEGER)

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

NEW (Pointer : Pointer_to_type)

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

DISPOSE (Pointer : Pointer_to_type)

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

MARK (Pointer : Pointer__to__type)

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

RELEASE (Pointer : Pointer__to__type)

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

Recursive Entry and Exit

The Pascal/64000 supports recursive and reentrant calling sequences by requiring a routine compiled in the $RECURSIVE$ mode to copy the current values of all of the local data area into a safe place (for the 8085 this is done using the 8085 stack itself) before executing any local code. In particular, the local storage must be saved before calling the RPARAM_ routine to copy any parameters.

RENTRY__

RENTRY_ is evoked if a procedure has been declared recursive ($RECURSIVE ON$). It must copy all of a procedure's local variables onto the stack and adjust the stack accordingly. Its calling sequence:

```
LXI B,DATA_LENGTH_ADDRESS  ;Address of data area size in bytes
LXI D,DATA_ADDRESS         ;Starting address of data
CALL RENTRY_
```

Note that the return address for the caller of X must be left on top of the stack after the call to RENTRY in order to permit RPARAM_ to work in the normal way.

This routine is called upon entry to a recursive Pascal procedure or function and RENTRY_ may destroy any or all registers.

REXIT__

After saving a procedure's data on the stack using RENTRY_, the procedure will need to restore its data prior to exit using the complementary routine REXIT_. Its calling sequence is:

```
LXI B,DATA_LENGTH_ADDRESS  ;Address of data area size in bytes
LXI D,DATA_ADDRESS         ;Starting address of data
CALL REXIT_
```

This routine is called prior to exit from a recursive Pascal procedure or function. Since the calling function has no way to preserve any local storage, REXIT_ preserves the values in the A and HL registers. All other registers and status bits may be modified as required.

Example

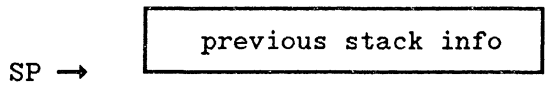
Consider the non-static procedure X, as defined in the following intermixed listing:

```
   PROCEDURE X (A : INTEGER);
    VAR B : BYTE;
    BEGIN

 X:                          ;X ENTRY POINT
  3:
      LXI     B,CX
      LXI     D,DX
      CALL    RENTRY_
  4:
      LXI     B,FX            ;DOPE_VECTOR
      LXI     D,DX
      LXI     H,2             ;TOTAL # OF BYTES FOR PARAMETERS
     CALL     RPARAM_         ;RECURSIVE PARAMETER PASSER
  5:
      ...                     ;BEGINNING OF LOCAL CODE FOR X
  6:
      LXI     B,CX
      LXI     D,DX
      CALL    REXIT_

       END;
  7:
      RET
```

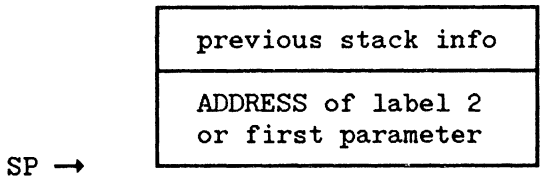
If the procedure were called at label 1, with the code:

```
1:  X(15)
2:
```

The stack would look like the following at label 1:

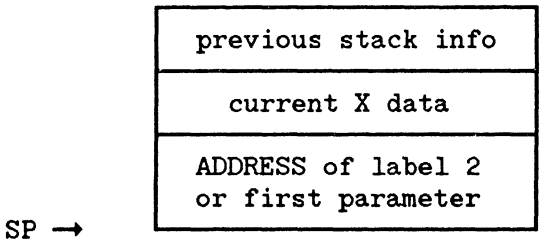


At label 3 the stack will have the address of the first parameter or the return address (the address of label 2) on top of the stack.



The routine, RENTRY_ will pop the stacked address, copy the designated number of bytes from the caller's data area onto the stack, and then push the previously popped address so that RPARAM_ will have the address it expects on the stack.

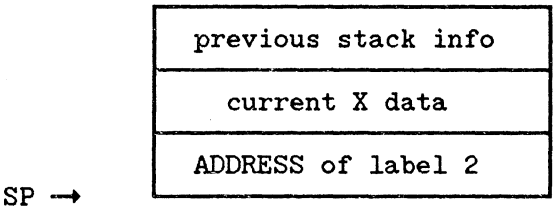
At label 4 the stack will have the copy of the local data as well as another copy of the address of label 2 or the first parameter. This is so that RPARAM_ can work properly.



When RPARAM_ is called, the stack is modified as the parameters are copied. This will continue until the actual return address (label 2) is left on top of the stack, after return from PARAMat label 5.

If there are no parameters, then RPARAM_ is not called and the stack will already have label 2 on it.

Thus the stack at label 5 looks like:



Since the local code of X must leave the stack unchanged upon its exit, the stack is again the same at label 6. A call must be made to REXIT_ to restore the previous set of current X data. After returning from REXIT- at label 7, the stack has been restored to its initial state (as upon entry at label 3). The return address has been updated as necessary to pass parameters and is now the address of label 2.

Parameter Passing

PARAM__ and RPARAM__

PARAM_ is the routine evoked to transfer parameters from the calling routine to the called routine.

```
LXI B,DOPE_VECTOR     ;Description of parameter formats
LXI D,DATA_ADDRESS    ;Address of DATA storage area
CALL PARAM_
```

The parameter DOPE_VECTOR is an integer array describing the parameters. It has the form:

```
                Number of parameters
          Description of first parameter
                        .
                        .
                        .
                        .
           Description of last parameter
```

The parameter description is the number of bytes for a value parameter. If the descriptor has the value -2, it is a parameter passed by an address (a Pascal VAR parameter).

The calling routine lists the parameters in order with a "DW address" for each parameter immediately following the call. The called routine will return to the instruction immediately following the last parameter. The calling routine may indicate that one level of indirect address is required by inserting a zero word before the actual parameter. This tells PARAM_ that the indicated parameter is the address of a variable pointing to the actual parameter.

```
CALL PROCWITH2PARAMS
DW FIRST              ;Address of first parameter
DW 0                  ;Indirect parameter flag
DW SECOND             ;Contains a pointer to actual param
...                   ;Next instruction after call
```

RPARAM_ (a special version of PARAM_) is required if the procedure or function is compiled in the $RECURSIVE ON$ mode. The calling sequence is similar to that used for PARAM_, with an additional instruction to load the HL register with the total number of bytes for parameters.

Example:

```
LXI     B,DOPE_VECTOR
LXI     D,DATA_ADDRESS
LXI     H,# of Bytes for parameters
CALL    RPARAM_
```

The special parameter passing routine is needed to ensure proper execution of recursive calling sequences.

NOTE

> Users who write assembly language programs that define and use procedures and functions, particularly with parameters, need to ensure that their use is consistent with the Pascal compiler. In order to accomplish this, it is recommended that the user write a simple Pascal program defining the procedure or function with the desired parameter list and an empty BEGIN END block for code. The user can then use the expanded listing file or the ASM_FILE option to determine how the Pascal compiler enters and exits the equivalent do- nothing procedures and how the parameter dope vector is defined. It is important that the user's assembly language routines follow the same entry, parameter passing, and exit code produced by the compiler. In particular, it is important that the parameter dope vector be identical to that produced by the compiler, and that recursive or static mode declaration (and use) be consistent.

Standard Integer Routines

Each group of routines in the Run-time Library has a standard method of passing parameters. In the case of the integer operations, the parameter values are passed in the DE and HL register pairs. They are re-entrant, non-recursive and safely callable from interrupted and interruptable processes. A "safe" interrupt routine saves all of the 8085 registers and restores them before returning control to the interrupted program.

The operands are 16-bit signed integers. The typical binary operation is of the form:

X op Y

where:
X is loaded in register pair DE
Y is loaded in register pair HL

The Library routine is called after loading the operands, X and Y, into the proper registers. Integer results are returned to register pair HL. The A register and the Z flag are also used to return boolean results for integer comparison operations.

There are three groups of integer operations: the unary ops, the binary arithmetic ops, and the binary comparison ops. Each group will be described separately.

Unary Integer Ops

```
Zintabs : Integer Absolute Value
Zintneg : Integer Two's Complement
Zintsqr : Integer Square
```

The unary operation is of the form:

```
op Y
```

where:

Y is loaded in register pair HL

The library routine is called after loading Y into the HL register pair. The result is returned in register pair HL.

Register Allocation Summary :: UNARY INTOPS	
INPUT:	HL contains parameter
OUTPUT:	HL contains result
REGISTERS:	
Modified:	H,L
Unchanged:	A,B,C,D,E

Binary Integer Ops

```
Zintadd :  Integer Addition
Zintsub :  Integer Subtraction
Zintmul :  Integer Multiplication
Zintdiv :  Integer Divide
Zintmod :  Integer Modulus
```

The binary arithmetic routines compute the integer result of the operation in the form:

```
X op Y
```

where :

X is loaded in register pair DE
Y is loaded in register pair HL

The library routine is called after loading the operands, X and Y, into the DE and HL register pairs, respectively. The result is returned in the register pair HL.

Register Allocation Summary :: BINARY INTOPS
INPUT: DE contains X, HL contains Y OUTPUT: HL contains result REGISTERS: (for Zintmul, Zintdiv, and Zintmod) Modified: D,E,H,L Unchanged: A,B,C REGISTERS: (for Zintadd and Zintsub) Modified: H,L Unchanged: A,B,C,D,E

NOTE

In specific instances, register-pair DE will contain useful information, namely:

for Zintmul - DE contains the high-order 16 bits of the 32-bit result.
for Zintdiv - DE contains the remainder value.
for Zintmod - DE contains the quotient value.

Integer Comparisons

```
Zintequ  :  Integer equality
Zintneq  :  Integer inequality
Zintleq  :  Integer less than or equal
Zintles  :  Integer less than
Zintgtr  :  Integer greater than
Zintgeq  :  Integer greater than or equal
```

The integer comparison routines compute the boolean result of the comparison operation:

X op Y

where:
X is loaded in register pair DE
Y is loaded in register pair HL

The library routine is called after loading the operands, X and Y, into the DE and HL register pairs, respectively. The boolean result is returned in register A and the Z flag. The Z flag is set as if the operation ORA A has just been performed on the boolean result of the comparison. If the result is false, register A will be zero and the Z flag will be set.

These routines change only register A and the status flags. The contents of registers B and C, as well as the input parameters, are either unchanged or restored by the Run-time Library routines.

Register Allocation Summary :: COMPARISON INTOPS
INPUT: DE contains X, HL contains Y OUTPUT: A set to 0, and Z flag set if result false A set to 1, and Z flag reset if result true REGISTERS: Modified: A,PSW Unchanged: B,C,D,E,H,L

Standard Byte Routines

For standard byte routines, parameter values are passed using registers A and L. The operands are 8-bit signed bytes. A typical binary expression is of the form:

X op Y

where:

X is loaded in register A
Y is loaded in register L

The library routine is called after loading operands, X and Y, into the proper registers. Byte results are returned to register A. Register A and the Z flag are also used to return boolean results for byte comparison operations.

There are three groups of byte operations: the unary ops, the binary arithmetic ops, and the binary comparison ops. Each group will be described separately.

Unary Byte Ops

```
Zbyteabs    : Byte Absolute Value
Zbyteneg    : Byte Two's Complement
Zbytesqr    : Byte Square
```

The byte operation is of the form:

op Y

where:

Y is loaded in register A

The library routine is called after loading Y into register A. The result is returned to register A.

Register Allocation Summary :: UNARY BYTEOPS
INPUT: A register contains parameter OUTPUT: A register contains result REGISTERS: Modified: A Unchanged: B,C,D,E,H,L

Binary Byte Ops

```
Zbyteadd    : Byte Addition
Zbytesub    : Byte Subtraction
Zbytemul    : Byte Multiplication
Zbytediv    : Byte Divide
Zbytemod    : Byte Modulus
```

For the byte arithmetic routines, the parameter values are passed in registers A and L. The binary arithmetic operations compute the byte result in the form:

X op Y

where:

X is loaded in register A
Y is loaded in register L

The library routine is called after loading the operands, X and Y, into the A and L registers. The result is returned to register A.

Register Allocation Summary :: BINARY BYTEOPS
INPUT: register A contains X, register L contains Y OUTPUT: register A contains the result REGISTERS: (for Zbytemul, Zbytediv, and Zbytemod) Modified: A,L Unchanged: B,C,D,E,H REGISTERS: (for Zbyteadd and Zbytesub) Modified: A Unchanged: B,C,D,E,H,L

NOTE

In specific instances, register L will contain useful information, namely:

for Zbytemul - L contains the high-order byte of the 16-bit result.

for Zbytediv - L contains the remainder value.

for Zbytemod - L contains the quotient value.

Byte Comparison

```
Zbyteleq   : Byte less than or equal
Zbyteles   : Byte less than
Zbytegeq   : Byte greater than or equal
Zbytegtr   : Byte greater than
```

The byte comparison routines compute the boolean result of the operation in the form:

X op Y

where:

X is loaded in register A
Y is loaded in register L

The library routine is called after loading the operands, X and Y, into registers A and L. The boolean result is returned to register A and the Z flag. The Z flag is set as if the operation ORA A has just been performed on the boolean result of the comparison. If the result is false, register A would be zero and the Z flag would be set.

These routines change only register A and the status flags. The contents of all other registers are either unchanged or restored by the Run-time library routines.

```
Register Allocation Summary :: COMPARISON BYTEOPS

 INPUT:  A contains X, L contains Y
OUTPUT:  A set to 0, and Z flag set if result false
         A set to 1, and Z flag reset if result true


 REGISTERS:
   Modified:  A,PSW
   Unchanged: B,C,D,E,H,L
```

Byte to Integer

The routine Zbtoint converts a signed byte into a signed integer.

```
   Register Allocation Summary :: Zbtoint

    INPUT:  A contains byte value to be converted
   OUTPUT:  HL contains the converted integer result


   REGISTERS:
     Modified:  H,L
     Unchanged: A,B,C,D,E
```

Byte and Word Shifts

Pascal/64000 supports logical and circular shifts of both byte and integer quantities using the functions SHIFT and SHIFTC. The DIV operator is used to do arithmetic right shifting: X DIV 2 would shift right with sign extension. A description of the shift functions is given in the following paragraphs.

SHIFT

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

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

where, expression represents any constant or variable quantity, and n is the number of positions to to be shifted as follows:

n>0 results in a shift left

n<0 results in a shift right

SHIFTC

Circular shifting rotates the quantity and fills the vacated bit position with the bit shifted out. The function call is of the form:

```
Variable := SHIFTC(expression,n):
```

Pascal/64000 determines the size (1 or 2 bytes) of the data being shifted by the type of the first parameter (expression) given. The 8085 library implementation of byte or word shifts are performed by the following routines:

```
Zbshift  -  logical shift of byte
Zwshift  -  logical shift of word
Zbshiftc -  circular shift of byte
Zwshiftc -  circular shift of word
```

The registers involved in word shifts (Zwshift and Zwshiftc) are as follows:

```
  INPUT:  HL = word to be shifted
           A = number of positions to shift word
 OUTPUT:  HL = shifted word
           A modified
```

The registers involved in byte shifts (Zbshift and Zbshiftc) are as follows:

```
  INPUT:   L = byte to be shifted
           A = number of positions to shift byte
 OUTPUT:   A = shifted byte
           L = unshifted byte
           H unchanged
```

Set Operations

Pascal/64000 supports 8- and 16-bit sets and are called byteset and wordset, respectively. For these sets all standard set operations are available. Sets of bytes or integers are assumed to be SET OF 0..15 and are wordsets. In the following descriptions of the set routines assume that some scalar and set types and data variables have been defined as follows:

```
TYPE
 BIT_8 = (BIT_0,BIT_1,BIT_2,BIT_3,BIT_4,BIT_5,BIT_6,BIT_7);
 SET_8 = SET OF BIT_8;
 BIT_16 = (BIT0,BIT1,BIT2,BIT3,BIT4,BIT5,BIT6,BIT7,BIT8,
           BIT9,BIT10,BIT11,BIT12,BIT13,BIT14,BIT15);
 SET_16 = SET OF BIT_16;

VAR
 SET8 : SET_8;
 SET16 : SET_16;
 Q,R : BIT_8;
 V,W : BIT_16;
 X,Y : SET_16;
```

Zbtoset8 Routine

This routine converts a byte into an 8-bit set. The only valid input values are 0 through 7. Out of range values are detected in debug library DLIB8085:HP, but may produce invalid results when using library LIB8085:HP. The Pascal statements:

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

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

Register Allocation Summary :: Zbtoset8
INPUT: A contains byte value to be converted OUTPUT: A contains the byteset result REGISTERS: Modified: A Unchanged: B,C,D,E,H,L

Zbinset8 Routine

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

```
R in SET8
```

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

Register Allocation Summary :: Zbinset8
INPUT: A contains byte value to be tested L contains the byteset being compared OUTPUT: A set to 0, Z flag set if value not in set A set to 1, Z flag reset if value in set REGISTERS: Modified: A,PSW Unchanged: B,C,D,E,H,L

NOTE

The Z flag is set or reset according to the result stored in register A.

Zbtoset16 Routine

This routine converts a byte into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library DLIB8085:HP, but may produce invalid results when using library LIB8085:HP. The pascal statements:

```
    V := BIT15;
SET16 := [V]
```

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

```
Register Allocation Summary  ::  Zbtoset16

 INPUT:  A contains byte value to be converted
OUTPUT:  HL contains the wordset result

REGISTERS:
  Modified:  H,L
  Unchanged: A,B,C,D,E
```

Zbinset16 Routine

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

V in SET16

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

```
Register Allocation Summary  ::  Zbinset16

  INPUT:  A contains byte value to be tested
          HL contains the wordset being compared
 OUTPUT:  A set to 0, Z flag set if value not in set
          A set to 1, Z flag reset if value in set

 REGISTERS:
   Modified:  A,PSW
   Unchanged: B,C,D,E,H,L
```

Zwtoset8 Routine

This routine converts a word into an 8-bit set. The only valid input values are 0 through 7. Out of range values are detected in debug library DLIB8085:HP, but may produce invalid results when using library LIB8085:HP.

```
Register Allocation Summary  ::  Zwtoset8

 INPUT:  HL contains word value to be converted
OUTPUT:  A contains the byteset result

REGISTERS:
  Modified:  A
  Unchanged: B,C,D,E,H,L
```

Zwinset8 Routine

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

```
W in SET8
```

is a boolean expression whose value is TRUE if bit W of SET8 is set and FALSE if bit W of SET8 is reset.

```
Register Allocation Summary  :: Zwinset8

 INPUT:  DE contains word value to be tested
         L contains the byteset being compared
OUTPUT:  A set to 0, Z flag set if value not in set
         A set to 1, Z flag reset if value in set

REGISTERS:
  Modified:  A,PSW
  Unchanged: B,C,D,E,H,L
```

Zwtoset16 Routine

This routine converts a word into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library DLIB8085:HP, but may produce invalid results when using library LIB8085:HP.

```
Register Allocation Summary  ::  Zwtoset16

 INPUT:  HL contains word value to be converted
OUTPUT:  HL contains the wordset result

REGISTERS:
  Modified:  H,L
  Unchanged: A,B,C,D,E
```

Zwinset16 Routine

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

```
W in SET16
```

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

```
Register Allocation Summary  ::  Zwinset16

 INPUT:  DE contains word value to be tested
         HL contains the wordset being compared
OUTPUT:  A set to 0, Z flag set if value not in set
         A set to 1, Z flag reset if value in set

REGISTERS:
  Modified:  A,PSW
  Unchanged: B,C,D,E,H,L
```

Zset16geq Routine

This routine is used to test the set inclusion of word sets. For example, the Pascal/64000 expression

```
SET16 >= SET_16[BIT0,BIT1,BIT7]
```

is a boolean expression whose value is TRUE if bits 0, 1, and 7 of SET16 are all set; otherwise, the value is FALSE. This is equivalent to asking if the set with bits 0, 1, and 7 set is a subset of SET16.

For expressions in the form:

```
X >= Y
```

the boolean result indicates whether Y is a proper subset of X.

```
Register Allocation Summary  ::  Zset16geq

 INPUT:  DE contains the wordset of the superset X
         HL contains the wordset of the subset Y
OUTPUT:  A set to 1, Z flag reset if Y is subset of X
         A set to 0, Z flag set otherwise

REGISTERS:
  Modified:  A,PSW
  Unchanged: B,C,D,E,H,L
```

Zset16leq Routine

This routine is used to test the set inclusion of word sets. For example, the Pascal/64000 expression

```
SET_16[BIT0,BIT1,BIT7] <= SET16
```

is a boolean expression whose value is TRUE if bits 0, 1, and 7 of SET16 are all set; otherwise, the value is FALSE. This is equivalent to asking if the set with bits 0, 1, and 7 set is a subset of SET16.

For expressions in the form:

```
X <= Y
```

the boolean result indicates whether X is a proper subset of Y.

```
Register Allocation Summary  ::  Zset16leq

 INPUT:  DE contains the wordset of the subset X
         HL contains the wordset of the superset Y
OUTPUT:  A set to 1, Z flag reset if X is subset of Y
         A set to 0, Z flag set otherwise

REGISTERS:
  Modified:  A,PSW
  Unchanged: B,C,D,E,H,L
```

Zset16uni Routine

This routine is used to compute the set union of two word sets.

For expressions in the form:

```
X + Y
```

the set union is a wordset containing all the elements in both wordset X and wordset Y.

```
Register Allocation Summary  :: Zset16uni

 INPUT:  DE contains the wordset X
         HL contains the wordset Y
OUTPUT:  HL contains the wordset result

REGISTERS:
  Modified:  H,L
  Unchanged: A,B,C,D,E
```

Zset16int Routine

This routine is used to compute the set intersection of two word sets.

For expressions in the form:

```
X * Y
```

the set intersection is the wordset containing only the elements contained in both wordset X and wordset Y.

```
Register Allocation Summary  ::  Zset16int

 INPUT:  DE contains the wordset X
         HL contains the wordset Y
OUTPUT:  HL contains the wordset result

REGISTERS:
  Modified:  H,L
  Unchanged: A,B,C,D,E
```

Zset16dif Routine

This routine is used to compute the set difference of two word sets.

For expressions in the form:

```
X - Y
```

the set difference is a set containing all the elements of wordset X which are not contained in wordset Y.

```
Register Allocation Summary  ::  Zset16dif

 INPUT:  DE contains the wordset X
         HL contains the wordset Y
OUTPUT:  HL contains the wordset result

REGISTERS:
  Modified:  H,L
  Unchanged: A,B,C,D,E
```

Appendix A

Run-time Error Descriptions

This appendix contains a description of the run-time errors that may occur.

Error Message	Indication
Z_ERR_CASE:	a jump to this error will occur when the test variable of a CASE statement is out of range and no OTHERWISE label exists. The following locations have valid information in them:
	Z_ZCALLER_H - contains the high byte of the address of the case statement.
	Z_ZCALLER_L - contains the low byte of the address of the case statement.
	Z_REG_H - contains the high byte of an integer test variable.
	Z_REG_L - contains the low byte of an integer test variable.
	Z_REG_A - contains the value of a byte test variable.
Z_ERR_DIV_BY_0:	a jump to this error will occur when the user attempts to divide a number by zero. The following locations have valid information:
	Z_ZCALLER_H - contains the high byte of the address of the statement which called the division routine.
	Z_ZCALLER_L - contains the low byte of the address of the statement which called the division routine.
	Z_REG_D - contains the high byte of an integer dividend.
	Z_REG_E - contains the low byte of an integer dividend.
	Z_REG_A - contains a byte dividend.

Z_ERR_HEAP: a jump to this error will occur when some misuse of the dynamic allocation keywords NEW, DISPOSE, MARK, or RELEASE has occurred. The following locations may have valid information:

Z_REG_A - contains the error type, see below.

Z_ZCALLER_H - may contain the high byte of the statement which called one of the dynamic allocation keyword procedures.

Z_ZCALLER_L - may contain the low byte of the statement which called one of the dynamic allocation keyword procedures.

Since these routines sometimes call one another, if the caller's address is the file HEAPSA, or HEAPS2A displaying the location (STACK_-10) will give the last ten subroutine calls and may lead to a call from the users modules.

Heap error types:

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

1 - heap has not been initialized, call INITHEAP 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 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.

Z_ERR_OVERFLOW: overflow may occur by multiplying or adding two numbers which result in a product of greater than 32,767 for an integer variable or 127 for a byte variable; by dividing -32,768 by -1; by squaring an integer greater than 181; by squaring a byte greater than 11; or by taking the absolute value of the minimum byte or integer.

If the called routine was a byte unary operation, then:

Z_REG_A contains the operand.

If the called routine was a byte binary operation, then:

Z_REG_A contains the operand X in the relation X op Y.

Z_REG_B contains Y.

If the called routine was an integer unary operation, then:

Z_REG_H contains the high byte of the operand.

Z_REG_L contains the low byte of the operand.

If the called routine was an integer binary operation, then:

Z_REG_D contains the high byte of X in the relation X op Y.

Z_REG_E contains the low byte of X.

Z_REG_H contains the high byte of Y.

Z_REG_L contains the low byte of Y.

Z_ERR_SET: operand is not a legal ordinal value for a set of the base type and:

Z_REG_A contains the operand.

Z_END_PROGRAM: a jump to this address occurs when the program completes execution of the main body code.

64810-90905, OCTOBER 1983
Replaces: 64810-90902, December 1980

PRINTED IN U.S.A.