UCSD p-System*

- Assembler
- Linker

Part Two: UCSD p-System Linker

*UCSD p-System is a trademark of the Regents of the University of California.

1041569-1
(Part B)

UCSD p-System*

- Assembler
- Linker

Part Two: UCSD p-System Linker

*UCSD p-System is a trademark of the Regents of the University of California.

This manual was developed by staff members of the Texas Instruments Education and Communications Center.

 Item involved met its quality assurance standards applicable to Version IV.0.

TABLE OF CONTENTS

SECTION 1: GENERAL INFORMATION

The UCSD p-System* and the Linker diskette allow you to link programs written in TMS9900 assembly language to programs written in UCSD Pascal*, allowing those programs to function as a single unit. This manual provides details on linking assembly language programs on the TI Home Computer.

The simplest configuration for running the Linker requires the TI Home Computer, the TI Color Monitor (or a Video Modulator and a television set), the Memory Expansion unit, the p-Code peripheral, and a Disk Memory System with at least one Disk Memory Drive. With this equipment, plus the diskette containing the Editor and Filer, you can link Pascal and assembly language programs. To enhance your system, you can add Disk Memory Drives, the RS232 Interface, or other peripherals available from Texas Instruments.

The p-System Editor (described in the UCSD p-System Editor owner's manual) allows you to create, edit, print, and save files. After a program file has been created, you can assemble it with the Assembler. Then, if you wish, you can use the Linker to link several object files.

After a file has been assembled and linked, you can load and run it as described in the UCSD p-System p-Code owner's manual.

*trademark of the Regents of the University of California.

1.1 USING THIS MANUAL

This manual assumes that you already know a programming language, preferably an assembly language. If you do not, there are many fine books available which teach the basics of assembly language use. After you know these basics, this manual gives the details of linking Pascal and assembly language files.

When terms that may be new to you are first used, they are defined. Section 2 explains the basics of using the Linker. Section 3 describes the detailed use of the Linker. The last section provides service and warranty information.

1.2 SET-UP INSTRUCTIONS

The steps involved in creating and linking an assembly language file and Pascal file are included in this section. Please read this material completely before proceeding.

Use your Disk Manager or the Pascal Filer to make a backup copy of the diskette which contains the Linker. You may use this copy for your own use. The original should be kept in a safe place.

The linking process combines a host object file which has been compiled from a Pascal source file and one or more external subroutines which have been assembled from assembly language programs. The combination produced is a single executable object file. See the UCSD Pascal Compiler manual for instructions on producing object code from a Pascal file, and the UCSD Assembler manual for instructions on producing assembly language programs.

There are two ways which may be used to link and run files. The first is to create an assembly language program and assemble it as SYSTEM.WRK.CODE or in a library using the LIBRARY utility (see the UCSD p-System Utilities manual). Then create the Pascal program which calls the assembly language program. Then the R(un command from the System promptline compiles the Pascal program, links it to the assembly language program, and runs the resulting code.

The following are the steps after the Pascal program has been created and the assembly language programs have been created and assembled.

1. Press **R**, for R(un. The Pascal program is compiled. Then the following messages appear.

```
Linking...
Opening <your file>
Opening *SYSTEM.LIBRARY
Reading <your file>
Reading <codefile>          (This is repeated for each
                            assembly language file to be
                            linked.)
Linking <your file>
   Copying proc <codefile>  (This is repeated for each
                            assembly language file to be
                            linked.)
```

2. The linked program is then run.

In the second way to link and run files, create the assembly language programs and assemble them. They may be placed in a library using the LIBRARY utility (see the UCSD p-System Utilities manual). Then create and compile the Pascal program which calls the assembly language program(s). Link the programs with the Linker, and then run the resulting object code with the X(ecute command from the System promptline.

The following are the steps after the Pascal and assembly language programs have been created and compiled or assembled.

1. If you have one disk drive, transfer the code files for the Pascal program and assembly language program to the diskette containing the Assembler and Linker. Use the Filer to make this transfer, as described in the UCSD p-System Filer owner's manual. If the Pascal program plus the assembly language program are too long to fit on this diskette, then two disk drives are required. If you have two disk drives, the Pascal code and assembly language code should be saved on the same diskette. If you have three disk drives, the Pascal program may be saved on any diskette.

2. Place the diskette that contains the Linker, the Pascal code, and the code from the assembly language program in a disk drive. If you have two or three disk drives, place the diskette that contains the Pascal and assembly language code in one of the drives.

3. Press **L**, for L(ink, to load the Linker.

4. The screen displays the message

   ```
   Linking...
   ```

   while the Linker is loaded.

   The following prompt appears.

   ```
   Host file?
   ```

Enter the location and name of the Pascal code file. For example, to link the Pascal program TEST.CODE, which is contained on the diskette in the second disk drive (#5), enter

```
#5:TEST
```

5. The message

```
Opening <file>
```

appears, with <file> the name of the Pascal program. The prompt

```
Lib file?
```

appears. Enter the location and name of the assembly language code file that you wish to have linked.

6. The message

```
Opening <file>
```

appears, where <file> is the assembly language file to be linked. Then the prompt

```
Lib file?
```

reappears and you can enter the location and name of another assembly language code file that you wish to have linked. This process is repeated until you just press <return> in response to the prompt, indicating that you have entered all of the assembly language code files which you wish to have linked. Then the following prompt appears.

```
Map name?
```

Enter the name of the file to which you wish the link map to be written. The link map is described in Section 3.

7. The following messages appear.

```
Reading <hostfile>
Reading <codefile>          (This is repeated for each assembly
                             language file to be linked.)
```

   Then the following prompt appears.

```
Output file?
```

   Enter the name of the linked file under which you wish the file saved. **Note:** The name must be followed by .CODE.

8. The following messages appear.

```
Linking <hostfile>
   Copying proc <codefile>  (This is repeated for each
                             assembly language file to be
                             linked.)
```

9. Run the linked file with the X(ecute command, as described in the UCSD p-System p-Code owner's manual.

If you have only one disk drive, the size of the programs which you may assemble, compile, and link is limited to the memory available on the diskette which contains the Compiler and the diskette which contains the Assembler and Linker. If you have two disk drives, then the programs and their code may occupy that memory available plus the space on the diskette which contains the program. You can compile the largest programs if you have three disk drives.

1.3 SPECIAL KEYS

In this manual, the keys that you press are indicated by surrounding them with <angle brackets>. The name <return> is used when the Pascal prompts on the screen refer to <return> or <cr> (carriage return). You should press the <ENTER> key. Pressing any key for more than approximately half a second causes that key to be repeated.

To obtain lower-case letters, press the key with the letter on it. To obtain all upper-case letters on the TI-99/4, use the alpha lock toggle to change to upper-case. On the TI-99/4A you may use the alpha lock toggle or press the <ALPHA LOCK> key. To obtain a single upper-case letter on the TI-99/4 when the computer is in lower-case mode, simultaneously press the key and the small space key on the left side of the keyboard or the space bar. On the TI-99/4A, press the key and <SHIFT>.

Name	TI-99/4	TI-99/4A	Action
<del>	SHIFT F	FCTN 1	Deletes a character.
<ins>	SHIFT G	FCTN 2	Inserts a character.
<flush>	SPACE 3	FCTN 3	Stops writing output to the screen.
<break>	SPACE 4	FCTN 4	Stops the program and initializes the System.
<stop>	SPACE 5	FCTN 5	Suspends the program until this key is pressed again.
<alpha lock>	SPACE 6	FCTN 6 or ALPHA LOCK	Acts as a toggle to convert uppper-case letters to lower-case and back again.
<screen left>	SPACE 7	FCTN 7	Moves the text displayed on the screen to the left 20 columns at a time.
<screen right>	SPACE 8	FCTN 8	Moves the text displayed on the screen to the right 20 columns at a time.
<line del>	SHIFT Z	FCTN 9	Deletes the current line of information.
{	SPACE 1	FCTN F	Types the left brace.
}	SPACE 2	FCTN G	Types the right brace.
[	SPACE 9	FCTN R	Types the left bracket.
]	SPACE 0	FCTN T	Types the right bracket.
<etx/eof>	SHIFT C	CTRL C	Indicates the end of a file.
<esc>	SPACE .	CTRL .	Tells the program to ignore previous text.
<tab>	SHIFT A	CTRL I	Moves the cursor to the next tab.
<up-arrow>	SHIFT E	FCTN E	Moves the cursor up one line.
<left arrow> or <backspace>	SHIFT S	FCTN S	Moves the cursor to the left one character.

Name	TI-99/4	TI-99/4A	Action
<right-arrow>	SHIFT D	FCTN D	Moves the cursor to the right one character.
<down-arrow>	SHIFT X	FCTN X	Moves the cursor down one line.
<return>	ENTER	ENTER	Tells the computer to accept the information you type.

SECTION 2: THE LINKER

The UCSD p-System Linker allows EXTERNAL code to be linked to Pascal and other p-code based languages. EXTERNAL routines are procedures, functions, or processes that are written in TMS9900 assembly language and follow the System's calling and parameter-passing rules. These routines are declared EXTERNAL in the host program and must be linked before the program is run. The Linker can also link separately assembled pieces of a single assembly language program. The Linker links code by installing the internal linkages that allow the pieces to function as a unified whole.

When a program which must be linked is R(un, the Linker automatically searches *SYSTEM.LIBRARY for the necessary external routines (See the p-System Utilities manual). In all other cases (for example, if you use X(ecute instead of R(un or if the library is not SYSTEM.LIBRARY), you must link the code before executing it. To link code without R(un, access the Linker by pressing **L** when the System promptline is displayed.

When the Linker is called automatically and cannot find the needed code in *SYSTEM.LIBRARY, it responds with an error message as shown below.

```
        Proc,
        Func,
        Global,
  or    Public <identifier> undefined
```

Then the System promptline is redisplayed.

2.1 USING THE LINKER

When you access the Linker, it asks for several file names and displays the names of what it is linking as it reads and links code together. The first prompt asks for the host file.

```
Host file?
```

The host file is the file into which the external routines are to be linked. File name conventions apply here, so .CODE is automatically appended to all file names unless you except type * and press <return> or a file name that ends in a period (.). Typing * and pressing <return> or simply pressing <return> causes the Linker to use *SYSTEM.WRK.TEXT.

The Linker then asks for the names of library files in which external routines are to be found.

```
Lib file?
```

Any number of library files can be specified. The prompt reappears until you press <return> without typing a file name. Typing * and pressing <return> opens *SYSTEM.LIBRARY. The success of opening each library file is reported. If you enter the name of a file that is not on line, the message

```
Type <sp>(continue), <esc>(terminate)
```

appears. Press the space bar to enter another file name, or <esc> to terminate the linking process.

The code file produced by the Linker contains routines in the order in which they were given as contained in the library files. The code file first contains routines from the host file, followed by library file routines, all in their original order.

The following is a sample portion of a run of the Linker.

Prompt	Your Input
Lib file?	*<return>
Opening *SYSTEM.LIBRARY	
Lib file?	FIX.8<return>
No file FIX.8.CODE	
Type <sp>(continue), <esc>(terminate)	<space>
Lib file?	FIX.9<return>
Opening FIX.9.CODE	
bad seg name	
Type <sp>(continue), <esc>(terminate)	<space>
Lib file?	

When the names of all library files have been entered, the Linker reads all the necessary routines from the designated code files. It then asks for a destination for the linked code output.

```
Output file?
```

This is a code file name and is often the same as the host file. The .CODE suffix must be included. If you just press <return>, output is to the work file, *SYSTEM.WRK.CODE.

After the last prompt the Linker starts linking. During linking, the names of all of the routines being linked are displayed. A missing or undefined routine causes the Linker to stop with the <identifier> undefined message described above. If linking is successful, you have a unified code file that can be X(ecuted if it contains p-code. See the UCSD p-Code manual for detailed instructions.

SECTION 3: PROGRAM LINKING AND RELOCATION

The Assembler produces either absolute or relocatable object code that can be linked to create executable programs from separately assembled or compiled modules.

Program linking directives generate information required by the Linker to link modules. Some of the advantages of linking are as listed below.

- Long programs can be divided into separately assembled modules to avoid a long assembly, reduce the symbol table size, and encourage modular programming techniques.
- Modules can be shared by other linked modules.
- Utility modules can be added to the System Library for use as external procedures by a large number of programs.
- Pascal programs can directly call assembly language procedures.

The Assembler generates linker information in both relocatable and absolute code files. The Linker accesses this information during the linking process and removes it from the linked code file.

Relocatable code includes information that allows a loader program to place that code anywhere in memory, while absolute code files, also called core image files, must be loaded into a specific area of memory to execute properly. Assembly procedures running in the UCSD p-System environment must always be relocatable, since the loading and relocation process is performed by the interpreter at a load address determined by the state of the System.

Code segments which contain statically relocatable code remain in main memory throughout the existence of their host program (or unit) and are position-locked for that duration. Thus, relocatable code can maintain and refer to its own internal data space (or spaces). In addition, statically relocatable code saves some space because its relocation information does not have to remain present throughout the existence of the program.

The directives .PROC and .FUNC designate statically relocatable routines, while .RELPROC and .RELFUNC designate dynamically relocatable routines. See The UCSD p-System Assembler manual for more information on these directives. Code segments which contain dynamically relocatable code do not necessarily occupy the same location in memory throughout their host's existence, but are maintained in the code pool along with other dynamic segments, and can be swapped in and out of main memory while the host program (or unit) is running. Thus, dynamically relocatable code cannot maintain internal data spaces. This means that data which is meant to last across different calls of the assembly routine must be kept in host data segments using the directives .PRIVATE and .PUBLIC.

In the following example, data space is embedded in the code, but the code does not move.

```
.PROC     FOON
.WORD     SPACE
...
.END
```

In this example, the code moves, but data space is allocated in the host compilation unit's global data segment.

```
.RELPROC  FOON
.PRIVATE  SPACE
...
.END
```

The following is an example of something that does not work. In it, the code moves and the data is embedded in the code, thus destroying the data.

```
.RELPROC  FOON
.WORD     SPACE
...
.END
```

Code pool management is described in the Internal Architecture Guide.

3.1 PROGRAM LINKING DIRECTIVES

This section describes overall usage of linking directives. Because all linking of assembly procedures involves word quantities, it is not possible to define and refer to data bytes or assembly-time constants externally. Arguments of these directives must match the corresponding name in the target module (a lower-case Pascal identifier matches an upper-case assembly name and vice versa) and must not have been used before their appearance in the directive. All following references to the arguments are treated by the Assembler as special cases of labels. These external references are resolved by the Linker and/or interpreter by adding the link-time and run-time offsets to the existing value of the word quantity in question. Thus, any initial offsets generated by the inclusion of external references and constants in expressions are preserved.

3.1.1 Pascal Host Communication Directives

The directives .CONST, .PUBLIC, and .PRIVATE allow the sharing of constants and data between an assembly procedure and its host compilation unit. See Section 3.2.2 for an example.

.CONST Accesses globally declared constants in the host compilation unit. All references to arguments of .CONST are replaced by the Linker with a word containing the value of the host's compile time constant.

.PUBLIC Accesses globally declared variables in the host compilation unit. This directive can be used to set up pointers to the start of multi-word variables in host programs because it is not limited to single-word variables.

.PRIVATE Declares variables in the global data segment of the host compilation unit that are inaccessible to the host. The optional length attribute of the arguments allows multi-word data spaces to be allocated. The default data space is one word.

3.1.2 External Reference Directives

The directives .REF and .DEF allow separately assembled modules to share data space and subroutines. See Section 3.2.2 for examples.

.DEF Declares a label defined in the current program as accessible to other modules. Note, however, that it is invalid to .DEF a label that has been equated to a constant expression or an expression containing an external reference.

.REF Declares a label that exists and is defined with .DEF in another module to be accessible to the current program.

3.1.3 Program Identifier Directives

The directives .PROC, .FUNC, .RELPROC, .RELFUNC, and .END delimit source programs. Every source program (relocatable or absolute) must contain at least one pair of delimiting directives (see the UCSD p-System Assembler manual).

The identifier argument of the .PROC or .RELPROC directive serves two functions. First, it is referred to by the Linker when linking an assembly procedure to its corresponding host. Second, it can be referred to as an externally declared label by other modules. Specifically, in a source program the declaration

```
.PROC   FOON                    ; Procedure heading.
```

is functionally equivalent to the following statements in the assembly environment.

```
        .DEF    FOON            ; FOON can be externally
                                  referenced.
FOON                            ; Declare FOON as a label.
```

Thus, one assembly module can call other external and eventually linked assembly modules by name. The .FUNC and .RELFUNC directives link an assembly function directly to a System host program. They are not intended for linking with other assembly modules.

The optional integer argument after the procedure identifier is referred to by the Linker to determine if the number of words of parameters passed by the System host's external procedure declaration matches the number specified by the assembly procedure declaration. The integer argument is not relevant when linking with other assembly modules.

3.2 LINKING PROGRAM MODULES

For information on linking with the System's high-level languages other than Pascal, refer to the documentation for that particular language.

3.2.1 Linking with a Pascal Host Program

External procedures and functions are assembly language routines declared in Pascal programs. To run Pascal programs with external declarations, you must compile the Pascal program, assemble the external procedure or function, and link the two code files. The linking process can be simplified by adding the assembled routine to the System Library using the LIBRARY program. (See the UCSD p-System Utilities manual.)

A Pascal host program declares a procedure to be external with a syntax similar to that of a forward declaration. The procedure heading is given (possibly with a parameter list), followed by the keyword "EXTERNAL". Calls to the external procedure use Pascal syntax, and the Compiler checks to be sure that the calls agree in type and number of parameters with the external declaration. All parameters are pushed on the stack in their order of appearance in the parameter list of the declaration. Thus, the rightmost parameter in the declaration is on the top of the stack.

Assembly language programs can use registers R0 through R7. The stack pointer is register R10. Return is through register R11 with

```
B       *R11
```

To pop a value off the stack, use an instruction of the form

```
MOV     *R10+,R1
```

To push a value on the stack, use instructions of the form

```
DECT    R10
MOV     R1,*R10
```

In both of these examples, R1 can be replaced by any address.

It is your responsibility to ensure that the assembly language routine maintains the integrity of the stack. This includes removing all parameters passed from the host, preserving any machine resources in use by the interpreter, and making a clean return to the Pascal environment using the return address originally passed to it. If something goes wrong, the System may cease to function, requiring rebooting, because assembly routines are outside the scope of the Pascal environment's run-time error facilities.

An external function is similar to a procedure but with some differences that affect the way in which parameters are passed to and from the Pascal environment. The external function call pushes one or four words on the stack (four for a function of type real and one for all other types) before any parameters have been pushed. The words are part of the System's function calling mechanism and are irrelevant to assembly language functions. The assembly routine must remove these one or four words before returning the function's result. The assembly routine must push the proper number of words (four for type real, one otherwise) containing the function result onto the stack before passing control back to the host.

The ability of external procedures to pass any variables as parameters gives you complete freedom to access the machine-dependent representations of machine-independent Pascal data structures. However, with this freedom comes the responsibility of respecting the integrity of the Pascal run-time environment. This section lists the System's parameter passing conventions for all data types. However, it does not describe the data representations because they are complex and are best found by examination.

Parameters can be passed either by value or by name (also known as variable parameters). For purposes of assembly language manipulation, variable parameters are handled in a more straightforward fashion than value parameters.

Variable parameters are referred to through a one-word pointer passed to the procedure. Thus, the procedure declaration

```
procedure pass_by_name(var i,j: integer; var q: some_type);
external;
```

passes three one-word pointers on the stack. The top of stack is a pointer to q, followed by pointers to j and i.

PROGRAM LINKING AND RELOCATION

A Pascal external procedure declaration can contain variable parameters lacking the usual type declaration, thus enabling variables of different Pascal types to be passed to an assembly routine through a single parameter. Untyped parameters are not allowed in normal Pascal procedure declarations.

For example, the procedure declaration

```
procedure untyped_var(var i; var q: some_type); external;
```

contains the untyped parameter i.

The way value parameters are handled depends on their data type. The following types are passed by pushing copies of their current values directly on the stack: Boolean, character, integer, real, subrange, scalar, pointer, set, and long integer. For instance, the declaration

```
procedure pass_by_value(i: integer; r: real); external;
```

passes an eight byte real number which contains the value of the real variable r followed by one word which contains the value of the integer variable i.

Variables of type RECORD and ARRAY are passed by value in the same manner as variable parameters; pointers to the actual variable are pushed onto the stack. Variables of type PACKED ARRAY OF CHAR and STRING are passed by value with a segment pointer.

Pascal procedures protect the original variables by using the passed pointer to copy their values into a local data space for processing. Assembly procedures should respect this convention and not alter the contents of the original variables.

A segment pointer consists of two words on the stack. The first word (the top of the stack) contains either the value 0 or another pointer. If the value passed is a variable, the second word (the top of the stack minus 2 bytes) points to the parameter and the first word is 0.

If the first word is not 0, the value passed is a string constant. The best way to get the value is to use a dummy Pascal procedure to pass the pointer to the string, as shown in the following example.

```
Program constant_string_example;
const   a_string = 'This is a constant string';
Procedure assembly_call(var fake_string:string); external;
Procedure pass_constant(a_parm:string);

{ This procedure makes the System copy the constant string to
a variable.  The variable is then passed to the assembly
language program, which can access the copy as discussed
above. }

begin { pass_constant }
        assembly_call(a_parm);
end;

begin {main program }
        pass_constant(a_string);
end.
```

3.2.2 Example of Linking to Pascal Host

In the following example, the host program passes control to the beginning of an assembly procedure, regardless of whether machine instructions are present. Therefore, all data sections allocated in the procedure must either occur after the end of the machine instructions or have a jump instruction branch around them.

```
PROGRAM EXAMPLE;                    { Pascal host program }
const size = 80;
var i,j,k: integer;
    lst1: array [0..9] of char;
    { PRT and LST2 get allocated here }
procedure do_nothing; external;
function null_func(xxyxx,z: integer): integer; external;
begin
  do_nothing;
  j := null_func(k,size);
end.
```

```
        .PROC    DONOTHING    ; Underscores are not significant
                              in Pascal.
        .CONST   SIZE         ; Can get size constant in host
        .PUBLIC  I,LST1       ; ... and also these two global
                              variables.
        .DEF     TEMP1        ; This allows NULLFUNC to get
                              temp1.
                              ; Code starts here.
        B        *R11         ; Does nothing.
TEMP1   .WORD                 ; End of procedure DONOTHING.
        .FUNC    NULLFUNC,2   ;Two words passed.
        .PRIVATE PRT,LST:9    ; 10 words of private data.
        .REF     TEMP1        ; Refers to data temp1 in
                              DONOTHING.
                              ; Code starts here.
        MOV      *R10+,TEMP1  ; Get value of Z.
        MOV      *R10+,PRT    ; Get value of XXYXX.
        MOV      *R10+,JUNK   ; Get one junk word since we
                              return a one word value.
        MOV      LST,LST      ; Perform null action.
        DECT     R10          ; Return result.
        MOV      LST+4,*R10
        B        *R11         ; Returns to calling program.
JUNK    .WORD                 ; Data starts here.
        .END
```

3.2.3 Stand-Alone Applications

The System does not include a linking loader or an assembly language debugger because the System architecture is not conducive to running programs (whether high- or low-level) that must reside in a dedicated area of memory. You are responsible for loading and executing the object code file. This can be done by the System with the understanding that the existing environment may be jeopardized in the process.

With the .ABSOLUTE and .ORG directives, you can create an object code file suitable for use as an absolute core image. .ABSOLUTE creates nonrelocatable object code and .ORG can be used to initialize the location counter to any starting value. A source file headed by .ABSOLUTE should not have more than one assembly routine because sequential absolute routines do not produce continuous object code and cannot be successfully linked with one another to produce a core image.

The code file format consists of a one-block code file header followed by the absolute code and terminated by one block of linker information. Thus, stripping off the first and last block of the code file leaves a core image file. The use of .ABSOLUTE should be limited to one routine because, although linker information is generated, it is difficult to link absolute code files to produce a correct core image file.

The following paragraphs describe one method of loading and executing absolute code files with the System. The program outlined is not the only solution. You could also use the System intrinsics to read and/or move the code file into the desired memory location, but this requires a knowledge of where the interpreter, Operating System, and user program reside so that you do not accidentally overwrite them and possibly cause the System to cease to function, requiring rebooting. The program outlined below allows the most freedom in loading core images. The only constraint is that the assembly code itself is not overwritten while being moved to its final location.

Note that in most cases loading object code into arbitrary memory locations while the System is resident adversely affects the System because the absolute assembly language program is then on its own, and rebooting may be necessary to revive the System.

PROGRAM LINKING AND RELOCATION

The loader program should consist of the following.

A Pascal host program that calls two external procedures.

One or more linkable absolute code files to be loaded. (.RELPROCs are not allowed.)

A small assembly procedure MOVE_AND_GO that moves the above object code files from their System load address to their proper locations and transfers control to them.

A small assembly language procedure LOAD_ADDRESS that returns the System load addresses of the previously mentioned assembly code to the host program.

The absolute code files are assembled to run at their desired locations, and MOVE_AND_GO contains the desired load addresses of each core image. Both LOAD_ADDRESS and MOVE_AND_GO have external references to the core images which are used to calculate the System load address and code size of each image file. The whole collection is linked and executed, with the Pascal host performing the following actions.

- Print the result of calling LOAD_ADDRESS to determine whether the area of memory in which the System loaded the assembly code overlays the known final load address of the core images.

- Issue a prompt to continue so that the program can be stopped if a conflict does arise.

- Call MOVE_AND_GO.

SECTION 4: IN CASE OF DIFFICULTY

1. Be sure that the diskette you are using is the correct one. Use the L(dir (list directory) command in the Filer to check for the correct diskette or program.

2. Ensure that your Memory Expansion unit, P-Code peripheral, and Disk System are properly connected and turned on. Be certain that you have turned on all peripheral devices and have inserted the appropriate diskette before you turn on the computer.

3. If your program does not appear to be working correctly, end the session and remove the diskette from the disk drive. Reinsert the diskette, and follow the "Set-Up Instructions" carefully. If the program still does not appear to be working properly, remove the diskette from the disk drive, turn the computer and all peripherals off, wait 10 seconds, and turn them on again in the order described above. Then load the program again.

4. If you are having difficulty in operating your computer or are receiving error messages, refer to the "Maintenance and Service Information" and "Error Messages" appendices in your User's Reference Guide or UCSD p-System P-Code manual for additional help.

5. If you continue to have difficulty with your Texas Instruments computer or the UCSD p-System Pascal Compiler package, please contact the dealer from whom you purchased the unit or program for service directions.

THREE-MONTH LIMITED WARRANTY
HOME COMPUTER SOFTWARE MEDIA

Texas Instruments Incorporated extends this consumer warranty only to the original consumer purchaser.

WARRANTY COVERAGE

This warranty covers the case components of the software package. The components include all cassette tapes, diskettes, plastics, containers, and all other hardware contained in this software package ("the Hardware"). This limited warranty does not extend to the programs contained in the software media and in the accompanying book materials ("the Programs").

The Hardware is warranted against malfunction due to defective materials or construction. **THIS WARRANTY IS VOID IF THE HARDWARE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIAL OR WORKMANSHIP.**

WARRANTY DURATION

The Hardware is warranted for a period of three months from the date of original purchase by the consumer.

WARRANTY DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE PRODUCT OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

PERFORMANCE BY TI UNDER WARRANTY

During the three-month warranty period, defective Hardware will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below. The replacement Hardware will be warranted for a period of three months from the date of replacement. TI strongly recommends that you insure the Hardware for value prior to mailing.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U. S. Residents:
Texas Instruments Service Facility
P. O. Box 2500
Lubbock, Texas 79408

Canadian Residents only:
Geophysical Services Incorporated
41 Shelley Road
Richmond Hill, Ontario, Canada L4C5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments Consumer Service
6700 Southwest 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

Texas Instruments Consumer Service
831 South Douglas Street
El Segundo, California 90245
(213) 973-1803

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAMS

The following should be read and understood before purchasing and/or using the software media.

TI does not warrant the Programs will be free from error or will meet the specific requirements of the consumer. The consumer assumes complete responsibility for any decisions made or actions taken based on information obtained using the Programs. Any statements made concerning the utility of the Programs are not to be construed as express or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAMS AND MAKES ALL PROGRAMS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAMS AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE SOFTWARE MEDIA. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE USER OF THE PROGRAMS.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.