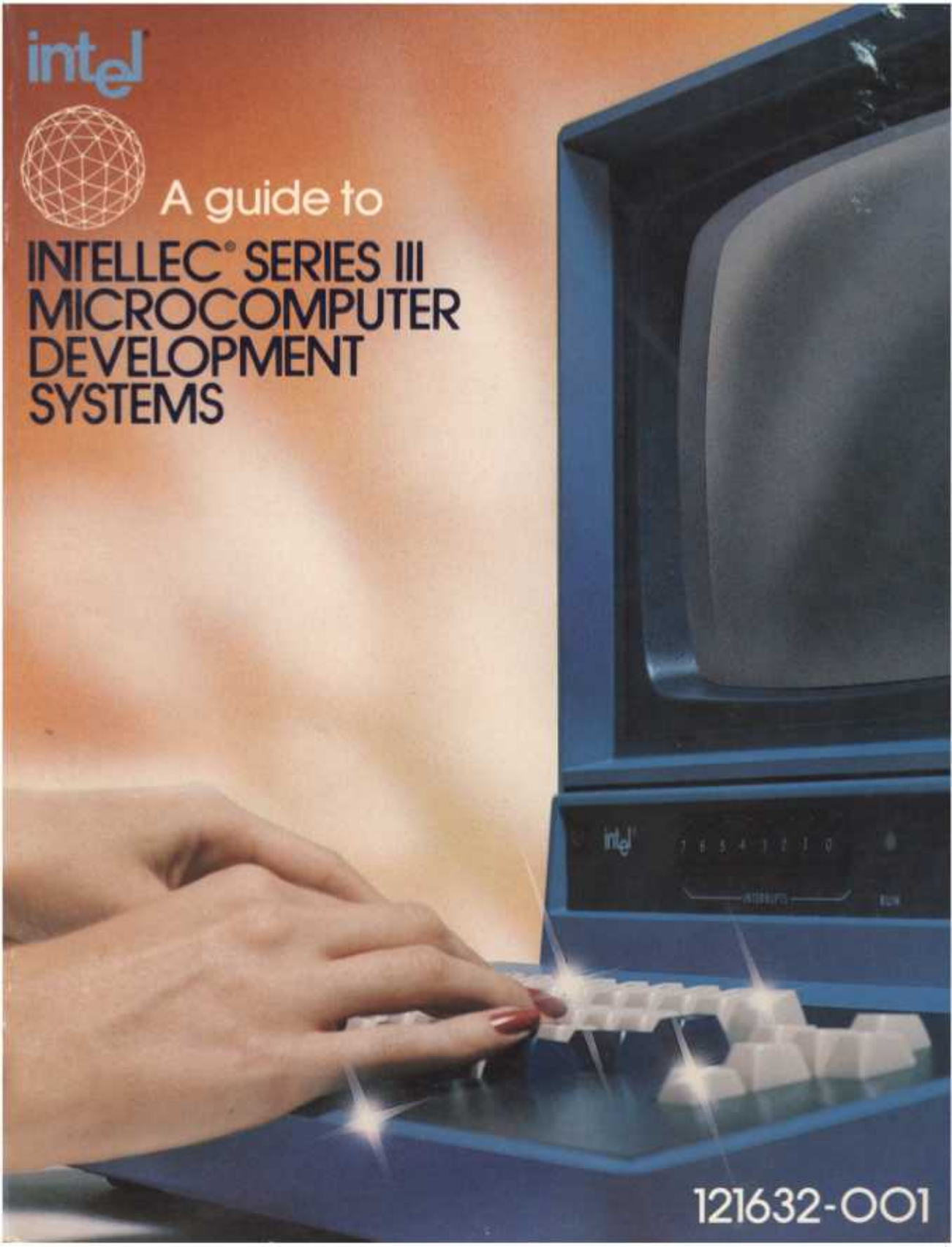
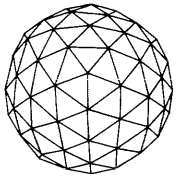
intel

A guide to INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEMS



121632-001

intel®



A guide to INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEMS

121632-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

PREFACE

This book is a welcome mat for the Intellec Series III Microcomputer Development System. It is also an introduction to the world of software development for micro-applications. We assume that you have some basic knowledge of microprocessors and their applications, but few demands are made on that knowledge. We do not assume that you have had exposure to any particular programming language. This book can be useful to beginners, and even sophisticated readers should find it rewarding to skim through.

This book is a tutorial for using the Series III system, especially the "8086 side" of it (the 8086 execution environment). The "8085 side" is similar to a Series II system, which is described in *A Guide to Intellec Microcomputer Development Systems* by Daniel McCracken.

We lead you through a typical software development process by providing an example of a micro-application: a climate control system for a building. To keep the example easy to understand, we only describe the software development effort, assuming that the hardware for the climate system is being developed simultaneously. In fact, we illustrate some typical problems in software development that occur as a result of changing hardware designs.

Chapter 1 gives an overall view of the Series III system and the application example. It also describes top-down design, stepwise refinement, modular programming, design considerations, and how to choose the proper software language for each module.

Chapter 2 is a step-by-step tutorial on the Series III operating system, showing typical operations.

Chapter 3 is a step-by-step tutorial on CREDIT, and it incidently shows the process of stepwise refinement of the application's main control algorithm.

Chapter 4 describes Pascal-86 programming, structured and modular design, parameter passing, data typing, and the Pascal-86 compiler.

Chapter 5 describes PL/M-86 programming, and it shows a sample PL/M-86 routine used in the application. It also briefly describes the PL/M-86 compiler and the 8086/8087/8088 Macro Assembler.

Chapter 7 describes program debugging with DEBUG-86 and hardware emulation with the ICE-88 emulator.

There is also a bibliography of related material, and a list of Intel manuals supplied with the Intellec Series III Microcomputer Development System.

CONTENTS

CONTENTS (Cont'd.)

ILLUSTRATIONS

CHAPTER 1
THE SOFTWARE DEVELOPMENT PROCESS

"Hardware is computing *potential*; it must be harnessed and driven by software to be useful."
--Andrew S. Grove, President of Intel Corp.

The Intellec Series III Microcomputer Development System is more than a keyboard, a video display, an integral disk drive, and a box with two microprocessors. It is a useful tool for designing microcomputer software for the iAPX 86,88 processor family or for the 8080/8085 processors. You can choose the appropriate language (PL/M, FORTRAN, Pascal, macro-assembly language) for each piece of software, debug these pieces separately, and link them in different ways for different applications. The applications can then be run on this system or any other system that is based on the iAPX 86,88 or 8080/8085 families of processors.

Intel's iAPX microprocessor family provides an architecture best suited for modular software development using high-level languages. The Intellec Series III Microcomputer Development System takes full advantage of this architecture to provide a more cost effective programming environment that guarantees a shorter development cycle.

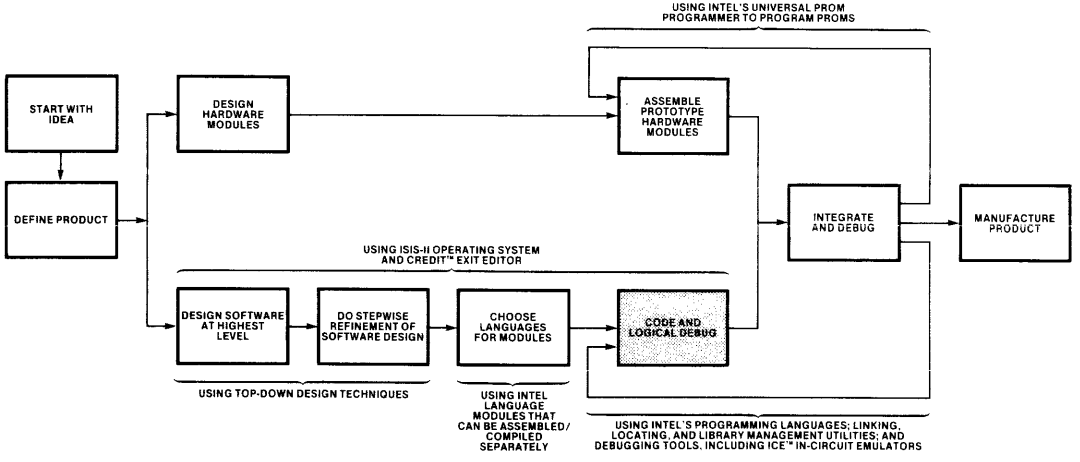
To design a product that will contain a microprocessor, you must coordinate two efforts: the design of the hardware that surrounds the microprocessor, and the design of the software that controls the microprocessor. Hardware development involves planning the interaction of the microprocessor, the associated memory and peripheral circuits, and the specialized input/output circuits and processors. Software development involves programming the microprocessor with instructions that will eventually be stored in the product's memory. These instructions must be designed to correctly perform the required tasks.

It is possible to carry out these development efforts independently--the hardware development separate from the software development. In practice, however, it takes a long time to develop error-free software on prototype hardware. To achieve good system integration and to save time, software debugging must usually begin long before prototype hardware is available to test the software.

The Intellec Series III with in-circuit emulation (ICE) is a development solution because it provides support for parallel hardware and software development efforts. Using the ICE-86 or ICE-88 emulator, you can emulate parts of your prototype hardware in order to test your software in a stable environment that resembles your final product. The ICE-86 or ICE-88 emulator also allows you to substitute memory and other resources from the Series III system for the memory and resources missing in your prototype hardware. With the ICE-86 or ICE-88 emulator, prototype hardware can be added to your product as you are designing it, and software and hardware testing can occur simultaneously (thereby speeding up the entire development process).

The chart in figure 1-1 summarizes a software development process, starting with an idea for a final product. Such a process always starts with an idea, which you refine step by step until you can define the actual product's environment, hardware, and logic.

To provide a tutorial on using the Intellec Series III system, and to show how Intel's software development tools are used in a development situation, we provide a simple software application for the iAPX 88 microsystem, at the heart of which is an 8088 microprocessor. The application is a climate control system for a building that uses a solar collector for heating and cooling, with backup methods of heating and cooling when the solar collector is not adequate. All methods use water, storage tanks, and a water-to-air exchanger or heat pump.



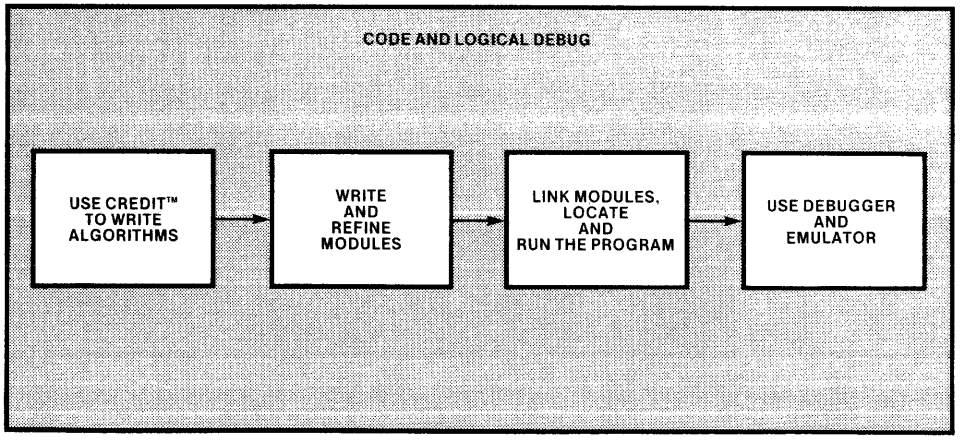


Figure 1-1. Developing Software on the Series III System 121632-1

Several decisions about this climate control system can be deferred to a later date. For example, by designing the system's software in a modular fashion, we can add more methods of heating or cooling as necessary, and we can decide how to handle water pumps and valves at a later date. We know now that the software's primary purpose is to choose a method of heating or cooling based on temperature data, and to operate the climate system's pumps and valves. Figure 1-2 is a simplified block diagram of the application.

DEFINING THE PRODUCT'S SOFTWARE

As you define your product's hardware, you must also define the *purpose* of its software. For example, the purpose of the software for our application is to gather and store the appropriate temperature data, decide on a heating or cooling method based on that data, implement the method in the climate system, and maintain the operation of the climate system. Each task can be designed as a piece of software called a *module*. By keeping tasks modular, you can change the details of any task without affecting the details of the other tasks.

Keep in mind that software provides the capability to change or add to the product. The entire product could consist of hardware and logic circuits, but then you might have to rebuild each unit to add more capabilities or change the flow of the logic. Software that is not modular and easy to maintain does not solve this problem; therefore, you must define the entire purpose of the software, with an eye to the future of your product. Keep it modular so that you can replace modules easily without rewriting modules that already work.

For our climate control system, we defined the software to be a set of modules that receive and store data, decide heating or cooling methods based on that data, and decide how to operate the hardware associated with the climate system.

At this time, we do not need a more detailed definition; in fact, more detail would hinder our process of step by step refinement. It is important to realize the order for these actions: first, the software has to start up the climate system. Once started, the software has to do several things over and over (unless the system shuts down): (1) read the various temperatures, (2) store the data for future reference, (3) decide on a heating or cooling method to use, and (4) operate the climate system to provide heating or cooling and to maintain the system (e.g., maintain heat gain).

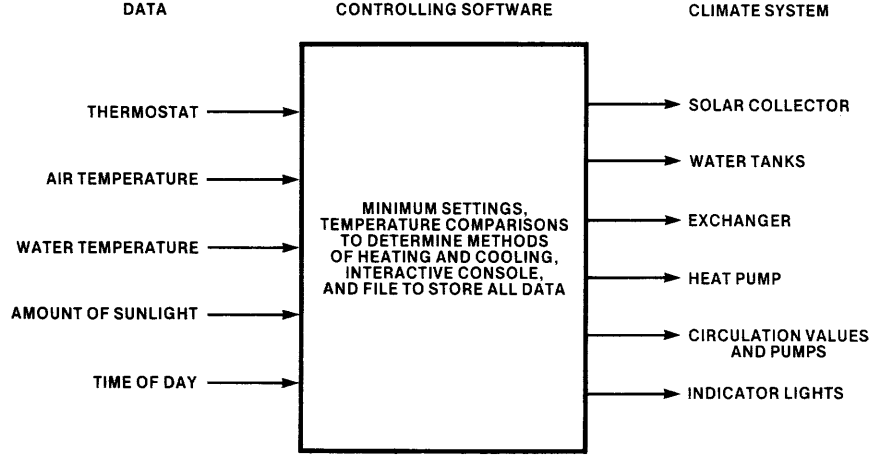
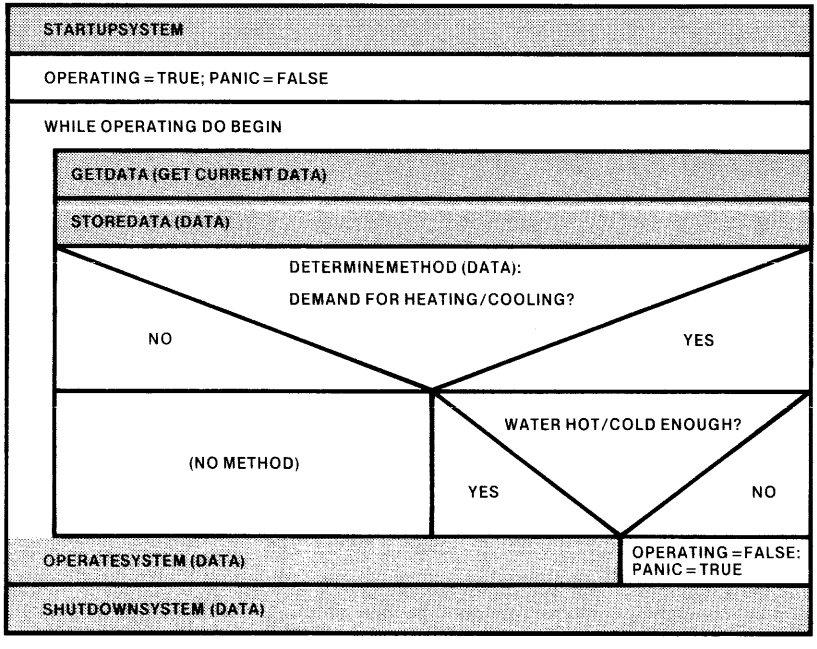


Figure 1-2. Block Diagram of Our Climate Control System 121632-2

A good software definition breaks the problem into solvable tasks. The order in which these tasks must occur also determines the *structure* of the software. If the structure is simple to understand, it will be that much easier to implement and maintain.

A *Nassi-Shneiderman Chart* (shown in figure 1-3) is useful for showing structured blocks of software. Use whatever charts you find useful, but hide many of the details so that you are not locked into doing things a certain way. In our application, we hid all details about data types, input and output, and actual heating or cooling methods. It is most important that we design each module to be self-reliant; that is, a module should not have to know about details hidden in another module, especially details that might change in the future.



Heating Methods

Collector To Exchanger
Tank To Exchanger
Collector To Heat Pump
Tank To Heat Pump
Heated Tank To Heat Pump
No Method (No heating demand)

Data

InsideTemp
ThermostatSetting
CollectorWaterTemp
TankWaterTemp
HeatedTankTemp

AmountOfSun (for collector operation)
Hour, Minute (for collector and immersion heater operation)

Figure 1-3. Nassi-Schneiderman Chart for Our Climate Control Software 121632-3

CHOOSING THE SOFTWARE DEVELOPMENT TOOLS

It is unfortunate but common in this industry to find software development systems sorely lacking in the tools of the trade. There are some systems that force you to put together a whole program in the limited space they allow, and they don't provide the facility to create a library of canned routines that you could use with many different programs.

The Intellec Series III system provides both the ability to put together partial programs, and the facilities to build libraries of routines that you can link to different programs. The concept of *module* is inherent in this system. Whenever you build a partial program, it is a module; and the module can refer to procedures, functions, and variables in other modules found in libraries.

For our application, we do not have to decide on one programming language—the Intellec Series III system supports several, including the high-level languages PL/M-86, Pascal-86, and FORTRAN-86. We can, however, decide whether or not to use modules that already exist, and design our application with that decision, or change the decision later and create another module to replace it.

For example, we already have a module written in PL/M-86 that performs a routine to convert thermocouple voltage into degrees Celsius. We can decide now to tentatively use it, thereby saving time by not coding this routine into our main program. We can also decide, at a later time, *not* to use it, and substitute our own module to do it. The decision to link which module does not have to be made until the main program is finished!

By choosing the right tools you can save time and defer decisions on specific details until you are ready to deal with the details. By deferring such decisions, you keep your software development effort from becoming too cluttered with rigid design decisions, and you keep the effort flexible enough to accept change.

The right tools are (1) appropriate high-level languages to choose from, (2) a way to manage libraries of canned routines, (3) a linker that allows you to link finished (or unfinished) modules in different ways for different applications, (4) a locater that will locate programs in memory for you, yet give you the opportunity to specify locations for sections of the program, (5) a symbolic debugger you can use to test modules and partial programs easily, and (6) an in-circuit emulator to emulate parts of your final product before they exist. The Intellec Series III Microcomputer Development System supports all of these tools.

LANGUAGES

When designing a system using top-down techniques, you think in and express concepts in the highest-level language possible at each refinement step in order to expose the logical concepts and conceal the details.

How can you tell whether you are thinking in a language that is "high-level"? A language is "high-level" for a given application if you use it to define the overall structure of the software. You use a lower-level language to express in great detail each piece of code. With a high-level language you gain a clear understanding of the control structures of the system at the highest level, and you expose logical flaws in the structure that would have led to subtle bugs in the code. With these advantages in mind, it makes sense to start your programming with the highest-level language: English. After you have defined the software in English, you can use a high-level programming language like Pascal, which allows you to express more detailed code in a language that resembles English.

In Chapter 3, we refine our climate control algorithm step by step. We use as a language something called Pidgin Pascal, which is really a language of concise declarative English sentences. Since our control structures can be translated easily from English into Pascal, we decided to use Pascal-86 for our main climate control module (this decision does not have to be final). We do not start the translation into Pascal-86 until after we have tested the logic of our Pidgin Pascal algorithm.

Pascal is a language that resembles the control structures of human thought. We don't think in terms of GO TO branches normally; we consider a job to be a set of tasks to DO WHILE something is true, or to DO UNTIL something is done. IF something is true or false, THEN do one thing; ELSE do another thing. In the CASE of several different problems, solve each one accordingly. Occasionally we might need a disaster bail-out (GO TO a panic routine), but we should be planning our algorithm to take care of disasters elegantly.

The point is: we should think about control structures of a system as structures, not as individual branch statements. Pascal is one language that was designed to express control structures; PL/M is another, and some new versions of FORTRAN can be structured accordingly.

Another point should now be obvious: you should choose the language best suited for the algorithm. For example, our module that operates the climate system has to manipulate pumps and valves to implement a chosen heating or cooling method. This operation module receives the data from the decision-making main module written in Pascal. The operation module might use bits in a word as control signals to send to the procedure that actually interfaces to the hardware of the climate system.

We can code this operation module in assembly language or PL/M, since both languages can easily manipulate bits in a byte or word and respond with appropriate actions based on the pattern of bits. Our final decision will be made later; in the meantime, we will use a test version of the operation module, until prototype hardware for the climate system is ready. Our test version will be written in Pascal, and it will simply display appropriate test information at the console.

It is most likely that the final version of the operation module will not be written in Pascal, since Pascal does not provide bit manipulation operations. We can also guess that the module will not be written in FORTRAN--the advantage FORTRAN has over PL/M is its ability to express complex mathematical formulas. Our climate control system has no complex mathematical formulas.

We would code the final version of the operation module in assembly language if the application required the most efficient use of processor time and memory; however, PL/M programming saves development and maintenance time since it is easier to learn and the programs are easier to read and maintain. The only drawback to PL/M is that it is not as efficient in time or memory, and it cannot compute decimal or real arithmetic. These are not drawbacks in our application, since we do not need decimal or real arithmetic, nor do we have severe speed and memory constraints. The advantage of saving development time and maintenance costs with PL/M far outweigh the advantages of using assembly language.

MODULAR PROGRAMMING

The Intellec Series III system has the linking and locating tools to support modular programming, and the library utility to maintain libraries of canned routines. The decisions we make at this time are not binding, but they can be very helpful: we can decide now what types

of routines will be separated into which modules. We have several design criteria for making these decisions. We must be able to (1) write the routines of one module with little knowledge of the code in other modules, and (2) reassemble and replace any module without affecting other modules. Each module of our climate control system will contain design decisions that are hidden from the other modules so that the decisions are not binding.

With these criteria in mind, we designed our system to have several modules: the **GetData** module gets and stores our data, the **Operation** module performs the actual climate system operation (turning pumps and valves on and off, etc.), and the **Main** module makes the high-level decisions.

The only binding decisions to be made at this time concern the data passed between modules. We try to keep data passing to a minimum, or we try to enforce a data-passing standard that is easy to comply with. In our climate system, we only need to pass a *reference* to a data record to the other modules; the other modules must know what to do with the reference. This data-passing technique is one of the two we can use: pass-by-reference and pass-by-value, which are described in more detail in Chapter 4.

With the Intellec Series III system capabilities, we can design different versions of the same module, test each version, and decide at a later time which version to link with the other modules. We can also defer decisions about the physical memory locations (locations in our final product's memory) for these modules until after we have debugged our prototype hardware with an In-Circuit Emulator (ICE-86 or ICE-88). In some applications, you never have to decide physical memory locations; the Intellec Series III locater can decide them for you.

DEBUGGING AND IN-CIRCUIT EMULATION

At any time during software development, you can test a compiled module using DEBUG-86. In certain cases you will want to alter the module to be self-contained; for example, our main module needs the appropriate data from the **GetData** module, but the **GetData** module is not yet written. We can quickly write a procedure that obtains the data from an interactive session at the console, link this temporary data acquisition module to our main module, and test our main module using DEBUG-86.

When we have our modules coded and compiled, we can test the modules in an ICE-86 or ICE-88 session that can emulate the final product's processor. We can use an ICE-86 emulator if our final product will contain an 8086 processor, or use an ICE-88 emulator if it will contain an 8088 processor (remember, our iAPX 86, 88 applications software can run on either an 8086 or an 8088).

For example, our data module will read data from ports of an 8088 processor in our final product. The ICE-88 emulator can emulate those ports before we ever have a prototype of the final product. Using ICE-88, you can begin testing your software before any prototype hardware exists. As portions of your prototype become available, you can use them and still borrow resources like memory from your Series III system.

With in-circuit emulation, you control, interrogate, revise, and completely debug your product in its own environment, or in a stable environment that emulates the final product's environment. Symbolic debugging is one of the key features of Intellec microcomputer development systems and in-circuit emulators. Symbolic debugging allows you to debug your program using its own symbols and line numbers—you do not have to convert your symbols to physical memory addresses.

USING YOUR FINAL PRODUCT

Intel supplies other tools to help you put together your final product. If you designed your product to have its software in ROM (read-only memory), you can use Intel's Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software to create the PROM (programmable read-only memory) device to hold the software. You can then install your device in an SDK-86 or SDK-88 (System Design Kit with an 8086 or 8088 processor), or in an iSBC (Single Board Computer) system.

You can also run your software in other systems, or in dedicated application environments. For example, you could transfer your software to RAM (random-access memory) on an SDK-86 or SDK-88, or to RAM on an iSBC 86/12A (Single Board Computer system with an 86/12A board), by first using the OH86 utility described in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems* to convert the program to hexadecimal object format. You would then use an appropriate tool to load the software into your execution board (the ICE-86 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957 Interface and Execution Package).

You can also use your Series III development system as the environment for your final software product. The Series III system has an 8086 processor (the "8086 side" or 8086 execution environment) to run iAPX 86, 88 applications software. As soon as you have debugged your software, it is ready to be used in any Series III system. Chapter 6 describes the run-time libraries you use to run your Pascal-86 programs in the Series III environment; you can also supply your own versions of these libraries to run these programs in another execution environment. Chapter 6 also shows how you can link modules and locate them in Series III memory in one easy step to prepare them for execution in the Series III system.

Whatever environment you choose for your final product, Intel provides the appropriate hardware and software tools to develop, debug, and produce your final product. The Intel environments that are suitable for final software products are excellent investments which can be upgraded for the software of the future.

CHAPTER 2
OPERATING THE SERIES III SYSTEM

"The benefits of using a standardized operating system should prove to be as significant as the benefits of using standardized microcomputer hardware. Development and programming costs will be reduced substantially, and you will have an upward compatible interface for future products."

—Andrew S. Grove, President of Intel Corp.

As you learned in Chapter 1, the Intellec Series III Microcomputer Development System contains the tools you need to develop software for your application. To use these tools, you must operate the system; that is, use the system's commands and utility programs.

A large multi-user computer system can serve as a useful analogy to point out the difference between *programming* the system and *using* the system. A working system usually supports both kinds of activities. In such a large system, one or more programmers might be designing programs to run on the system. One or more users (who might also be programmers) might be simply *using* the programs that have already been developed for the system. Obviously, a user does not need to know complicated details about the system to use it, whereas a programmer needs to know such details to write programs for the system.

The Series III system has most of the capabilities of larger systems, but it is only used by one person at a time. This person could be using the system and its programs, or writing programs for the system.

We wrote this chapter for people who need to know how to use the system. This information is important to anyone using or programming the system, but it is not burdened with details that a first-time user does not need. As we describe system commands and utility programs, keep in mind that we swept the more complicated details under the rug to keep the chapter easy to read. For more details on all of the system commands, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

TURNING ON YOUR SYSTEM

Every microcomputer has something called an operating system, which is sometimes called a supervisor or monitor. It is usually the first piece of software you use—the software that is controlling your computer when you first turn it on, and the software that responds to your first typed command. Typically, you type a command to ask the computer for a list of the files you have on disk, or to execute a particular program.

When you turn on the Intellec Series III Microcomputer Development System, the following message appears on your screen:

```
SERIES II MONITOR, Vx.y
```

This message tells you that the monitor is up and running (the *x* and *y* represent version numbers). The *monitor* is a piece of software that gives you direct control of the Series III hardware. Although you can use the monitor for debugging and other operations, for your purposes the monitor performs one quick activity: it loads the operating system from the system disk in drive 0 into the computer when you push the RESET button.

To load or "boot" the system, you need a *system disk*: a disk that holds the operating system files. Intel supplies a flexible disk labeled "ISIS-II Operating System" which is your system disk. One of the first operations you will perform is a *formatting* operation to create another copy of the system disk—a copy designed to be used with the examples in this book.

You insert the "ISIS-II Operating System" disk into flexible disk drive 0, and push the RESET button (for more detailed instructions, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*). The following message should appear on your screen:

```
ISIS-II, Vx.y
```

The operating system for the Intellec Series III Microcomputer Development System is called ISIS-II ("ISIS" stands for Intel Systems Implementation Supervisor). The *x* and *y* represent version numbers. ISIS-II is actually a version of the ISIS operating system that runs on older Intellec systems (the Intellec Microcomputer Development System and the Intellec Series II Microcomputer Development System).

ISIS-II manages your use of the software tools supplied with the system. Using ISIS-II, you can run utility programs like CREDIT (a text editor) to write programs, or LINK86 to link program modules into a final program. You also use ISIS-II to run compilers like the Pascal-86 and PL/M-86 compilers, and to run your own developed programs. You can also make copies of programs and control the devices attached to your system by using ISIS-II commands.

In most computer operations, you manipulate *files*, which are collections of information. Every file has a name, called a *filename*. Files and filenames are described in detail after an introduction to the most popular operating system command—the DIR command.

You use the DIR command to display a directory of filenames on a certain disk. You use other commands to perform file management operations. For example, you use the COPY command to make copies of files, or to send copies of files to the printer to be printed. You use the RENAME command to change a file's filename. This chapter explains some of these commands.

THE DIRECTORY LISTING

The first command to learn is the easiest to use—the DIR command. When your system is up and running, you will see a dash (-) on the left side of the screen. This dash is called a *prompt*, and it tells you that the system is ready for your next command. You can now type the DIR command by typing the letters "DIR", and execute the DIR command by pressing the RETURN key. The following example is shown in blue to show that it is something *you* type; the part of the example that is in ordinary black shows what the system displays. The symbol "<cr>" means that you must press the RETURN key ("cr" stands for the "carriage return" key found on most typewriters).

```
-DIR<cr>

DIRECTORY OF :F0:970003.06
NAME  .EXT  BLKS  LENGTH ATTR     NAME  .EXT  BLKS  LENGTH ATTR
SYSTEM.LIB    24    2849 WS       FPAL  .LIB    74    9125 W
PLM80 .LIB    45    5615 W
                              143
1317/4004 BLOCKS USED
```

When you execute the DIR command, the system displays a list of *filenames* called a *directory listing*. These filenames are the names of the files that are stored on the disk in drive 0. The DIR command displays the directory for the disk in drive 0 unless you specify another drive number with the command, as explained in the next paragraph. Disk drive 0 is usually the drive that holds the system disk (the disk that contains the system files), so by executing the DIR command by itself, you get a directory listing of some of the files on the system disk in drive 0 (not all, because some files are invisible).

You can use the DIR command to display the directory listing for disks in other disk drives, and for a hard disk subsystem, by specifying the drive number with the DIR command. For example, to see a directory listing for a disk in drive 1, insert the "CREDIT ISIS-II CRT-Based Text Editor" disk in drive 1, and type the following command:

```
-DIR 1<cr>

DIRECTORY OF :F1:970049.02
NAME  .EXT  BLKS   LENGTH ATTR      NAME  .EXT  BLKS   LENGTH ATTR
CREDIT       156    19470          CREDIT.HLP    25     2985 W
ADDS  .MAC     3      171 W        MICROB.MAC     3      158 W
VT52  .MAC     3      163 W        VT100 .MAC     3      190 W
1510T .MAC     3      165 W        1510E .MAC     3      170 W
LEAR  .MAC     2      123
                              201
310/4004 BLOCKS USED
```

The *Intellec Series III Microcomputer Development System Console Operating Instructions* give details about typical hard disk and flexible disk configurations, and the associated drive numbers.

On each line of the directory listing, you are shown the name of a file. Some of these filenames are followed by a three-character *extension*, which is an optional identifier used to show types of files. These extensions are sometimes required for certain files that are used with certain programs. The section on filenames (following disk formatting) explains some of these extensions.

Following each filename entry are three columns labeled "BLKS", "LENGTH", and "ATTR". The "BLKS" column tells you how many "blocks" of space the file occupies, where one block equals 128 bytes. The "LENGTH" column gives the actual number of bytes occupied by the file. At the bottom of the listing appears a message showing how many blocks are used out of the total number of blocks available on the disk. You can use block figures to describe file sizes and to determine whether a file of a given size (in blocks) will fit on a disk.

The "ATTR" column might be empty, or it may contain a "W", "S", or both. This column tells you the *attributes* of each file—certain characteristics that govern the file's use. If a file has the "W" attribute, it is *write-protected*; that is, you cannot write to or delete (overwrite) the file. If a

file has the "S" attribute, it is a *system* file that occurs on most system disks. You use the ATTRIB command and file attributes to protect your files from inadvertant delete or write operations, and to designate certain files as system files (as shown in the next section).

There are other files not displayed through normal use of the DIR command. These files are *invisible*; that is, they have the I attribute. You can see them if you use a special form of the DIR command:

```
-DIR I<cr>

DIRECTORY OF :F0:970003.06
NAME   .EXT  BLKS    LENGTH ATTR      NAME   .EXT  BLKS    LENGTH ATTR
ISIS   .DIR    26      3200   IF       ISIS   .MAP     5       512   IF
ISIS   .TO     24      2944   IF       ISIS   .LAB    54      6784   IF
ISIS   .BIN    94     11740  SIF       ISIS   .CLI    20      2407  SIF
ATTRIB         40      4909 WSI        COPY           69      8489 WSI
DELETE         39      4824 WSI        DIR            55      6815 WSI
EDIT           58      7240 WSI        FIXMAP         52      6498 WSI
HDCOPY         48      5994 WSI        HEXOBJ         34      4133 WSI
IDISK          63      7895 WSI        FORMAT         62      7794 WSI
LIB            82     10227 WSI        LINK          105     13074 WSI
LINK   .OVL    37      4578 WSI        LOCATE        120     15021 WSI
OBJHEX         28      3337 WSI        RENAME         20      2346 WSI
SUBMIT         39      4821 WSI        SYSTEM.LIB     24      2849 WS
FPAL   .LIB    74      9125 W          PLM80  .LIB     45      5615 W
                                  1317
1317/4004 BLOCKS USED
```

The "I" is called a *switch*, and it displays files that have the invisible attribute. When you specify the "I" switch in a DIR command, the DIR command displays all of the filenames, including ones that are invisible. The files that weren't displayed during the execution of a normal DIR command are now displayed, along with their attributes (one of which is the "I" attribute). The "F" attribute is reserved for system files that are used to format the disk. These files are called *format files*, and you should never alter their attributes.

You should practice using the DIR command, and at the same time, take a look at the directory listings for each disk you received. Take each flexible disk, insert it into drive 1, and type "DIR 1" to see the directory listing.

NOTE

If you have a hard disk subsystem with only one flexible disk drive, use the following form of the DIR command:

```
-DIR P<cr>
LOAD SOURCE DISK, THEN TYPE (CR)
```

Take out the system disk and insert the disk whose directory you want to display, then press the RETURN key. After the directory listing, the following message appears:

```
LOAD SYSTEM DISK, THEN TYPE (CR)
```

Put the system disk back into the flexible drive. To see other disk directory listings, repeat these steps for each disk.

FORMATTING DISKS

We present in this section a typical scheme for storing copies of your Intel-supplied files on either hard disk or flexible disk. Subsequent examples in this book assume that certain programs reside in hard disk drives 0 and 1, or flexible disk drives 0 and 1. You may use your own scheme and distribute files over disk drives as you wish, but in order to type the examples as they are, you must use the scheme presented here. If you understand the use of pathnames as described in this chapter, and if you use the COPY command correctly, you can copy files to disks in any distribution scheme you choose, and still use the examples in this book as long as you substitute your own pathnames.

If you are using a hard disk subsystem, you must follow the procedures in the *Intellec Series III Microcomputer Development System Console Operating Instructions* to install your disk platters and prepare them for use. If you are using flexible disk drives, you should refer to the same manual for instructions on the care and insertion of flexible disks. This section shows you how to prepare one hard disk platter or one flexible disk as the system disk, and another hard disk or flexible disk as a non-system disk (to hold your files and other programs).

Hard Disk Subsystem Users

Follow the instructions in the *Intellec Series III Microcomputer Development System Console Operating Instructions* to install and power-up your hard disk subsystem. When the hard disk subsystem is ready, insert your "ISIS-II Operating System" flexible disk into the flexible disk drive of your computer, and hit the RESET button. When the dash (-) prompt is displayed (after the ISIS-II message appears), type the following command:

```
-:F4:FORMAT :F0:SYSTEM.HDK S FROM 4<cr>
```

This command prepares ("formats") drive 0 of the hard disk to be the system disk. We chose "SYSTEM.HDK" for its name, but you can use any name that has at most six characters to the right of the period, and three to the left. When this operation is finished, and the system displays the dash (-) prompt, use the following command to transfer your system files to drive 0 of the hard disk:

```
-:F4:COPY :F4:*.* TO :F0:<cr>
```

This command will copy all of the files from the "ISIS-II Operating System" disk in the flexible disk drive (called drive 4) to the hard disk drive 0. When this operation is finished, you can remove the flexible disk from drive 4 and insert another flexible disk. In addition to the "ISIS-II Operating System" disk, you should also copy files from the "CREDIT ISIS-II CRT-Based Text Editor" disk, the "Resident 8086/8087/8088 Macro Assembler" disk, and the "Resident 8086/8088 Utilities and Linkage Libraries" disk. For each flexible disk, execute the following command:

```
-COPY :F4:*.* TO :F0:<cr>
```

This command copies all files from the flexible disk to drive 0 of the hard disk. If you insert each flexible disk you received and perform this operation, you will have in drive 0 all of the files from those flexible disks. If you want to be more selective about the files you are copying to drive 0, read the "Copying Files" section in this chapter.

You should also copy files to drive 1 of the hard disk subsystem. First, you must prepare drive 1 by typing the following command:

```
-FORMAT :F1:PROG86.HDK<cr>
```

We chose "PROG86.HDK" for its name, because we intend to use it for our iAPX 86,88 application programs. If you are a Pascal-86 user, insert the "Pascal-86 Compiler and Run-Time Libraries" disk into flexible disk drive 4 and copy all of the files to the hard disk drive 1:

```
-COPY :F4:*.* TO :F1:<cr>
```

Do the same operation with your PL/M-86 flexible disk, and any other disks you have for your Series III system. The hard disk platters have plenty of room for your own files.

Flexible Disk Users

This section assumes that you have at least two double-density flexible disk drives. If you have single-density drives, you may run out of room if you try these examples. Refer to instructions in the *Intellec Series III Microcomputer Development System Console Operating Instructions* for information about flexible disks.

To bring up your system, insert the "ISIS-II Operating System" disk into drive 0 and push the RESET button. The ISIS-II message should appear, followed by the dash (-) prompt. Insert a blank disk into drive 1, and type the following FORMAT command:

```
-FORMAT :F1:SYSTEM.FLX S<cr>
```

This command creates a new system disk and automatically copies from drive 0 all files that have the "S" attribute. When this operation is finished, you can test your new system disk by removing the "ISIS-II Operating System" disk from drive 0, inserting your new system disk into drive 0, and pushing the RESET button to restart the system.

With your new system disk in drive 0, insert the "CREDIT ISIS-II CRT-Based Text Editor" disk into drive 1, and type the following command:

```
-COPY :F1:CREDIT TO :F0:<cr>
```

This command copies the program CREDIT to our system disk in drive 0. Remove the CREDIT disk from drive 1 and insert the "Resident 8086/8087/8088 Macro Assembler" disk into drive 1. Type the following command:

```
-COPY :F1:RUN.* TO :F0:<cr>
:F1:RUN COPIED TO :F0:RUN
:F1:RUN.OV0 COPIED TO :F0:RUN.OV0
```

This command copied two files at once—RUN and RUN.OV0—from drive 1 into drive 0. You need both files in order to use the special RUN command described later.

Remove the flexible disk from drive 1 and replace it with the "Resident 8086/8088 Utilities and Linkage Libraries" disk. Now type the following commands:

```
-COPY :F1:LIB86.86 TO :F0:<cr>
:F1:LIB86.86 COPIED TO :F0:LIB86.86
-COPY :F1:LOC86.86 TO :F0:<cr>
:F1:LOC86.86 COPIED TO :F0:LOC86.86
-COPY :F1:LINK86.86 TO :F0:<cr>
:F1:LINK86.86 COPIED TO :F0:LINK86.86
```

You now have a system disk that is complete for the examples in this book. You still need another disk for the Pascal-86 compiler if you have one, to hold the compiler, the run-time libraries, and your sample program used in examples in this book. To prepare a blank disk to be a non-system disk, insert the blank disk into drive 1 and type the following command:

```
-IDISK :F1:PASC86.FLX<cr>
NON-SYSTEM DISK
```

The IDISK command can prepare both system and non-system disks, but it does not automatically copy files except the ISIS-II format ("F") files.

With the new blank disk in drive 1 and the system disk in drive 0, you could put the "Pascal-86" disk in drive 2 (if you have a drive 2). The following example assumes that you only have drive 0 and 1. To copy the files from the "Pascal-86" disk, first type the following command (don't forget the "P"!):

```
-COPY *.* TO :F1: P<cr>
LOAD SOURCE DISK, THEN TYPE (CR)
```

The system pauses (because you specified a "P" at the end of the command), and waits for you to insert the "source" disk into drive 0. The "source" disk in this case is the "Pascal-86" disk, assuming you are a Pascal-86 user (if you're using PL/M-86 only, you would substitute your PL/M-86 disk in this example). Insert this flexible disk into drive 0 and press the RETURN key.

```
LOAD OUTPUT DISK, THEN TYPE (CR)
```

Since the disk to receive the Pascal-86 files is already in drive 1, you only have to press RETURN. The system should then display the above messages (along with the "COPIED" messages) every time it returns to drive 0 to copy more files—all you have to do is continue to press RETURN until the entire copy operation is finished.

You can repeat these steps with another blank disk for the PL/M-86 compiler. After these formatting and copying operations, you should have disks that match the ones we used for the examples in this book.

FILENAMES, PATHNAMES, AND FILE ATTRIBUTES

Most operating system commands manipulate files. A file can be any collection of information including text, numeric data, program instructions, and combinations of all of these. You refer to a file by using a filename, and filenames follow certain naming conventions so that the name of a file also tells you the probable use of the file.

Filenames can have six or fewer characters, followed by an optional period and three-character extension. Extensions (and filenames themselves) follow certain conventions, but there are no fixed rules. Certain extensions are necessary for certain programs, as you shall see in the following discussion. The file naming conventions are relaxed so that you have the flexibility to create your own file naming conventions for your particular application.

The term filename refers to both the name of the file and the extension, if any. Each new file you create must have a unique name for that disk (that is, you can have two files with the same name on two different disks, but not on the same disk). Some extensions have specific meanings to utility programs in the system, but these should not cramp the style with which you impose your own naming conventions. Here are the extensions that mean something to Intel-supplied programs:

- The ".BAK" extension denotes a backup of a text file created by CREDIT, the text editor described in Chapter 3. For example, LETTER.BAK is the backup file for the text file LETTER.TXT (a text file does not need a particular extension, but "TXT" helps identify it).
- The ".MAC" extension denotes a "macro" file used with certain programs like CREDIT. The ".MAC" files supplied with CREDIT enable you to use CREDIT on other non-Intel consoles connected to Intel microcomputers.
- The ".OV0" extension denotes an "overlay" file used with certain programs.
- The ".OBJ" extension denotes an *object module*, which is created by a compiler or assembler, as described in Chapters 4 and 5. Compilers and assemblers also create files with the ".LST" extension to denote program listings (also described in Chapters 4 and 5). Examples are PROG1.OBJ and PROG1.LST.
- The ".LNK" extension denotes a collection of object modules that were linked together by the linker utility program, described in Chapter 6. When you run the locater utility program on a file with the ".LNK" extension, the locater strips the ".LNK" extension away, leaving only the file's name without an extension.
- The ".LIB" extension denotes a library module that is maintained by the library utility. Library modules are described in Chapter 6.
- The ".86" extension denotes a program that should be executed in the 8086 execution mode (the "8086 side" of the system). Examples are PASC86.86 and PROG1.86.

Most completed programs have either the ".86" extension (if they run in the 8086 execution mode, described later in this chapter), or no extension. For example, the DIR command is actually a program named "DIR" without any extension.

In addition to the above extensions, we use the ".SRC" extension to denote a special text file that holds program instructions called source statements. A ".SRC" (for "source") file can be created by CREDIT and filled with assembly language, PL/M, or Pascal source statements; you can then compile this file to create an object module (".OBJ" file), as described in Chapters 4 and 5.

You can find more extensions explained in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

You can display the directory information for a single file by using a version of the DIR command, DIR FOR:

```
-DIR FOR SUBMIT<cr>
DIRECTORY OF SYSTEM.FLX
NAME   .EXT   BLKS     LENGTH   ATTR
SUBMIT          39       4821
```

The file SUBMIT is in the directory for drive 0, so the DIR command had no trouble finding it. However, if SUBMIT happened to be on another disk in drive 1, you would have to tell DIR to look in drive 1 for the file. To do this, you use a *pathname*—a filename with a directory specifier. Whenever you specify a filename for a command or utility program, if the filename is in the directory for drive 0, you only have to specify the filename. If the filename is *not* in the directory for drive 0, you have to specify a pathname.

A pathname consists of a directory specifier and a filename:

```
:F1:PROG1.SRC
```

The ":F1:" part of the pathname is the directory specifier. The "1" stands for disk drive 1. Therefore, to get the directory information for the file PROG1.SRC on the disk in drive 1, type the following command:

```
-DIR FOR :F1:PROG1.SRC<cr>
```

The number of separate disk drives depends on the configuration of your system. The format for directory specifiers is:

```
:Fn:
```

where *n* can be any drive number from 0 to 9 (0 would refer to drive 0, which does not have to be specified).

If you type a directory specifier that is incorrect, you get an error message. For example, suppose you typed the following DIR command with an incorrect directory specifier in the pathname for PROG1.SRC:

```
-DIR FOR :G1:PROG1.SRC<cr>
:G1:PROG1.SRC, UNRECOGNIZED DEVICE NAME
```

The "UNRECOGNIZED DEVICE NAME" is the ":G1:" directory specifier. ISIS-II thinks you are referring to a device by that name, and there is no device by that name. You will find legal device names in the section that describes copying files.

Another common error occurs when you forget to use the word FOR in the DIR command when you are trying to get the directory information for one file. For example, you might type this command:

```
-DIR :F1:PROG1.SRC<cr>
```

and get this error message:

```
:F1:PROG1.SRC, UNRECOGNIZED SWITCH
```

Since you forgot the word FOR, ISIS-II thinks you are trying to specify a switch to the DIR command—and it doesn't recognize ":F1:PROG1.SRC" as a switch! Switches are usually only one letter or number. Valid switches are described with the DIR command in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

The ATTRIB command assigns attributes to files. You use file attributes to protect your files from accidental deletions or modifications. You use the ATTRIB command to turn on or off the file attributes that are described below:

- "W" for write-protected (The file cannot be written to, modified, renamed, or deleted unless this attribute is turned off.)
- "I" for invisible (The file's name and information is only displayed when you execute the DIR command with the "I" switch.)
- "S" for system files (System files on a source disk are automatically copied to the output disk in a FORMAT operation with the "S" switch.)
- "F" for format files (Format files are automatically copied by the FORMAT and IDISK commands to format disks. *Do not* turn off this attribute, nor use it with your own files, unless you've read its description in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.)

For example, to turn on the "W" (write-protected) attribute for file CREDIT, type the following command:

```
-ATTRIB CREDIT W1<cr>
    FILE                CURRENT ATTRIBUTES
:F0:CREDIT                      W
```

To turn off the "W" attribute, specify a 0 instead of a 1:

```
-ATTRIB CREDIT W0<cr>
    FILE                CURRENT ATTRIBUTES
:F0:CREDIT
```

RENAMING AND DELETING FILES

Occasionally you will have a file with a name that needs to be changed for some reason. For example, you could rename LIB86.86 to LIBRY.86 by using the RENAME command:

```
-RENAME LIB86.86 TO LIBRY.86<cr>
```

The RENAME command always expects to see the original name first, followed by a space, then the keyword TO, followed by another space, and finally the new name. Both names must be legal Series III filenames or pathnames. You cannot RENAME a file in one directory to a name with a different directory specifier—the new name must have the same directory specifier.

If you actually renamed LIB86.86 to LIBRY.86, you should rename it back to LIB86.86, the name used for it in future examples.

The DELETE command is very simple. Since there are no files in your system disk that you can delete, we'll use in this example a file that doesn't actually exist on your disk:

```
-DELETE SAMPLE.TXT<cr>
:F0:SAMPLE.TXT DELETED
```

The DELETE command will irrevocably delete the file you specify. When you DELETE a file, you cannot retrieve it—it's gone. Use DELETE with caution.

To delete a file on a disk in a drive other than drive 1, simply use the appropriate pathname for the file. For example, suppose the file JUNK.TXT existed on the disk in drive 1. You would use this command to delete it:

```
-DELETE :F1:JUNK.TXT<cr>
:F1:JUNK.TXT DELETED
```

You can also delete many files at once. As you will see in Chapter 3, CREDIT creates a backup file with a ".BAK" extension for every file you edit. At some point (to utilize disk space), you might want to delete all of your backup files at once. For example, to delete all of the backup files on the disk in drive 1, use the following command:

```
-DELETE :F1:*.BAK<cr>
```

The asterisk (*) matches any name, and the ".BAK" extension matches only filenames with that extension. This "filename matching" technique is shown in more examples to come.

COPYING FILES TO DISKS AND DEVICES

The COPY command is useful for several operations:

- To make a copy of a file
- To send a copy of a file to another hard or flexible disk
- To send a copy of a file to a device like a printer or a paper tape punch
- To receive a file from a sending device like a paper tape reader
- To copy all non-system files from one disk to another

The COPY command is similar to the RENAME command. For example:

```
-COPY :F1:PROG1.SRC TO :F1:PROG1.BAK<cr>
COPIED :F1:PROG1.SRC TO :F1:PROG1.BAK
```

The COPY command expects to see the name of the *source* file or device (the source of the information to be copied, not to be confused with a *source file* of program source statements), followed by a space, then the keyword TO, followed by a space, and then the name of the *output* file or device (the destination of the copied information). After the output (or destination) file or device name, you can optionally specify a *switch* like "P" for pause, or "U" for update (these switches are explained in the *Intellec Series III Microcomputer Development System Console Operating Instructions*).

More frequent uses of the COPY command are copying to devices, copying files onto other disks, and copying all the files in one disk to another disk. In all of these cases, you must specify the device or disk that is the source of the information, and the device or disk that will receive the copied information. For example, the RENAME program is in the directory for drive 0. If you want to put a copy of it on the disk in drive 1, you would type the following command:

```
-COPY RENAME TO :F1:<cr>
COPIED :F0:RENAME TO :F1:RENAME
```

When the source file is in the directory for drive 0, you don't have to specify :F0: for the file, because :F0: is the default directory if none is specified. When the destination is a disk in another drive, you have to specify the directory (in this case, :F1:) for the destination file.

In the example above, the file RENAME was copied to the disk in drive 1, and the drive 1 version has the same name. When you make copies of files for other disks, you will probably want the files on the other disks to have the same name. To keep the same name for the copy, specify only the destination directory without a new filename, as we did in the above example.

To change the name for the copy, specify a different name with the destination directory. For example, if you want to use RNAM as the name of the new copy in drive 1, use this command:

```
-COPY RENAME TO :F1:RNAM<cr>
:F0:RENAME COPIED TO :F1:RNAM
```

In the last two examples, the RENAME file was copied from drive 0 to drive 1. To demonstrate another feature of the COPY command, we will first copy a file from drive 1 to drive 0 (you should also try this example):

```
-COPY :F1:PROG1.SRC TO :F0:<cr>
:F1:PROG1.SRC COPIED TO :F0:PROG1.SRC
```

Let's suppose that we used CREDIT (described in the next chapter) to modify the new copy of PROG1.SRC on the disk in drive 0 (:F0:PROG1.SRC), and that we want to copy the newly-modified PROG1.SRC to the disk in drive 1. Since we already have a PROG1.SRC on the disk in drive 1, we might want to DELETE that one first, then COPY the newly-modified one to the disk in drive 1. However, let's suppose that we forgot to DELETE the old one first, or that we don't even know the old one exists on the disk in drive 1. We would type the following COPY command:

```
-COPY PROG1.SRC TO :F1:<cr>
:F1:PROG1.SRC FILE ALREADY EXISTS
DELETE?
```

The COPY command found :F1:PROG1.SRC, and now it is asking us if we want to delete it in order to replace it with the newly-modified :F0:PROG1.SRC. We type a "Y" (or "y") to delete the old one and replace it with the new one, or type an "N" (or "n" or any other letter) to keep the old one and abort the copy operation. In this case, we want to replace the old :F1:PROG1.SRC with the newly-modified :F0:PROG1.SRC, so we type a "Y" followed by the RETURN key:

```
DELETE?Y<cr>
:F0:PROG1.SRC COPIED TO :F1:PROG1.SRC
```

By using this feature of the COPY command, you can selectively update existing files by typing "Y" (or "y") for the ones you want to update, and "N" (or "n") for the ones you don't want to update.

To send a file to a device like a line printer or a paper tape punch, specify the device name as the destination device:

```
-COPY PROG1.SRC TO :LP:<cr>
:F0:PROG1.SRC COPIED TO :LP:
-COPY PROG1.SRC TO :HP:<cr>
:F0:PROG1.SRC COPIED TO :HP:
```

The device name :LP: is the name of the line printer. The second example sends the file to the high-speed paper tape punch, whose device name is :HP:. For other device names, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

The COPY command can also be used to copy several files at once, if you make use of *wild card filenames*. You can specify a wild card filename with the DELETE, RENAME, COPY, DIR, HDCOPY, and ATTRIB commands. A wild card filename matches a group of filenames in order to perform the action on several files at once. For example:

```
-COPY *.* TO :F1:<cr>
```

This command copies all of the files in directory :F0: to directory :F1:.

NOTE

To perform this example, you should insert a new disk into drive 1 and use the IDISK command (described in the next section in more detail) to prepare the new disk in drive 1 before copying all of the files in drive 0 to it.

The above example demonstrates use of the wild card filename "*.*". The first asterisk will match any number of characters in the name, and the second asterisk will match any three characters in the extension of the filename.

You can also specify some of the characters of a filename in a wild card filename. For example, if you wanted to copy all of the files that have ".SRC" as an extension, you would use this COPY command:

```
-COPY *.SRC TO :F1:<cr>
```

There are other wild card filenames that are described in detail in the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

There are several ways to copy entire disks. The IDISK and FORMAT commands are used to prepare ("format") disks for use in the Series III system. The IDISK command will simply format the disk as a system or non-system disk, copying over to the new disk only the format ("F") files that are needed to format the new disk as a system or non-system disk.

The FORMAT command will format a new disk as a system or non-system disk depending on the source disk implied (drive 0) or specified (using FROM). The FORMAT command will also copy certain files from the source disk. FORMAT with the "S" switch copies all files from the source drive that have the "S" (system) attribute. See the previous section on formatting disks for an example of the FORMAT command with the "S" switch.

To copy *all* of the files from the source disk, use the "A" switch. To show an example of the FORMAT command with the "A" switch, insert another blank disk into drive 1, and type this command:

```
-FORMAT :F1:SYSTEM.BAK A<cr>
SYSTEM DISK
.
.
.
```

In this example, the FORMAT command formats the disk in drive 1 as a system disk because the source disk in drive 0 is a system disk. The new disk is called SYSTEM.BAK. The "A" switch specified that *all* files from the source disk in drive 0 should be copied to SYSTEM.BAK in drive 1.

So far, all of the FORMAT examples used the disk in drive 0 as the source disk. If, for example, you have more disk drives and you want to FORMAT a disk in drive 1 using as a source a disk in drive 2, you would specify FROM 2 as in the following example:

```
-FORMAT :F1:MYDISK A FROM 2<cr>
```

In a previous section on formatting disks, we described formatting operations to prepare your disks in a way that conforms with our examples. We do not necessarily recommend this formatting scheme for your particular configuration. Chapter 2 of the *Intellec Series III Microcomputer Development System Console Operating Instructions* gives you the specific details for setting up disks in any Series III configuration, using the IDISK, FORMAT, and HDCOPY commands.

EXECUTING COMMANDS AND PROGRAMS

The Series III has two processors: the 8085 (8-bit processor), and the 8086 (16-bit processor). You can execute 8085-based programs in the 8085 environment (which is called the *8085 execution mode* or "8085 side"), and execute 8086-based (or 8088-based) programs in the 8086 environment (which is called the *8086 execution mode* or "8086 side"). When you type the filename of a program as a command (as you have been doing when you type "DIR" or "COPY"), you are executing the program in the 8085 execution mode. In order to execute a program or command in the 8086 execution mode, you must prefix the command RUN to the program or command you wish to execute. For example, if you want to execute LIB86.86, the 8086 librarian (which can only be run in the 8086 execution mode), you would type the following:

```
-RUN LIB86<cr>
```

You should notice two things that are different: the use of the command RUN, and the fact that you do not have to supply the ".86" extension for LIB86.86 when you RUN it.

The RUN command is actually a program supplied on the system disk that activates the 8086 execution mode. When you supply a filename with no extension, the RUN command automatically attaches the ".86" extension to the name you supplied, and looks for the file by that name (i.e., the name with the ".86" extension). This is a protection feature you can use for your 8085 and 8086 programs: you can use the same name for both, with an ".86" extension for the 8086 program and no extension for the 8085 program. When you specify filenames without extensions, RUN only looks for files that are *supposed* to run in the 8086 execution mode; i.e., files with the ".86" extension. This protection feature assumes that you would put the ".86" extension on files that are meant to run in the 8086 execution mode.

If you actually executed LIB86.86, the following would appear on your screen:

```
-RUN LIB86<cr>
SERIES III 8086 LIBRARIAN, Vx.y
*
```

The LIB86.86 program is now in control. To leave this program and return to the operating system, use the librarian's EXIT command:

```
*EXIT<cr>
-
```

You can, of course, supply an extension with the filename you supply to the RUN command, and the RUN command would not supply the ".86" extension. For example, if you have an 8086 program called MYPROG.PRG and you want to run it in the 8086 execution mode (on the "8086 side"), you would type the following:

```
-RUN MYPROG.PRG<cr>
```

You can also RUN a program whose filename has no extension. However, you must supply a period to show that there should be no extension (i.e., to keep RUN from supplying the ".86" extension). The following is an example, assuming that the name of the program is not MYPROG.PRG but is MYPROG:

```
-RUN MYPROG.<cr>
```

When you use RUN as a command (as in the examples above), the 8086 execution mode is turned on for program execution, and when the program terminates, the 8086 execution mode also terminates, returning you to the 8085 execution mode. The 8085 execution mode is controlled by ISIS-II, which signifies its control by displaying the dash (-) prompt. You can, however, use RUN as a program and *stay in the 8086 execution mode*.

To use RUN as a program, simply type RUN:

```
-RUN<cr>
ISIS-II RUN 8086, Vx.y
>
```

The RUN program displays a sign-on message and its own prompt, the right angle (>) bracket, to signify that the RUN program is now in control. This mode is called the *interactive 8086 mode*, and you use it to execute more than one program. When you get the angle (>) prompt, you can type the filename of a program in order to execute the program. The rule stated above about the use of the ".86" extension still applies.

The following example shows the execution of several 8086-based programs: PASC86.86 (the Pascal-86 compiler), LINK86.86 (the linker and binder), and PROG1.86 (the assembled, linked, and bound program). If your Pascal-86 disk (containing the compiler, run-time libraries, and sample PROG1.SRC) is in drive 1, and LINK86.86 is on the disk in drive 0, you can type this example exactly as presented. The LINK86.86 command line shows use of the "&" continuation character that allows you to type commands over several lines without executing them. The EXIT command turns off the 8086 execution mode and returns you to ISIS-II and the 8085 execution mode:

```
-RUN<cr>
ISIS-II RUN 8086, V1.0
>:F1:PASC86 :F1:PROG1.SRC<cr>

.                          This executes PASC86.86 to compile
.                          PROG1.SRC, both of which are on the
.                          disk in drive 1. The Pascal-86
                           compiler displays some information.

>LINK86 :F1:PROG1.OBJ,:F1:P86RN0.LIB,:F1:P86RN1.LIB,&<cr>
>>:F1:P86RN2.LIB,:F1:P86RN3.LIB,:F1:E8087.LIB,&<cr>
>>:F1:E8087,:F1:LARGE.LIB TO :F1:PROG1.86 BIND<cr>

.                          This example shows use of the ''&''
.                          command line continuation character.
.                          The entire command links the object
                           program with the run-time libraries,
                           and binds the linked module to form
                           an executable program called
                           PROG1.86
```

```
>:F1:PROG1<cr>              This command executes :F1:PROG1.86

Farenheit temperature is:72<cr>
.
.                          PROG1.86 is a temperature conversion
.                          program. When it asks for more
                           input, you can stop the program by
                           typing ''N''.

Another temperature input?   :N<cr>

>EXIT<cr>                  This terminates the 8086 execution
                           mode and returns you to the 8085
                           execution mode.
```

In the above examples, commands and programs were executed directly by typing the command name or program name (sometimes followed by the name of a file to be acted upon by the command or program; this is called an *actual parameter*). When you execute a program or command directly, it is called *interactive execution*.

In many cases you will want to submit a batch of commands or programs to be executed in a sequence. The SUBMIT command allows you to submit a file of commands as a job to be handled by the system without any interaction on your part. This is called *non-interactive execution*.

You use SUBMIT by first providing a file of commands to be executed—a *command sequence definition file*, which has the extension ".CSD". This file can contain operating system commands, parameters for the commands, and *comments*—any line starting with a semicolon (;), or any text following a semicolon, is a comment, not an executable command. In the following example, note that the first command is the RUN command, which starts the 8086 execution mode, and that blank spaces in command lines are allowed to improve readability:

```
;
; CSD file to SUBMIT for linking a Pascal-86 object module
; to run-time support libraries (with E8087 emulator)
;
;      Parameter 0 = :fn:myprog
;      Parameter 1 = drive containing run-time libraries
;
;

RUN
LINK86 %0.OBJ, &
       :F%1:P86RN0.LIB, &
       :F%1:P86RN1.LIB, &
       :F%1:P86RN2.LIB, &
       :F%1:P86RN3.LIB, &
       :F%1:E8087.LIB, &
       :F%1:E8087, &
       :F%1:LARGE.LIB TO %0.86 BIND

EXIT
;
; Returns to 8085 execution mode.
;
```

When this file, called LNKBND.CSD, is submitted, the system will execute the RUN command, then the LINK86.86 program (the 8086 linker), and finally the EXIT command to leave the "8086 side" and return to the 8085 execution mode.

The percent symbols (%) indicate the use of *formal parameters*. When you execute the SUBMIT command, you supply *actual parameters* for each of these formal parameters. The following is an example:

```
-SUBMIT LNKBND(:F1:PROG1,1)<cr>
```

The SUBMIT command first looks for LNKBND.CSD (it supplies the missing ".CSD" extension), and then it substitutes the first actual parameter (:F1:PROG1) for "%0" and the second actual parameter (1) for "%1". The resulting command sequence is as follows:

```
RUN
LINK86 :F1:PROG1.OBJ, &
       :F1:P86RN0.LIB, &
       :F1:P86RN1.LIB, &
       :F1:P86RN2.LIB, &
       :F1:P86RN3.LIB, &
       :F1:E8087.LIB, &
       :F1:E8087, &
       :F1:LARGE.LIB TO :F1:PROG1.86 BIND

EXIT
```

The SUBMIT program creates another file with a ".CS" extension to hold the resulting command sequence. You should not modify this file.

SUMMARY OF THE SERIES III OPERATING SYSTEM

The Series III system has two execution environments: the 8085 environment (the "8085 side") where the ISIS-II commands and 8080/8085 programs run, and the 8086 environment (the "8086 side"), activated by the RUN program, where 8086 and 8088 programs run. Programs that run in the "8086 side" usually have an ".86" extension in their filenames. The RUN program actually looks for an ".86" extension if you specify a filename without an extension.

You use the Series III operating system and its tools to develop software, but there is another way to use it: your software products can use it as an execution vehicle. The *Intellec Series III Programmer's Reference Manual* describes the "innards" of the Series III operating system and how your programs can make use of sections of the system. By using PL/M or assembly language, you can call operating system procedures directly (as described in the summary of Chapter 5) from your program. Pascal-86 programs automatically use the operating system procedures by using run-time libraries which interface between the Pascal-86 programs and the Series III system.

The Series III system has a standard set of procedures that your software products can use; by using this standard programming interface, you can be sure that your future programs will remain compatible with present and future Intel operating systems.

There are a few more commands that were not introduced because they are used infrequently, or they are easy to use. These commands are adequately introduced in the *Intellec Series III Microcomputer Development System Console Operating Instructions*. You should consult this manual anyway, to be aware of details not mentioned in this tutorial.

CHAPTER 3
TEXT EDITING

"A good workman is known by his tools." —proverb

You use a text editing program to create any kind of document, including a document that consists of program instructions. CREDIT is a text editor that takes advantage of the capabilities you have with CRT screens to:

- Display and scroll text on the screen
- Rewrite text by typing new letters over old ones
- Rearrange lines of text and insert new lines of text between old lines
- Move to any position in the text file or to any point on the screen instantly
- Correct typing mistakes easily as you type

To simplify everyday text editing operations, CREDIT also provides features that allow you to:

- Save both the newly edited version and the old unedited version of the same document
- Find any string of characters and substitute another string of characters automatically
- Copy sections of a document to use in another document
- Create macros to execute several commands at once, or to repeat sets of commands (command iteration)

CREDIT is useful for all documents; for example, we used CREDIT to write this book. CREDIT is more often used to write the Pascal statements, PL/M statements, and assembly language instructions that comprise program modules. Text files created by CREDIT are called documents if they are meant to be read by humans; and they are also called *source programs* if they consist of program statements (Pascal, PL/M, or FORTRAN statements, or assembly language instructions) that are read by compilers or assemblers in order to be compiled or assembled into working programs.

In the following examples, we show you how to create and edit a text file consisting of English sentences that you can translate (eventually) into Pascal-86 statements. As described in Chapter 1, this step of generating Pidgin Pascal (English sentences describing a Pascal program) is a very important one in program development, because the program logic expressed in English sentences closely resembles human thinking and is therefore easier to debug.

CREATING A TEXT FILE AND INSERTING TEXT

When you run CREDIT, you also specify the name of a file to be edited. If CREDIT can find the file you are specifying, it will open the file for editing. If CREDIT cannot find the file you are specifying, it will create a new file by that name.

For example, let's create a file called PIDGIN.TXT in the directory for drive 0. To create and edit PIDGIN.TXT, type the following command:

```
-CREDIT PIDGIN.TXT<cr>
```

CREDIT responds with:

```
ISIS-II CRT-BASED TEXT EDITOR V2.0
NEW FILE    1982 FREE DISK BLOCKS
```

The number of free disk blocks depends on the number of files on your disk and the size of your disk—you do not have to worry about it unless you get a "DISK FULL" warning. If you get such a warning, terminate your CREDIT session by typing "EX" (followed by RETURN), and specify another disk (like :F1:) for your text file.

The screen is partitioned into two areas, as shown in figure 3-1.

Notice the vertical line " | " in the text area? This symbol marks the end of the text in the file. Since the file is new and holds no text yet, this symbol appears at the beginning of the file. As you type text into the file, the symbol moves and continues to mark the end of the text.

The blinking cursor sits underneath this vertical line. You can immediately type text into the file:

```
just type!|
```

As you type, you can make changes to the text you already typed by moving the cursor back to the previous text and typing over it. Move the cursor by using the cursor control keys that surround the HOME key (do not use the HOME key yet!). If you inadvertantly typed the HOME key, hold down the CNTL key and type a V (Control-V) to return the cursor to the text body.

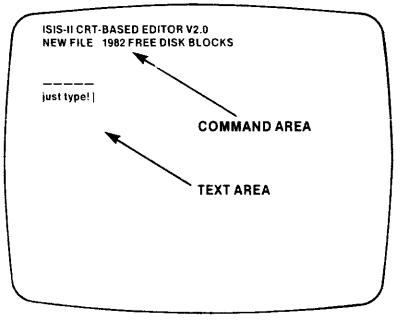


Figure 3-1. The CREDIT™ Video Display 121632-4

You end each line of text with a carriage return by typing the RETURN key, just like a typewriter. CREDIT displays the carriage return operation as an uparrow (↑). The carriage return operation is actually performed by two ASCII codes: one for the carriage return character, and one for the line feed character. CREDIT uses one symbol, the uparrow (↑), for both characters. This symbol is called the *line terminator*.

There are several keys used often in editing:

- The RPT (Repeat) repeats whatever key you hold down. For example, hold down both the RPT key and a cursor movement key, and the cursor will move faster. Use the RPT key with RUBOUT to erase a line.
- The TPWR (Typewriter) key, when in the down position, displays every character in lower case like a typewriter. When in the up position, all characters are in upper case.
- The ESC (Escape) key will cause a *break* in an executing command.

If your keyboard seems to freeze and does not display characters, you probably moved the cursor past the vertical line. You cannot type any characters after the vertical line, since the vertical line marks the end of the text. You can move the cursor to the vertical line and type, thereby moving the vertical line, or you can move the cursor to any position before the vertical line and type over the previously typed characters.

The *ISIS-II CREDIT (CRT-Based Text Editor) User's Guide* contains an interesting tutorial session. Before you learn how to use some of CREDIT's powerful editing commands, you should learn how to end a CREDIT session properly, without losing the edits you have made. Figure 3-1 shows the layout of the screen; in the Command Area you'll find an asterisk. To move the cursor to this asterisk, press the HOME key.

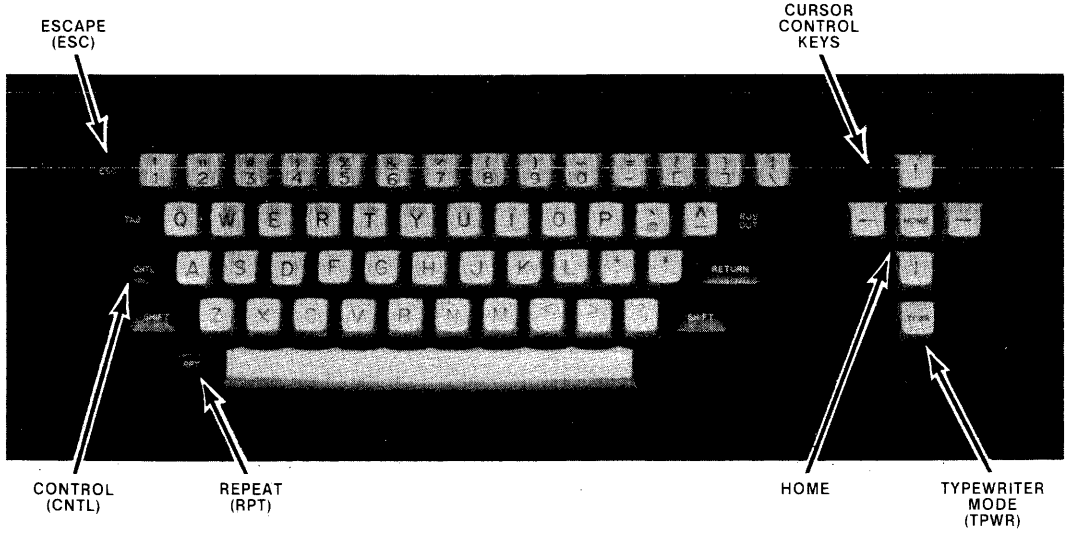


Figure 3-2. The Series III Keyboard 121632-5

When the cursor appears after the asterisk, you can type a CREDIT command. The EX (exit) command terminates a CREDIT session properly. Type "EX", followed by RETURN:

```
*EX<cr>
```

CREDIT will clear the screen and display the following message:

```
EDITED TO :F0:PIDGIN.TXT
```

By using the DIR command, you can see that the directory listing for :F0: now contains a new file called PIDGIN.TXT.

You can use CREDIT to edit PIDGIN.TXT by invoking CREDIT again:

```
-CREDIT PIDGIN.TXT<cr>
```

This time, CREDIT does not have to create the file, because it already exists in the directory for the disk in drive 0. CREDIT responds with:

```
ISIS-II CRT-BASED TEXT EDITOR V2.0
OLD FILE   SIZE=2       1980 FREE DISK BLOCKS
```

The text you typed during the last example appears under this message. The message tells you that PIDGIN.TXT is an old file, its size is 2 blocks (a block is 128 bytes), and you have 1980 free disk blocks in which to expand the file (these numbers vary depending on the space taken up by files on your disk). If you do not have enough free disk blocks to expand the file, CREDIT displays a warning message.

You can continue to experiment with CREDIT by typing and retyping lines of text. When you are finished typing extraneous characters and are ready to type meaningful English sentences, move the cursor to the first character or screen position, and type CNTL and Z (hold down the CNTL key and type Z). The @ symbol will replace the character. Now move the cursor to the end of the text—to the vertical line—and type CNTL and Z together again. ZAP! You just deleted all the text you typed. All of the characters between the two @ symbols were deleted, and you are left with an empty text file again.

Let's start with the simple problem definition. Type the following sentence:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
```

NOTE

The uparrow (↑) stands for the RETURN key. Use the RETURN key to end each line of text, including blank lines.

Using CREDIT, you can type the sentences you know you need, and insert more sentences later. For example, you know you need the following sentences:

```
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

Two questions should immediately come to mind: what information does the program need, and what else would a climate control program do? Let's add new sentences by using CREDIT's insert capability. Here is our text file so far:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

Move the cursor to the beginning of the fourth line of text (the line that begins with "Based on ..."). Hold down the CNTL key and type A (CNTL-A). The rest of the text disappears, and you are now able to type sentences. Type the following sentences and blank lines:

```
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
```

Now type CNTL and A together again. The rest of the text file reappears with the new lines inserted in their proper places. By typing CNTL and A together, you turn on the Add Text Mode; by typing them again, you turn off Add Text mode. Here is your text file with the newly inserted lines:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
```

MOVING AROUND IN THE TEXT FILE

Using the cursor movement keys, you can move the cursor anywhere within a screenful of text. If the text file is larger than twenty lines, it will not fit entirely in one screen—you have to *scroll* the screen by using the scrolling commands.

The scrolling commands only operate when the cursor is in the text area (when the cursor is in the text area, you are in *screen mode*). Refer to figure 3-1 for an illustration of the text and command areas. When the cursor is in the command area (*command mode*), you can only execute command mode commands; to move into screen mode (and move the cursor to the text area), type CNTL-V (hold down the CNTL key and type V).

The scrolling commands are CNTL-N (to see the next screenful of text), and CNTL-P (to see the previous screenful of text). You can also use CNTL-V to see which character CREDIT is currently pointing to, and to move into screen mode. If you experiment with CNTL-V, you will notice that the first use of CNTL-V moves the cursor to the character that CREDIT's *pointer* is pointing to (more on this pointer in the next paragraph). Subsequent executions of CNTL-V rearrange the lines of text so that the line that contains the character pointed to by CREDIT becomes the third line in a screenful of text. This is very useful for partial scrolling. You can use these commands without fear, since they do not modify the text.

NOTE

> The CNTL key, when used with another key (e.g., CNTL-V), is sometimes represented by the uparrow (↑) symbol in our manuals and pocket references. For example, CNTL-V is shown as ↑V, and CNTL-A is shown as ↑A.

The *character pointer* mentioned above is a reference point for all of CREDIT's commands. In screen mode, the cursor represents the pointer. It is sometimes called "the CP". Most text editors have some pointer or marker that points to a place in the file, and commands that insert characters, delete characters, or search for characters use this pointer to find the place in the file to perform their operations. CREDIT's pointer points to a single character, and this pointer moves whenever you move the cursor within the text area. When you type the HOME key and move the cursor to the command area, the pointer stays where it is, pointing to the character in the text area. The CNTL-V command moves the cursor back to the text area and back to the character pointed to.

In screen mode, the cursor movement keys and the scrolling commands move this pointer. In command mode, all editing changes are made relative to the position of this pointer. Deleting a single character, for example, erases the character pointed to by the pointer, and moves the pointer to the next character. When you insert text, the text is inserted preceding this pointer. Many other commands also move this pointer. CNTL-V will always move the cursor to the character pointed to by this pointer.

One command that always moves the pointer is the J (jump) command. You can only use the J command in command mode, and the J command leaves you in command mode (so you have to do another CNTL-V to get into screen mode). You can jump to any location relative to the pointer, or you can jump to specific locations called *tags*. You can set your own tags by using the TS (tag set) command, and delete tags by using the TD (tag delete) command. There are two permanent tags that need not be set, and that cannot be deleted: the beginning of the file, known as TT (tag for top), and the end of the file, known as TE (tag for end). For example:

```
*JTT<cr>
```

This command moves the pointer to the top of the text file (the beginning of the first line).

Since CREDIT usually puts the pointer at the top of the file when you start an editing session, the JTT command (jump to tag for top of file) is more useful during an edit session. The JTE command (jump to tag for end of file) is useful at any time:

```
*JTE<cr>
```

After using the J command, you are still left in command mode. Use the CNTL-V command to return to screen mode, and to return the cursor to the character pointed to by the CP.

Use the JTE command (jump to tag for end of file), followed by a CNTL-V command to return to screen mode, in order to add a sentence to the end of our Pidgin Pascal software definition. The following example shows the execution of the JTE command, followed by a CNTL-V (displayed as ↑V), followed by a display of the current text file with the added sentence at the end:

```
*JTE<cr>
*↑V


Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a demand,↑
and determine the type of demand.↑
If there is no demand, simply continue operating the climate system.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
↑
```

FINDING OLD TEXT AND SUBSTITUTING NEW TEXT

There will be numerous occasions when you will want to find a specific word or group of words, and move the character pointer at the same time. There will also be times when you will want to substitute a new word or group of words for an old one. For example, you might write a program that displays the name of your product (for example, "ACME Solar Controller") in several different places. Sometime later, you find out that the marketing people changed the name to "ACME Climate Controller". With one simple CREDIT command, you could substitute the new word ("Climate") for the old word ("Solar") wherever the old word occurs in the text file. You would only have to make sure that you specified the old name using upper and lower case characters as they appear in the file, and that the new name looks exactly as it should look.

The F (Find) command finds any *string* (group of characters) you specify. The S (Substitute) command finds the old string you specify and substitutes the new string you specify, and rearranges the text so that spaces aren't introduced into the file. The SQ (Substitute after Query) command finds the old string you specify, then asks you for a yes-or-no answer: a yes tells CREDIT to substitute the new string you specified, and a no tells CREDIT not to substitute the new string. If you executed the SQ command iteratively (see next section), CREDIT would continue looking for more instances of the old string.

To show examples of these commands, we'll return to our Pidgin Pascal text file to add some new text and substitute a new word for an old one. Use the F command to find the string "simply continue":

```
*F/simply continue/<cr>
```

When the F command finishes, it displays the asterisk once again. To see where it put the character pointer, type CNTL-V. The cursor should be under the space after the last letter of "continue." We want to rewrite the sentence so that "no method" is one of the methods used to operate the climate system.

At this point, it is easy to move the cursor to the appropriate place to insert new text. You can type over the old text, and use the RETURN key to continue typing a line. You can also use the CNTL-A combination to insert a lot of text. After inserting the new text, the sentences should read as follows:

```
If there is no demand, choose ''no method'' as the method,↑
     and operate the system with this method.↑
If there is a demand for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a demand for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
```

To illustrate use of the S and SQ commands, we will substitute the word "request" for "demand" throughout our text file (and thereby make our program more polite). First, use the JTT command to move the character pointer to the beginning of the file; then use the S command in the following manner:

```
*JTT<cr>
*S/demand/request/<cr>
```

The S command found the first instance of "demand" and substituted "request" for it. Check to see the result by typing CNTL-V:

```
Based on temperature data, see if there is a request,↑
and determine the type of demand.↑
```

Note that S command only found the first instance of "demand" and substituted "request" for it only. Note also that the S command must have moved the comma after the first "demand" in order to fit the word "request" in that place.

To execute the S or SQ commands repeatedly, you would use a form of command iteration. The SQ command performs the same operation as the S command if you answer with a yes; on a no answer, the SQ command does not substitute text. Here is an example of the SQ command, with a sneak preview of the easiest form of command iteration:

```
*!<SQ/demand/request/><cr>
```

The angle brackets around the entire SQ command, in conjunction with the exclamation point, cause the SQ command to be executed repeatedly until the command reaches the end of the text file. Do not be confused by the angle brackets surrounding the "cr"—that is the symbol depicting use of the RETURN key. Angle brackets delimit the command to be executed iteratively, and the exclamation point replaces a number that would specify the number of iterative executions. This is explained in more detail in the next section.

The SQ command displays the line that contains the old text, and then displays a question mark. You must respond with a "Y" for yes, or an "N" for no:

```
and determine the type of demand.↑
?Y
```

You do not have to type RETURN after typing the "Y" or "N", because CREDIT is expecting such an answer. The SQ command goes on to find more instances of "demand":

```
If there is no demand, choose ''no method'' as the method,↑
?Y
If there is a demand for heat, determine the heating method,↑
?Y
If there is a demand for cold, determine the cooling method,↑
?Y

*
```

When the SQ command has found the end of the text file, the condition for iterative execution is satisfied, and the execution ends. Type CNTL-V, followed by a CNTL-P, to see the previous page and the substitutions:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
Get the data needed for each pass: the time, the temperatures,↑
the weather, the state of the solar collector, etc.↑
Store this data.↑
↑
Based on temperature data, see if there is a request,↑
and determine the type of request.↑
If there is no request, choose ''no method'' as the method,↑
     and operate the system with this method.↑
If there is a request for heat, determine the heating method,↑
     and operate the system with this method.↑
If there is a request for cold, determine the cooling method,↑
     and operate the system with this method.↑
↑
If no method is possible (abnormal conditions),↑
shut down the climate system.↑
↑
```

MACROS AND COMMAND ITERATION

Our Pidgin Pascal program still needs a modification that will vastly improve its readability. We need to indent all of the sentences that occur after the sentence "While the system is operating, do (and repeat) the following:". This will improve the readability, since it will then be obvious that the indented sentences are the actions that have to be repeated.

To make this improvement, we will define a macro to hold several editing commands that will be executed iteratively. Rather than explain the process of defining macros and the syntax for command iteration, we'll show you the steps to take to make this improvement to our text file, and then we'll explain them.

To begin, move the cursor (in screen mode, so that it also moves the character pointer) to the beginning of the blank line that follows the sentence "While the system is operating...". Your cursor (and the character pointer) should now be at the beginning of this blank line, as shown by the underscore "_" symbol:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
_
```

Now type the HOME key to return to command mode.

You can only define macros while in command mode. We will now define a macro called X that will make the improvement. Here is our definition of X:

```
*MSX@L0;%<L;I/□□□□□/;P>@<cr>
```

Type this macro definition exactly as it is shown above (the "□" symbol stands for a space—type five spaces—and the "<cr>" stands for the RETURN key). When CREDIT finishes digesting this definition, it displays the asterisk again. Now type the following command:

```
*MFX(13)<cr>
```

You will see each line following the sentence "While the system is operating..." being acted upon by macro X. All of the lines following that sentence are now indented by five spaces, as shown below:

```
Maintain the climate of a building using a system comprised of↑
heating and cooling methods.↑
↑
Startup the climate system.↑
↑
While the system is operating, do (and repeat) the following:↑
↑
     Get the data needed for each pass: the time, the temperatures,↑
     the weather, the state of the solar collector, etc.↑
     Store this data.↑
     ↑
     Based on temperature data, see if there is a request,↑
     and determine the type of request.↑
     If there is no request, choose ''no method'' as the method,↑
          and operate the system with this method.↑
```

```
        If there is a request for heat, determine the heating method,↑
             and operate the system with this method.↑
        If there is a request for cold, determine the cooling method,↑
             and operate the system with this method.↑
        ↑
    If no method is possible (abnormal conditions),↑
    shut down the climate system.↑
    ↑
```

"What was that gibberish I typed?" To answer your question, display the macro definition again by typing the following command in command mode:

```
*?M<cr>
XL0;%<L;I/      /;P>
```

The display is exactly what you typed before, without the "@" symbols. The "@" symbols were used to define the macro—they delimit the actual text of the macro, so they are not necessary in this display (therefore CREDIT does not display them).

The macro starts with its name: X. Following the name is the first command in the macro: the L0 command, which moves the character pointer (called CP from now on) to the beginning of the current line (the line that the CP is currently sitting on—this line changes every time the entire macro executes). Following the L0 command is a semicolon, which separates one command from the next.

The next command in macro X is a set of commands to be executed repeatedly, or *iteratively*. The "%" (percent) symbol is a special one in this case, and is explained in the next paragraph. You saw the angle brackets in the last section—they delimit the set of commands to be executed repeatedly. The commands to be executed repeatedly are the L command (move CP to the beginning of the next line), the I command (insert the text that is specified within the following "/" (slash) symbols), and the P command (display the current line). The text within the "/" (slash) symbols consists of the five spaces we need to insert in order to indent the lines.

Now we'll explain the "%" (percent) symbol. When you typed the command to execute macro X, you typed:

```
*MFX(13)<cr>
```

The "MF" command executes the macro "X". The number 13 in parentheses is called a *parameter*—the MF command substitutes the parameter you specify (in this case, 13) for the "%" (percent) symbol. The result is that the iterative command set ("<L;I/ /;P>") is executed 13 times. Why did we pick 13? Because there were only 13 lines ahead of the character pointer to be indented. We could have specified a "!" (exclamation point) symbol to execute the iterative command set over and over until it reached the end of the file, but we chose this method to show you the use of parameters in macros, and to make sure that the last two lines would not be indented.

We also chose to use the "@" symbol to delimit the entire macro, but you can use any symbol that is not used within the macro.

Obviously, there are more details you should learn before using macros and other advanced editing features. However, you do not need these features to perform simple editing operations. They exist only to make your text editing sessions easier, if you first learn how to use them.

ENDING A TEXT EDITING SESSION AND MANAGING BACKUP FILES

When you are editing text, the edited text is in the computer's memory, but it is not yet on disk. To update the text file with the edited text, you have to end your text editing session properly. The EX command properly replaces the old text in the file with the newly edited text, and it also saves the old version of the file in *another* file called a *backup* file. When the edit session ends, control of the system returns to the operating system (ISIS-II).

If you have been following the examples in this chapter, you are now in CREDIT with your edited text, and you are ready to end your edit session. If you are not in command mode (with the asterisk prompt), use the HOME key to get into command mode. When you are in command mode, type the following command:

```
*EX<cr>
```

The text on the screen should disappear, and another message should appear:

```
EDITED TO :F0:PIDGIN.TXT
```

The file PIDGIN.TXT, on the disk in drive 0, now contains the newly edited text. Use the DIR command to see the directory for drive 0. In the directory listing, you should see the filename PIDGIN.BAK. CREDIT created this file to be the backup file that contains the old unedited version of PIDGIN.TXT. Every time you use the EX command by itself, CREDIT automatically creates a backup file with the ".BAK" extension to contain the previous version of the file.

NOTE

You should not execute CREDIT to edit a backup file (a file with the ".BAK" extension), because the EX command would first put the new edits into the ".BAK" file, then it would overwrite the ".BAK" file with the previous (unedited) version of it. Use the REN (rename) command to rename the file before editing it, or use the filename option with the EX command, described in the next paragraph.

You can optionally specify a filename with the EX command, so that the newly edited text becomes a new text file, and the old text file remains unedited. For example, if you had typed the following version of the EX command (rather than the preceding version):

```
*EX NEW.TXT<cr>
```

the following message would appear:

```
EDITED TO :F0:NEW.TXT
```

The file NEW.TXT would contain the newly edited text, and the old PIDGIN.TXT would not have been updated (and PIDGIN.BAK would not have been created as a backup file).

NOTE

If you type "EXIT" rather than "EX", CREDIT assumes that you want to store the newly edited text in the file IT (:F0:IT).

Another way to end an edit session is to use the EQ (quit) command. The EQ command will keep all files unchanged, as if nothing had happened. All edits you made while in the edit session vanish, and the text and backup files (if they exist) remain unchanged. If you use the EQ command in a session that created the text file, the new text file would not exist.

To keep you from making a mistake, the EQ command will first ask you for a Y or N answer before it ends the session:

```
*EQ<cr>
QUIT?Y
```

If you reply with anything other than a "Y" or "y", the EQ does not end the edit session.

DISPLAYING AND PRINTING TEXT FILES

Now that you have a text file, we can show you how to display the file on your screen and copy the file to a printing device. You can display the text file on your screen without using CREDIT by using the COPY command and the device name :CO: (for "console output"):

```
-COPY PIDGIN.TXT TO :CO:<cr>
```

To print the text file PIDGIN.TXT, you use the COPY command to copy the file to the device name :LP: (for "line printer"), or to the device name :TO: (for "teletype output"). You can COPY a file to any output device. You can find a complete list of device names in the *Intellec Series III Microcomputer Development System Console Operating Instructions*. This example assumes that you have an :LP: device to receive a copy of the file :F0:PIDGIN.TXT:

```
-COPY PIDGIN.TXT TO :LP:<cr>
```

FROM TEXT TO PROGRAM

Your PIDGIN.TXT text file now contains an algorithm that is actually a program in disguise (the disguise is English, or Pidgin Pascal). You should keep a copy of it somewhere, perhaps on another disk (or leave it in drive 0 if you have a hard disk subsystem). To use this algorithm in the next chapter, you will want to copy the text file to the Pascal-86 disk, and call it MAIN.SRC, since it will become the source file of your main control algorithm:

```
-COPY PIDGIN.TXT TO :F1:MAIN.SRC<cr>
:F0:PIDGIN.TXT COPIED TO :F1:MAIN.SRC
```

The examples in this and subsequent chapters assume that you either have a hard disk subsystem, or at least two double-density flexible disk drives. If you have single-density flexible disk drives, you should have more than two of them; in this case, you should put your program on the third disk, since it probably will not fit on the Pascal-86 disk in drive 1.

Although MAIN.SRC only has Pidgin Pascal statements at this time, you will edit them to make them real Pascal-86 statements as they appear in figures in Chapter 4.

CHAPTER 4
PROGRAMMING IN PASCAL-86

"One of the most important aspects of any computing tool is its influence on the thinking habits of those who try to use it..."

—E. W. Dijkstra

The Series III Microcomputer Development System was designed to support a variety of programming techniques with several programming languages. The preceding chapters give you the background you need to use this system wisely, and this and the following chapters help you decide the criteria for decomposing your application into modules and picking the appropriate language to use for each module.

One popular approach to programming is the top-down approach, where you define the problem completely, design an abstract algorithm to solve the problem, and refine this algorithm into self-supporting modules that can be coded and compiled separately. Typically, the *main module* would contain the most abstract algorithm—the control algorithm at the top of the design that solves the entire problem. The subordinate modules perform the procedures dictated by the main module.

Pascal-86 is a language that is ideal for the main module of such a modular solution. Using Intel's Pascal-86 compiler, you can decompose a program into modules that can be compiled separately, whereas other Pascal compilers only compile whole programs which have to be tailored to fit into microprocessor environments.

Perhaps the most important reason for Pascal's wide acceptance is the fact that it is a language that closely resembles English. In the past, programmers had to keep their algorithm designs well within the constraints of programming languages that were designed to express mathematical equations. In other words, at the outset they had to think in terms of the programming language available to them. This approach reinforced the practice of giving implementation ("how to") information in the problem definition (for example, "the input to this program is to be formatted on cards in columns 0-15 ...").

In Chapter 3 we designed an algorithm for the climate control of a building using English, which we jokingly call Pidgin Pascal. Since our control structures are in English, we have been able to communicate this algorithm easily and test its logic before translating it into Pascal-86. By now we should have a complete problem definition and a clean algorithm that could be translated into *any* programming language.

TRANSLATING PIDGIN PASCAL TO PASCAL-86

Figure 4-1 shows the main climate control algorithm, described in Pidgin Pascal. Several assumptions are made: that another subordinate module will operate the climate system, that yet another module will access and store the data, and that the data itself will be in the form of a record, which will be available to this algorithm.

As in typical development situations, a change has just occurred in our climate system that we software engineers have to accomodate: the first version of our climate system will not have cooling methods—only heating methods. We must design the software to make room for cooling methods in the future.

We made another change to the algorithm to accomodate a "panic" condition. An algorithm is not complete unless it can handle *any* situation; remember, Murphy said that if it can go wrong, it will. Therefore, we added a test to see if the climate system can handle the request for heat. If neither the collector water nor the tank water is hot enough to heat the building, a panic condition occurs that stops the normal operation of the climate system. At this time, it is sufficient to simply stop the program and output warning messages; later, we can add more procedures to handle such panics.

Using CREDIT (as described in Chapter 3), you can change this algorithm into Pascal-86 by adding the Pascal-86 statements and using *comment symbols* to turn the English sentences into program comments. The (* and *) symbols tell the Pascal-86 compiler to ignore whatever is between them. Some comments are only a few words surrounded by the (* and *) symbols, but comments can take up many lines, as shown at the end of the program in figure 4-2. As soon as the compiler sees the (* symbols, it ignores the characters and lines following it until it sees the *) closing symbols.

These comments are carried over with the program statements to the listing file produced by the compiler. You use the listing file as documentation for the program. You'll see a listing file later in this chapter.

Figure 4-2 shows the same algorithm expressed in Pascal-86 statements, with the English sentences masquerading as program comments. We also added more comments. It is a good practice to write the comments of a program before writing the actual program statements.

```
Maintain the climate of a building using a system comprised of
heating methods.

Startup the climate system.

While the system is operating, do (and repeat) the following:

     Get the data needed for each pass: the time, the temperatures,
     the weather, the state of the solar collector, etc.
     Store this data.

     Based on temperature data, see if there is a request
     for heat.
     If there is no request, choose ''no method'' as the method,
          and operate the system with this method.
     If there is a request for heat, determine whether the
          system can handle the request. If not, cause a panic.
          Otherwise, determine the heating method,
          and operate the system with this method.

If no method is possible (panic or abnormal conditions),
shut down the climate system.
```

Figure 4-1. Algorithm for the Climate Control Main Module

```
(**Type and variable declarations to be supplied later**)
(**Public procedures external to this program will be supplied later**)

PROGRAM MainControl(INPUT,OUTPUT);

BEGIN (**Main Control Algorithm**)
StartUpSystem; (*procedure to start up the climate system*)
Operating:=TRUE;
Panic:=FALSE;
   WHILE Operating DO (*While the system is operating, do
                       (and repeat) the following:*)
      BEGIN
         GetData(CurrentData);
             (*Get data needed for each pass: temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)

  (**If there is a request for heat, determine whether the system
         can handle the request.  If not, cause a panic.
         Otherwise, determine the heating method,
         and operate the system with this method.
      If there is no request for heat, choose ''no method,''
         and operate the system with this method.       **)

         WITH CurrentData DO
             BEGIN
                IF InsideTemp<ThermostatSetting THEN (*if request*)
                 BEGIN
                   IF CollectorWaterTemp>MinimumForExchanger THEN
                      BEGIN ChosenMethod:=CollectorToExchanger;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF CollectorWaterTemp>MimimumForHeatPump THEN
                      BEGIN ChosenMethod:=CollectorToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF TankWaterTemp>MinimumForExchanger THEN
                      BEGIN ChosenMethod:=TankToExchanger;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF TankWaterTemp>MimimumForHeatPump THEN
                      BEGIN ChosenMethod:=TankToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                      BEGIN ChosenMethod:=HeatedTankToHeatPump;
                            OperateSystem(CurrentData);
                      END
                   ELSE Panic:=TRUE; Operating:=FALSE;
                 END (*if heating request*)
                ELSE (*no heating request*)
                   BEGIN ChosenMethod:=NoMethod;
                         OperateSystem(CurrentData);
                   END;
             END; (*End routine WITH CurrentData*)
      END; (*While operating*)
ShutDownSystem(CurrentData); (* panic or abnormal condition *)
END.
```

Figure 4-2. First Try at Coding the Main Program

```
(**The following are only comments:

   The following procedures will be coded in another module
   called the Operation Module:

   PROCEDURE OperateSystem(CurrentData)

   This procedure will operate the system and constantly
   maintain heat gain in the system. Depending on the
   heating method chosen, it opens certain valves and
   closes others, and turns on certain pumps and turns
   off others. It also maintains a flow of heated water
   to the storage tank.

   NOTE: for our testing purposes, a dummy OperateSystem
   procedure will only display messages telling us what
   heating method was chosen, and temperature data.


   PROCEDURE ShutDownSystem(CurrentData)

   This procedure will perform an orderly shut down if a
   panic or abnormal condition occurs.  The shut down must
   keep warm water flowing through the solar collector and
   close any extraneous valves, etc.  It must also send a
   warning messages to the console, advising manual operation
   of the furnace, etc.

   NOTE: for our testing purposes, a dummy ShutDownSystem
   procedure will only display data and a shutdown message.


   PROCEDURE StartUpSystem

   This procedure will start the climate system (cold start,
   or after ShutDownSystem occurs), open necessary valves,
   etc. It will also display a startup message.

   NOTE: for our testing purposes, a dummy StartUpSystem
   procedure will only display a startup message.


   The following procedures will be coded in another module
   called the GetData Module:

   PROCEDURE GetData(CurrentData)

   This procedure will obtain the data from a PL/M-86 module
   called PLMDATA that accesses ports to obtain temperature
   data.  Data other than temperature data will come from
   ports via port input/output procedures in this Pascal-86
   module.

   NOTE: for our testing purposes, a dummy GetData procedure
   will obtain all data from the console.
```

Figure 4-2. First Try at Coding the Main Program (Cont'd.)

```
PROCEDURE StoreData(CurrentData)

This procedure will store the data record CurrentData in a
file (the file would probably reside in non-volatile bubble
memory).

NOTE: for our testing purposes, a dummy StoreData procedure
will simply output the data to the console.


This is the last line of comments.**)
```

Figure 4-2. First Try at Coding the Main Program (Cont'd.)

This guide cannot possibly explain Pascal syntax—there are several books mentioned in the Bibliography that can give you the background you need, and the *Pascal-86 User's Guide* provides all the information you need to use Intel's extensions to standard Pascal.

PASCAL-86 DATA TYPES

A major advantage that Pascal has over other high-level languages is its strong type checking mechanisms that enforce data typing. By using Pascal's data types, you avoid some of the classic causes of errors in programs—the ambiguities involved with using simple X and Y variables to hold truly non-numeric data, the mistakes that occur when you attach arbitrary meanings to numeric data, and the complexities that are magnified by ambiguous variable names.

An example is worth a thousand explanations. In the Pascal-86 algorithm in figure 4-2, we make assignments like this one:

```
ChosenMethod:=TankToExchanger;
```

If **ChosenMethod** and **TankToExchanger** are declared properly in the module heading (not shown in figure 4-2, but shown later in this section), the Pascal-86 compiler will know their meanings. When you read this assignment, you know exactly what heating method has been chosen. The data type is a type of heating method, not an integer representing a method. In other programming languages you might be able to have a variable named "ChosenMethod" and another variable named "TankToExchanger", but you would also have to be sure to assign proper numeric or string values to them. A typical way of expressing the above assignment in PL/M would be:

```
      CHOSEN$METHOD = 2
/*where 2 is the appropriate method*/
```

or

```
      CHOSEN$METHOD = T$TO$EXCH
/*where T$TO$EXCH has already been assigned the appropriate
value*/
```

In both cases, you have to know a numeric code for the heating method. In Pascal, however, you only have to define a set of heating methods, and pick one for the assignment. Here is an example of such a definition:

```
TYPE HeatingMethods = (CollectorToExchanger,
                       CollectorToHeatPump,
                       TankToExchanger,
                       TankToHeatPump,
                       HeatedTankToHeatPump,
                       NoMethod);

(*The above defines the data type HeatingMethods, which is
  used to define the variable ChosenMethod below.*)

VAR ChosenMethod  : HeatingMethods;
```

In order to assign a value to **ChosenMethod**, the program must assign one of the methods in the set of type **HeatingMethods**. Any other assignment would cause a compiler error message to occur. By enforcing this data typing mechanism, the Pascal-86 compiler reduces the number of run-time errors by flagging the data type errors early in the game.

There are several standard Pascal data types that are useful. For example, you can define a variable to be of type BOOLEAN, which means that the only values that can be assigned to the variable are the values TRUE and FALSE. The variables **Operating** and **Panic** are of type BOOLEAN; they are either TRUE or FALSE.

Note that **ChosenMethod** and **CHOSENMETHOD** would refer to the same variable, since lower case characters are treated as upper case. This feature allows you to create long identifying names with combinations of upper and lower case characters that are easy to read and understand.

ANOTHER LOOK AT MODULARIZING AND HIDING INFORMATION

The programming technique called information-hiding is *not* an excuse for designers to withhold information from their documentors—it is more akin to a technique we use to hold a lot of information in our minds. When we have to interface with several different organizations within a company in order to get a job done, we don't pay attention to the inner workings of an organization; we simply assume that the organization will do its job, and we define our interface with the organization. Their organization is one module, and ours is another module; the job gets done because the modules know how to communicate to each other without interfering in each other's details.

Most logical algorithms are designed with assumptions about working modules. In our algorithm in figure 4-2, we assume that the procedures **StartUpSystem**, **GetData**, **StoreData**, **OperateSystem** and **ShutDownSystem** will work as planned, even though they are not yet written. We also assume that another group may write them. We can make these assumptions because we designed our main module to hide most of the details about choosing heating methods.

So far, the main module's algorithm decides the appropriate heating method based on a set of data. Once the algorithm is written, it may never change; and if it had to change, its change should not affect the other modules. However, we could change the control algorithm so that changes to the heating methods, or additional heating methods, would not even affect the main control algorithm. A simple way to do this would be to turn the heating method determina-

tion algorithm into an independent procedure called **DetermineMethod**, extract from this algorithm the calls to **OperateSystem**, and put the call to **OperateSystem** in the control algorithm.

The resulting main module is shown in figure 4-3. We added the module heading, but we still need the interface specification and variable declarations (shown later).

```
MODULE MainControl;

(* Interface specification goes here, to be supplied later. *)
(* Type definitions and variable declarations to be supplied later. *)

PROGRAM MainControl(INPUT,OUTPUT);


PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
   BEGIN
     WITH CurrentData DO
          BEGIN
            IF InsideTemp<ThermostatSetting THEN
               BEGIN
                  IF CollectorWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=CollectorToExchanger
                  ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=CollectorToHeatPump
                  ELSE IF TankWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=TankToExchanger
                  ELSE IF TankWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=TankToHeatPump
                  ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                     ChosenMethod:=HeatedTankToHeatPump
                  ELSE Panic:=TRUE; Operating:=FALSE;
               END
            ELSE (*no heating request*)ChosenMethod:=NoMethod
          END; (*With CurrentData*)
   END; (*DetermineMethod)

(************** MAIN PROGRAM ************************************)

BEGIN
   StartUpSystem;
   Operating:=TRUE;
   Panic:=FALSE;
   WHILE Operating DO (*while system is operating, do:*)
      BEGIN
         GetData(CurrentData); (*Get the temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)
         DetermineMethod(CurrentData); (*this detects a panic*)
         OperateSystem(CurrentData);
      END; (*while operating*)
   ShutDownSystem(CurrentData);
END. (*Main Control Algorithm*)
```

Figure 4-3. Second Try at Coding the Main Program

The **DetermineMethod** procedure now hides all the information about choosing the appropriate heating method. We could also rewrite it to include cooling methods, or to change the heating methods. The procedure expects to receive the record **CurrentData**, and it only changes the value of the variable **ChosenMethod**.

The **OperateSystem** procedure will not be written until more facts are known about the hardware of the actual climate system. However, we already know that if we make a decision about a chosen method, include that method in the data record **CurrentData**, and send that data record to the **OperateSystem** procedure properly, the **OperateSystem** procedure will know how to operate the system. We defer these details to a later time when we'll have a prototype system to operate.

PASSING DATA TO OTHER MODULES—PARAMETER PASSING TECHNIQUES

We designed our main module so that it will receive a record of information. A Pascal-86 *record* is much like a PL/M-86 *structure* which can be defined to hold certain data types. We have to define this record in order to write the main module, but we can defer decisions about obtaining the data in order to preserve our options.

A PL/M-86 procedure could easily obtain the data and build the structure according to interface specifications; so could an 8086/8087/8088 Macro Assembly Language program, or an 8089 Assembly Language program. In fact, we might be able to use existing routines to obtain the data, and simply write another routine to structure the data accordingly.

In any case, we only have to pass the *address* of the structure to the Pascal-86 main module, which knows what to do with it. This parameter-passing technique is known as *pass by reference*, because the main module only needs a reference to the address of the structure in order to treat the structure as a Pascal record.

Another parameter-passing technique is *pass by value*, where a procedure calls another procedure and sends it a value rather than an address. We don't use this technique in our application, since our procedures need to access data in the record. We decided against passing specific values from this data record, and decided instead to make the entire data record available to the appropriate procedures.

To define the data record properly and still provide the ability to change it easily, we created a data type for the record:

```
TYPE (*definitions publicly defined in this module*)
.
.
.
     SystemData        = RECORD
                           ChosenMethod          : HeatingMethods;
                           InsideTemp,
                           ThermostatSetting     : AirTemperature;
                           CollectorWaterTemp,
                           TankWaterTemp,
                           HeatedTankTemp        : WaterTemperature;
                           AmountOfSunlight      : Integer;
                           Hour                  : 00..24;
                           Minute                : 00..59;
                         END (*SystemData*);
```

Using the data type **SystemData**, we defined the variable **CurrentData** to be of that type:

```
VAR
   CurrentData                : SystemData;
.
.
.
```

We pass the variable **CurrentData** to other procedures. If we were to change the data fields in the record, we would only have to change the definition of **SystemData**; we would not have to change any of the calls that pass the variable **CurrentData**. If the data fields kept the same names (**ChosenMethod**, **ThermostatSetting**, etc.), we would not have to change the routines that use those data fields.

THE INTERFACE SPECIFICATION

A module that calls a procedure in another module must have some information about where the other procedure is, and it must provide some information to the other procedure about the data being passed. Intel's Pascal-86 provides a mechanism for supplying the appropriate information to all modules that are to be linked to form a program—it is called an *interface specification*.

The interface specification typically holds the type definitions and variable declarations that are needed by all modules, and it also contains the names of procedures (with their parameters) that are public to other modules; that is, they can be called from other modules. Each module of the entire program contains this information. Figure 4-4 shows the interface specification for our program.

In addition to PUBLIC definitions in the interface specification, a module can have PRIVATE type definitions and variable declarations for variables used only within the module. Our **Operation** module will have a PRIVATE section for all variables that are only used within the module, but our **MainControl** main module does not need one. In Pascal-86, a PRIVATE heading is used in non-main modules instead of a PROGRAM heading.

Several *enumerated types* are defined in our program: **AirTemperature** and **WaterTemperature** are defined as types that can only have values in the ranges specified. The variables **Hour** and **Minute** are also of enumerated types, but since their ranges do not change, their types are not defined separately. By defining the temperature types separately, we can easily change their ranges without affecting the data record.

By defining the data record as type **SystemData**, we can easily change data fields in the record without changing the **CurrentData** declaration. By defining and declaring types and variables in the interface specification, we can maintain the interface specification separately (and change definitions and declarations) without affecting the procedures.

TEST VERSION OF THE CLIMATE CONTROL SYSTEM

Since our hardware designs are not yet firm, we should put together a test version of our system that does not interact with any prototype hardware. This version should include dummy procedures for the procedures that would normally rely on 8088 ports and other hardware.

```
PUBLIC MainControl; (*section of interface specification*)

   CONST (*declarations declared publicly in this module*)

     MinimumForExchanger   =  35;(*degrees Celsius*)
     MinimumForHeatPump    =  13;

   TYPE (*definitions publicly defined in this module*)

     AirTemperature   =-20..120;(*degrees in Celsius*)
     WaterTemperature =0..120;
     HeatingMethods   =(CollectorToExchanger,
                        CollectorToHeatPump,
                        TankToExchanger,
                        TankToHeatPump,
                        HeatedTankToHeatPump,
                        NoMethod);
     SystemData       = RECORD
                          ChosenMethod          : HeatingMethods;
                          InsideTemp,
                          ThermostatSetting     : AirTemperature;
                          CollectorWaterTemp,
                          TankWaterTemp,
                          HeatedTankTemp        : WaterTemperature;
                          AmountOfSunlight      : Integer;
                          Hour                  : 00..24;
                          Minute                : 00..59;
                        END (*SystemData*);

   VAR (*variables publicly defined in this module.*)

     CurrentData                : SystemData;
     Operating, Panic           : BOOLEAN;

PUBLIC GetData; (*GetData Module containing GetData & StoreData*)

   PROCEDURE GetData(VAR CurrentData:SystemData);
   PROCEDURE StoreData(VAR CurrentData:SystemData);

PUBLIC Operation; (*Operation Module containing OperateSystem,
                    StartUpSystem and ShutDownSystem*)
   PROCEDURE StartUpSystem;
   PROCEDURE OperateSystem(VAR CurrentData:SystemData);
   PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
```

Figure 4-4. The Interface Specification

The dummy versions are shown in figure 4-5. You should type these versions exactly as you see them, with the **StartUpSystem**, **OperateSystem**, and **ShutDownSystem** dummy procedures in the dummy **Operation** module stored in the file :F1:DUMOP.SRC, and **GetData** and **StoreData** dummy procedures in the dummy **GetData** module stored in the file :F1:DUMDAT.SRC. The **MainControl** module should be stored in :F1:MAIN.SRC, and the *interface specification* shown in figure 4-4 should be typed into the file :F1:INSPEC.SRC. If you cannot fit all of these files on the disk in drive 1, you should put all of them on another disk—and use your own pathname (:F*n*:) for the ".SRC" files. For our examples we assume that these files are on the disk in drive 1, along with the Pascal-86 compiler and run-time libraries.

Figure 4-5 shows each module and the dummy procedures. Since the interface specification is repeated in each module, we use a shortcut when compiling the modules by putting the common interface specification in a seperate file (:F1:INSPEC.SRC), and we use the INCLUDE control in a *control line* as shown:

```
$INCLUDE(:F1:INSPEC.SRC)
```

If you put INSPEC.SRC on a disk in a drive other than drive 1, use your own pathname instead of :F1:INSPEC.SRC.

```
MODULE MainControl;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PROGRAM MainControl(INPUT,OUTPUT);

(* end of interface specification *)

PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
   BEGIN
     WITH CurrentData DO
          BEGIN
            IF InsideTemp<ThermostatSetting THEN
               BEGIN
                  IF CollectorWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=CollectorToExchanger
                  ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=CollectorToHeatPump
                  ELSE IF TankWaterTemp>MinimumForExchanger THEN
                     ChosenMethod:=TankToExchanger
                  ELSE IF TankWaterTemp>MinimumForHeatPump THEN
                     ChosenMethod:=TankToHeatPump
                  ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
                     ChosenMethod:=HeatedTankToHeatPump
                  ELSE Panic:=TRUE; Operating:=FALSE;
               END
            ELSE (*no heating request*)ChosenMethod:=NoMethod
          END; (*With CurrentData*)
   END; (*DetermineMethod)
```

Figure 4-5. Test Version of Our Climate Control System

```
(************** MAIN PROGRAM ************************************)

BEGIN
   StartUpSystem;
   Operating:=TRUE;
   Panic:=FALSE;
   WHILE Operating DO (*while system is operating, do:*)
      BEGIN
         GetData(CurrentData); (*Get the temps, time, etc.*)
         StoreData(CurrentData); (*Store this data as record*)
         DetermineMethod(CurrentData); (*this detects a panic*)
         OperateSystem(CurrentData);
      END; (*while operating*)
   ShutDownSystem(CurrentData);
END. (*Main Control Algorithm*)


(*This is a dummy GetData module, with dummy GetData
and StoreData procedures, for use with MainControl
module in testing phases. It only performs console
input to get Celsius temperatures, the time of day,
and the amount of sunlight (insolation) for the
solar collector. Use PLMDATA module for real
application.*)

MODULE GetData;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PRIVATE GetData;
(* end of interface specification *)


PROCEDURE GetData(VAR CurrentData:SystemData);
BEGIN
   WITH CurrentData DO BEGIN
      WRITE('Type the thermostat setting in degrees Celsius:');
      READLN(ThermostatSetting); WRITELN;
      WRITE('Type the inside temperature reading in Celsius:');
      READLN(InsideTemp); WRITELN;
      WRITE('Type the temperature of the collector water in Celsius:');
      READLN(CollectorWaterTemp); WRITELN;
      WRITE('Type the temperature of the tank water in Celsius:');
      READLN(TankWaterTemp); WRITELN;
      WRITE('Type the temperature of the heated tank water in Celsius:');
      READLN(HeatedTankTemp); WRITELN;
      WRITE('Type the hour of day, as in 04 or 24:');
      READLN(Hour); WRITELN;
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

```
      WRITE('Type the minute of the hour, as in 01 or 59: ');
      READLN(Minute); WRITELN;
      WRITE('Type the amount of sunlight, any integer will do for now:');
      READLN(AmountOfSunlight); WRITELN;
   END;  (*with CurrentData*)
END;
PROCEDURE StoreData(VAR CurrentData:SystemData);
BEGIN
(*Dummy procedure, eventually will store CurrentData in a file*)
   WITH CurrentData DO BEGIN
      WRITELN('---------------------------');
      WRITELN('CURRENT DATA IS AS FOLLOWS:');
      WRITELN('---------------------------');
      WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
      WRITELN('Inside temperature is ',InsideTemp,'C');
      WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
      WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
      WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
      WRITELN('Time of day is ',Hour,':',Minute);
      WRITELN('Amount of sunlight is ',AmountOfSunlight);
      WRITELN; (*a blank line*)
   END; (*with CurrentData*)
END.

(*This is a dummy Operation module, with dummy StartUpSystem,
ShutDownSystem, and OperateSystem procedures,
for use with MainControl module in testing phases.*)

MODULE Operation;

(* Interface specification common to all modules *)

$INCLUDE(:F1:INSPEC.SRC)

PRIVATE Operation;
(* end of interface specification *)


PROCEDURE StartUpSystem;
BEGIN
     WRITELN ('Climate system is now on.');
     WRITELN ('-------------------------');
     WRITELN;
END;


PROCEDURE OperateSystem(VAR CurrentData:SystemData);
BEGIN
  WITH CurrentData DO BEGIN
    WRITELN('=============================================');
    WRITELN('The Climate System is now operating.');
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

```
      WRITELN;
      WRITELN('The Time is ',Hour,':',Minute);
      WRITELN('The inside temperature is ',InsideTemp,'C');
      WRITELN('The thermostat setting is ',ThermostatSetting,'C');
      WRITE('Method chosen to heat the building: ');
      CASE ChosenMethod OF
         CollectorToExchanger: WRITE('Solar Collector to Exchanger');
         CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
         TankToExchanger     : WRITE('Tank to Exchanger');
         TankToHeatPump      : WRITE('Tank to Heat Pump');
         HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
         NoMethod            : WRITE('No heat required');
      END;
      WRITELN;
      WRITELN('=============================================');
      WRITELN; (*write a blank line.*)
    END;
  END;(*OperateSystem*)


  PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  BEGIN
  WRITELN(':::::::::::::::::::::::::::::::::::::::::::::::::' );
     IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
              ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
     WRITELN(':::::::::::::::::::::::::::::::::::::::::::::::::') ;
     WITH CurrentData DO BEGIN
        WRITE('Last chosen heating method was: ');
        CASE ChosenMethod OF
           CollectorToExchanger: WRITE('Solar Collector to Exchanger');
           CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
           TankToExchanger     : WRITE('Tank to Exchanger');
           TankToHeatPump      : WRITE('Tank to Heat Pump');
           HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
           NoMethod            : WRITE('No heat required');
        END;
        WRITELN;
        WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
        WRITELN('Inside temperature is ',InsideTemp,'C');
        WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
        WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
        WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
        WRITELN('Time of day is ',Hour,':',Minute);
        WRITELN('Amount of sunlight is ',AmountOfSunlight);
     END;(*with CurrentData*)
     WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::: :::');
     WRITELN;
     WRITELN('Goodnight, Irene...');
  END.(*ShutDownSystem*)
```

Figure 4-5. Test Version of Our Climate Control System (Cont'd.)

THE PASCAL-86 COMPILER

A *compiler* is a program that translates your high-level language statements into machine code. Machine code, sometimes called *object code*, consists of the instructions that machines understand, whereas high-level language statements are instructions that humans understand. You must translate your high-level language statements into machine code by *compiling* your high-level language program.

The Pascal-86 statements we typed (using CREDIT) are *program source statements*. We now have three *source files*: :F1:MAIN.SRC (for the **MainControl** module), :F1:DUMDAT.SRC (for the dummy **GetData** module), and :F1:DUMOP.SRC (for the dummy **Operation** module). To translate these source statement modules into object code modules, we must compile each source module separately.

The Pascal-86 compiler is supplied as the file PASC86.86. You invoke the compiler by using the RUN command to load and execute it in the "8086 side" (8086 execution mode) of the Series III system. The compiler usually produces two files: a *listing file* that contains a listing of the source program as the compiler saw it, and an *object file* that contains the actual machine code. The listing file usually contains a listing of the source statements, additional information about the compilation, and any errors that occurred during the compilation.

For example, assume that MAIN.SRC, the first module, is in directory :F1:. The Pascal-86 Compiler, PASC86.86, is also in directory :F1:. To compile this module, use the following command:

```
-RUN :F1:PASC86 :F1:MAIN.SRC DEBUG<cr>
```

Let's analyze this command line. RUN is the command used to execute the program in the "8086 side" of the system. :F1:PASC86 is the pathname (without the ".86" extension) of the Pascal-86 Compiler (RUN supplies the ".86" extension). :F1:MAIN.SRC is the pathname for the MAIN.SRC module. Finally, DEBUG is a *compiler control* which tells the compiler to do something special (described later).

This compilation produced two files: :F1:MAIN.LST is the listing file, and :F1:MAIN.OBJ is the object file that contains the object code. :F1:MAIN.SRC is still in drive 1. All of these files are in directory :F1:, since that is where MAIN.SRC resides.

Compiler controls tell the compiler to perform certain operations. Most controls have *default settings* that you do not have to specify. For example, the PRINT control is always on unless you specify NOPRINT. The PRINT control tells the compiler to produce a listing file, and use the name of the source file with an ".LST" extension (e.g., :F1:MAIN.LST). We could have used this version of the PRINT control:

```
-RUN :F1:PASC86 :F1:MAIN.SRC DEBUG PRINT(:LP:)<cr>
```

This version of the PRINT control sends the listing to the line printer (:LP:), rather than creating :F1:MAIN.LST as a listing file.

Most of the compiler control default settings are useful for everyday compiling; that is, there is no need to learn how to use the compiler controls unless you want to do something special. For example, if you want the compiler to issue a warning message whenever it sees a non-standard Pascal statement (an Intel extension to the standard Pascal language), use the NOEXTENSIONS control.

We used the DEBUG control for a good reason: we want to do symbolic debugging while the program is running (using DEBUG-86, described in Chapter 7). You will want to do symbolic debugging during the first run of your program. Use the DEBUG control to prepare your program for symbolic debugging unless your program is extremely large.

Most compiler controls can be specified in the invocation line as we show above. Most compiler controls can also be imbedded in the source file—as *control lines*. For example, we used the INCLUDE control in a control line:

```
$INCLUDE(:F1:INSPEC.SRC)
```

The INCLUDE control allows you to insert source statements from another file into this compilation. In this case, we wanted to insert the common interface specification (in :F1:INSPEC.SRC) into our compilation. The INCLUDE control saved us from typing the same interface specification for all three source files.

We also need to compile the other modules separately. The following invocation line compiles our **GetData** module in DUMDAT.SRC:

```
-RUN :F1:PASC86 :F1:DUMDAT.SRC DEBUG<cr>
SERIES III Pascal-86 V1.0
PARSE
  68   99  0  0
***WARNING, input:  ''END ''
***was repaired to: ''END ;''
END PARSE(1), ANALYZE(0), NOXREF, OBJECT

    COMPILATION OF GETDATA COMPLETED, 1 ERROR DETECTED.
    END OF Pascal-86 COMPILATION.
```

The compiler displays a sign-on message, then the word "PARSE" to show that it is parsing the program statements. During the parsing phase, the compiler discovered an error—the "END " statement was not punctuated correctly. The compiler repairs our error, and continues to compile. Each phase of compiling is displayed with a number in parentheses—the number of errors detected during the phase. The compiler only detected that one error, and since the error was easily repaired, the compilation was successful. We now have :F1:DUMDAT.OBJ containing the object module.

To compile our dummy **Operation** module, we use the following invocation line:

```
-RUN :F1:PASC86 :F1:DUMOP.SRC DEBUG<cr>
.
.
.
```

The listing files :F1:MAIN.LST, :F1:DUMDAT.LST, and :F1:DUMOP.LST are shown in figure 4-6.

```
SERIES-III Pascal-86, X031                                   09/01/80      PAGE  1
                                                                           MAINCONTROL


Source File: :F1:MAIN.SRC
Object File: :F1:MAIN.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:MAIN.SRC
   1    1  0  0         MODULE MainControl;
   2    2  0  0
                       (* Interface specification common to all modules *)

                       $INCLUDE(:F1:INSPEC.SRC)
                    =1 PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                    =1    CONST (*declarations declared publicly in this module*)
                    =1
                    =1      MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1          MinimumForHeatPump    =  13;
   5    7  0  0 =1
                    =1    TYPE (*definitions publicly defined in this module*)
                    =1
                    =1      AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1          WaterTemperature =0..120;
   7   12  0  0 =1          HeatingMethods   =(CollectorToExchanger,
                    =1                          CollectorToHeatPump,
                    =1                          TankToExchanger,
                    =1                          TankToHeatPump,
                    =1                          HeatedTankToHeatPump,
                    =1                          NoMethod);
   8   18  0  0 =1
                    =1          SystemData       = RECORD
   8   20  0  1 =1                                   ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                                   InsideTemp,
                    =1                            ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                                   CollectorWaterTemp,
                    =1                            TankWaterTemp,
                    =1                            HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                                   AmountOfSunlight      : Integer;
  12   27  0  1 =1                                   Hour                  : 00..24;
  13   28  0  1 =1                                   Minute                : 00..59;
  14   29  0  1 =1                                 END (*SystemData*);
  15   30  0  0 =1
                    =1    VAR (*variables publicly defined in this module.*)
                    =1
                    =1      CurrentData                 : SystemData;
  16   34  0  0 =1          Operating, Panic            : BOOLEAN;
  17   35  0  0 =1
                    =1    PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0 =1
                    =1      PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0 =1          PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0 =1
                    =1    PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0 =1                             StartUpSystem and ShutDownSystem*)
                    =1      PROCEDURE StartUpSystem;
  22   44  0  0 =1          PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0 =1          PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0 =1



                       PROGRAM MainControl(INPUT,OUTPUT);
  25    8  0  0
                       (* end of interface specification *)


                       PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
  26   13  1  0              BEGIN
  26   14  1  1                WITH CurrentData DO
  27   15  1  1                     BEGIN
  27   16  1  2                       IF InsideTemp<ThermostatSetting THEN
  28   17  1  2                          BEGIN
  28   18  1  3                            IF CollectorWaterTemp>MinimumForExchanger THEN
  29   19  1  3                               ChosenMethod:=CollectorToExchanger
                                          ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
  31   21  1  3                               ChosenMethod:=CollectorToHeatPump
                                          ELSE IF TankWaterTemp>MinimumForExchanger THEN
  33   23  1  3                               ChosenMethod:=TankToExchanger
                                          ELSE IF TankWaterTemp>MinimumForHeatPump THEN
  35   25  1  3                               ChosenMethod:=TankToHeatPump
                                          ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
  37   27  1  3                               ChosenMethod:=HeatedTankToHeatPump
                                          ELSE Panic:=TRUE; Operating:=FALSE;
  40   29  1  3                          END
  41   30  1  2                     ELSE (*no heating request*)ChosenMethod:=NoMethod
  42   31  1  2                END; (*With CurrentData*)
  44   32  1  1           END; (*DetermineMethod)
  45   33  0  0
```

Figure 4-6. Listings of Our Test Modules

```
                        (************** MAIN PROGRAM ************************************)

                        BEGIN
  45   37  0  1            StartUpSystem;
  46   38  0  1            Operating:=TRUE;
  47   39  0  1            Panic:=FALSE;
  48   40  0  1            WHILE Operating DO (*while system is operating, do:*)
  49   41  0  1               BEGIN
  49   42  0  2                  GetData(CurrentData); (*Get the temps, time, etc.*)
  50   43  0  2                  StoreData(CurrentData); (*Store this data as record*)
  51   44  0  2                  DetermineMethod(CurrentData); (*this detects a panic*)
  52   45  0  2                  OperateSystem(CurrentData);
  53   46  0  2               END; (*while operating*)
  55   47  0  1            ShutDownSystem(CurrentData);
  56   48  0  1         END. (*Main Control Algorithm*)


Summary Information:

PROCEDURE                       OFFSET    CODE SIZE     DATA SIZE    STACK SIZE
DETERMINEMETHOD                  0011H  008FH   143D                 0006H     6D

Total                                   0147H   327D  0016H    22D  004CH    76D

  94 Lines Read.
   0 Errors Detected.


 33% Utilization of Memory.




SERIES-III Pascal-86, X031                                          09/01/80          PAGE   1


Source File: :F1:DUMDAT.SRC
Object File: :F1:DUMDAT.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMDAT.SRC
   1    1  0  0         (*This is a dummy GetData module, with dummy GetData
                        and StoreData procedures, for use with MainControl
                        module in testing phases.  It only performs console
                        input to get Celsius temperatures, the time of day,
                        and the amount of sunlight (insolation) for the
                        solar collector.  Use PLM86$DATA module for real
                        application.*)

                        MODULE GetData;
   2   10  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                  =1    PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                  =1       CONST (*declarations declared publicly in this module*)
                  =1
                  =1         MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1           MinimumForHeatPump    =  13;
   5    7  0  0 =1
                  =1       TYPE (*definitions publicly defined in this module*)
                  =1
                  =1         AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1           WaterTemperature =0..120;
   7   12  0  0 =1           HeatingMethods   =(CollectorToExchanger,
                  =1                           CollectorToHeatPump,
                  =1                           TankToExchanger,
                  =1                           TankToHeatPump,
                  =1                           HeatedTankToHeatPump,
                  =1                           NoMethod);
   8   18  0  0 =1
                  =1           SystemData       = RECORD
   8   20  0  1 =1                                 ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                                 InsideTemp,
                  =1                              ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                                 CollectorWaterTemp,
                  =1                              TankWaterTemp,
                  =1                              HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                                 AmountOfSunlight      : Integer;
  12   27  0  1 =1                                 Hour                  : 00..24;
  13   28  0  1 =1                                 Minute                : 00..59;
  14   29  0  1 =1                               END (*SystemData*);
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
  15   30  0  0 =1
                =1          VAR (*variables publicly defined in this module.*)
                =1
                =1            CurrentData                 : SystemData;
  16   34  0  0 =1            Operating, Panic            : BOOLEAN;
  17   35  0  0 =1
                =1        PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0 =1
                =1           PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0 =1
                =1        PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                =1           PROCEDURE StartUpSystem;
  22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0 =1

                        PRIVATE GetData;
  25   16  0  0
                        (* end of interface specification *)


                        PROCEDURE GetData(VAR CurrentData:SystemData);
  26   21  1  0          BEGIN
  26   22  1  1             WITH CurrentData DO BEGIN
  27   23  1  2                WRITE('Type the thermostat setting in degrees Celsius:');
  28   24  1  2                READLN(ThermostatSetting); WRITELN;
  30   25  1  2                WRITE('Type the inside temperature reading in Celsius:');
  31   26  1  2                READLN(InsideTemp); WRITELN;
  33   27  1  2                WRITE('Type the temperature of the collector water in Celsius:');
  34   28  1  2                READLN(CollectorWaterTemp); WRITELN;
  36   29  1  2                WRITE('Type the temperature of the tank water in Celsius:');
  37   30  1  2                READLN(TankWaterTemp); WRITELN;
  39   31  1  2                WRITE('Type the temperature of the heated tank water in Celsius:');
  40   32  1  2                READLN(HeatedTankTemp); WRITELN;
  42   33  1  2                WRITE('Type the hour of day, as in 04 or 24:');
  43   34  1  2                READLN(Hour); WRITELN;
  45   35  1  2                WRITE('Type the minute of the hour, as in 01 or 59: ');
  46   36  1  2                READLN(Minute); WRITELN;
  48   37  1  2                WRITE('Type the amount of sunlight, any integer will do for now:');
  49   38  1  2                READLN(AmountOfSunlight); WRITELN;
  51   39  1  2             END;  (*with CurrentData*)
  53   40  1  1          END;
  54   41  0  0
                        PROCEDURE StoreData(VAR CurrentData:SystemData);
  55   43  1  0          BEGIN
  55   44  1  1          (*Dummy procedure, eventually will store CurrentData in a file*)
                             WITH CurrentData DO BEGIN
  56   46  1  2                WRITELN('----------------------------');
  57   47  1  2                WRITELN('CURRENT DATA IS AS FOLLOWS:');
  58   48  1  2                WRITELN('----------------------------');
  59   49  1  2                WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  60   50  1  2                WRITELN('Inside temperature is ',InsideTemp,'C');
  61   51  1  2                WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  62   52  1  2                WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  63   53  1  2                WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  64   54  1  2                WRITELN('Time of day is ',Hour,':',Minute);
  65   55  1  2                WRITELN('Amount of sunlight is ',AmountOfSunlight);
  66   56  1  2                WRITELN; (*a blank line*)
  67   57  1  2             END; (*with CurrentData*)
  69   58  1  1          END
***WARNING, input: "END "
***was repaired to "END ; "
  70   58  0  0             .


Summary Information:

PROCEDURE                    OFFSET    CODE SIZE     DATA SIZE     STACK SIZE
STOREDATA                     0405H  0222H   546D                  0010H    16D
GETDATA                       0294H  0241H   577D                  0010H    16D

Total                                06F7H  1783D  0000H     0D   0020H    32D

 104 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
SERIES-III Pascal-86, V031                                    09/01/80        PAGE   1


Source File: :F1:DUMOP.SRC
Object File: :F1:DUMOP.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMOP.SRC
   1    1  0  0         (*This is a dummy Operation module, with dummy StartUpSystem,
                        ShutDownSystem, and OperateSystem procedures,
                        for use with MainControl module in testing phases.
                        *)

                        MODULE Operation;
   2    7  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                    =1  PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0     =1
                    =1      CONST (*declarations declared publicly in this module*)
                    =1
                    =1        MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0     =1        MinimumForHeatPump    =  13;
   5    7  0  0     =1
                    =1      TYPE (*definitions publicly defined in this module*)
                    =1
                    =1        AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0     =1        WaterTemperature =0..120;
   7   12  0  0     =1        HeatingMethods   =(CollectorToExchanger,
                    =1                          CollectorToHeatPump,
                    =1                          TankToExchanger,
                    =1                          TankToHeatPump,
                    =1                          HeatedTankToHeatPump,
                    =1                          NoMethod);
   8   18  0  0     =1
                    =1        SystemData       = RECORD
   8   20  0  1     =1                             ChosenMethod          : HeatingMethods;
   9   21  0  1     =1                             InsideTemp,
                    =1                             ThermostatSetting     : AirTemperature;
  10   23  0  1     =1                             CollectorWaterTemp,
                    =1                             TankWaterTemp,
                    =1                             HeatedTankTemp        : WaterTemperature;
  11   26  0  1     =1                             AmountOfSunlight      : Integer;
  12   27  0  1     =1                             Hour                  : 00..24;
  13   28  0  1     =1                             Minute                : 00..59;
  14   29  0  1     =1                           END (*SystemData*);
  15   30  0  0     =1
                    =1      VAR (*variables publicly defined in this module.*)
                    =1
                    =1        CurrentData                 : SystemData;
  16   34  0  0     =1        Operating, Panic            : BOOLEAN;
  17   35  0  0     =1
                    =1    PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0     =1
                    =1      PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0     =1      PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0     =1
                    =1    PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0     =1                       StartUpSystem and ShutDownSystem*)
                    =1      PROCEDURE StartUpSystem;
  22   44  0  0     =1      PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0     =1      PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0     =1

                        PRIVATE Operation;
  25   13  0  0
                        (* end of interface specification *)


                        PROCEDURE StartUpSystem;
  26   18  1  0     BEGIN
  26   19  1  1           WRITELN ('Climate system is now on.');
  27   20  1  1           WRITELN ('-------------------------');
  28   21  1  1           WRITELN;
  29   22  1  1     END;
  30   23  0  0

                        PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  31   26  1  0     BEGIN
  31   27  1  1       WITH CurrentData DO BEGIN
  32   28  1  2         WRITELN('=========================================');
  33   29  1  2         WRITELN('The Climate System is now operating.');
  34   30  1  2         WRITELN;
  35   31  1  2         WRITELN('The Time is ',Hour,':',Minute);
  36   32  1  2         WRITELN('The inside temperature is ',InsideTemp,'C');
  37   33  1  2         WRITELN('The thermostat setting is ',ThermostatSetting,'C');
  38   34  1  2         WRITE('Method chosen to heat the building: ');
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

```
  39   35  1  2             CASE ChosenMethod OF
  40   36  1  3                CollectorToExchanger: WRITE('Solar Collector to Exchanger');
  41   37  1  3                CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
  42   38  1  3                TankToExchanger     : WRITE('Tank to Exchanger');
  43   39  1  3                TankToHeatPump      : WRITE('Tank to Heat Pump');
  44   40  1  3                HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
  45   41  1  3                NoMethod            : WRITE('No heat required');
  46   42  1  3             END;
  48   43  1  2             WRITELN;
  49   44  1  2             WRITELN('=============================================');
  50   45  1  2             WRITELN; (*write a blank line.*)
  51   46  1  2          END;
  53   47  1  1       END;(*OperateSystem*)
  54   48  0  0
                       PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  55   51  1  0       BEGIN
  55   52  1  1       WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
  56   53  1  1          IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
  57   54  1  1                   ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
  59   55  1  1          WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
  60   56  1  1          WITH CurrentData DO BEGIN
  61   57  1  2             WRITE('Last chosen heating method was: ');
  62   58  1  2             CASE ChosenMethod OF
  63   59  1  3                CollectorToExchanger: WRITE('Solar Collector to Exchanger');
  64   60  1  3                CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
  65   61  1  3                TankToExchanger     : WRITE('Tank to Exchanger');
  66   62  1  3                TankToHeatPump      : WRITE('Tank to Heat Pump');
  67   63  1  3                HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
  68   64  1  3                NoMethod            : WRITE('No heat required');
  69   65  1  3             END;
  71   66  1  2             WRITELN;
  72   67  1  2             WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  73   68  1  2             WRITELN('Inside temperature is ',InsideTemp,'C');
  74   69  1  2             WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  75   70  1  2             WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  76   71  1  2             WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  77   72  1  2             WRITELN('Time of day is ',Hour,':',Minute);
  78   73  1  2             WRITELN('Amount of sunlight is ',AmountOfSunlight);
  79   74  1  2          END;(*with CurrentData*)
  81   75  1  1          WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
  82   76  1  1          WRITELN;
  83   77  1  1          WRITELN('Goodnight, Irene...');
  84   78  1  1       END
***WARNING, input: "END "
***was repaired to "END ; "
  85   78  0  0          .(*ShutDownSystem*)

Summary Information:

PROCEDURE                    OFFSET   CODE SIZE     DATA SIZE     STACK SIZE
SHUTDOWNSYSTEM               0698H  0398H   920D                  0010H    16D
OPERATESYSTEM                0440H  0258H   600D                  0010H    16D
STARTUPSYSTEM                03E2H  005EH    94D                  0010H    16D

Total                                0A30H  2608D  0000H     0D   0030H    48D

 125 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

Figure 4-6. Listings of Our Test Modules (Cont'd.)

Summary

We now have three Pascal-86 modules: **MainControl**, **GetData**, and **Operation**. Two of these modules, **GetData** and **Operation**, are dummy versions that do not interact with any hardware except the Series III system; we will use them for examples in subsequent chapters. The **MainControl** module can remain unchanged even in our final application.

The final application will probably use an 8088 processor. In Chapter 5, we show PL/M-86 procedures we can use to obtain data from thermocouples via the input/output ports on the 8088. Since you probably do not have prototype climate control hardware with an 8088, we will not include these procedures in our test versions for execution on a Series III system; nevertheless, they are good examples of PL/M-86 procedures.

Our Pascal-86 modules cannot run by themselves on a Series III. Certain built-in procedures (like WRITELN and READLN) rely on *run-time support software*, which consists of public modules that contain the software needed to perform console input/output and other operations. In Chapter 6, we'll show you how to link the run-time support libraries to these Pascal-86 modules to make them executable on a Series III system.

CHAPTER 5
PROGRAMMING IN OTHER LANGUAGES

"It is possible by ingenuity and at the expense of clarity ... [to do almost anything in any language]. However, the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there."

—Christopher Strachey
NATO Summer School in Programming

The Intellec Series III system also supports PL/M, FORTRAN, and assembly language programming for iAPX 86,88 and 8080/8085 applications. In the previous chapter we used Pascal-86 for our main control algorithm, but a modular strategy might take advantage of other languages for other modules.

ANOTHER LOOK AT CHOOSING LANGUAGES FOR MODULES

In the best of all possible worlds, would we all speak the same language? Tower of Babel enthusiasts would have us coding our entire program in one language—but which? We do not want to return to the stone age and lose cultural variety and language diversity. There are expressions that can only be expressed in Chinese characters, and there are problem-solving statements that are better expressed in PL/M than in FORTRAN. A good carpenter should have more tools than nails and a hammer; a good programmer should be fluent in several programming languages.

When you design an algorithm, design it using a comfortable language. You will find the algorithm easier to debug, and you will notice the paradigm inherent in your design. When you are ready to translate your algorithm into code for a computer, use the language best suited for the paradigm.

If you have many algorithms that must work together, you should keep the communication among them simple. With simple interfaces, you can code each algorithm in the language best suited for the algorithm. In some cases, you can use an algorithm that has already been developed for use in another application—another reason for keeping your algorithms and interfaces simple. With the Intellec Series III development tools, you can link these algorithms in different configurations to form several applications.

For example, we chose to write our main control algorithm for the climate system in Pascal-86. We still need an algorithm for retrieving the data and converting it to Celsius temperatures. We decided to write a simple Pascal-86 routine for testing purposes only (this routine only retrieves the data from the Series III console); however, our final product will rely on thermocouples and other sources of data, and we will need an algorithm to convert thermocouple voltages to degrees Celsius. Fortunately, we already have a routine in PL/M-86 that performs this activity, and we can save development time and money by using it.

PROGRAMMING IN PL/M-86

PL/M is renowned for its structure and versatility. PL/M is one of the only structured high-level languages that allow you to manipulate bits with AND, OR, and shift (SHR for "shift right" and SHL for "shift left") operations. PL/M's data types are not as strictly enforced as Pascal's—PL/M's BYTE, WORD (ADDRESS), and POINTER types have loose definitions so that you can use them for different kinds of data. For this reason, PL/M is easier to use for system programming (designing computer systems or control mechanisms), yet harder to use in application programming where enforced data typing makes it easier to write error-free programs.

In our climate control system, there are two routines whose paradigms lend themselves easily to PL/M: the routine to get BCD digits from a thermostat device and convert them to a Celsius temperature, and the routine to get voltage data from a thermocouple and convert the voltages to Celsius temperatures. Figure 5-1 shows the algorithm and the actual PL/M-86 code for the routine to retrieve data from a thermostat device. You are already familiar with comments in Pascal programs that occur between the (* and *) symbols; in PL/M, comments occur between the /* and */ symbols.

A PL/M *typed procedure* is like a Pascal or FORTRAN function: it is called in an assignment statement (as in X:=FUNCTION(Y)), and it returns a value (X receives the value of FUNCTION(Y)). In figure 5-1, the typed procedure THERMOSTAT$SETTING$FROM$PORTS returns the value of THERMO$SETTING, which it computes by accessing the two ports HIGH and LOW and converting the BCD digits to a Celsius temperature. The value of THERMO$SETTING is assigned to **ThermostatSetting** in the **GetData** procedure's assignment statement:

```
ThermostatSetting:=THERMOSTAT$SETTING$FROM$PORTS(StatPort1,Stat Port2);
```

```
PLMDATA: DO;

/* This module holds the procedures THERMOSTAT$SETTING$FROM$PORTS,
   TEMP$DATA$FROM$PORTS, and INTERPOLATE (a procedure used by
   TEMP$DATA$FROM$PORTS). These will be used in the final testing
   phase of the climate control system (when prototype hardware is
   available). For intermediate testing (and examples in this book),
   do not use this module; use the dummy GetData module. */

/*
The algorithm for getting a Celsius temperature from a thermostat
device that can send BCD digits to two ports of the 8088:

   The final version of the GetData procedure (to be
   written in Pascal-86) will use this statement to get
   the thermostat reading:

ThermostatSetting:=THERMOSTAT$SETTING$FROM$PORTS(StatPort1,StatPort2);

   A PL/M-86 typed procedure called THERMOSTAT$SETTING$FROM$PORTS
   receives two port numbers from GetData: StatPort1 and StatPort2.
   THERMOSTAT$SETTING$FROM$PORTS must access these ports, convert
   the BCD digits to a Celsius temperature, and return the
   temperature.
```

Figure 5-1. The PL/M-86 Typed Procedure THERMOSTAT$SETTING$FROM$PORTS

```
INPUTS
------

THERMOSTAT$SETTING$FROM$PORTS:

Formal parameters HIGH and LOW receive port numbers as actual
parameters.

Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                    Port HIGH, bits 3-0: hundred's digit
                    Port LOW, bits 7-4: ten's digit
                    Port LOW, bits 3-0: unit's digit


OUTPUT
------

THERMOSTAT$SETTING$FROM$PORTS: Return WORD with Celsius temperature
*/

/*Here is the typed procedure THERMOSTAT$SETTING$FROM$PORTS:*/

THERMOSTAT$SETTING$FROM$PORTS:
     PROCEDURE (HIGH, LOW) WORD;
        DECLARE (HIGH, LOW) WORD;
        DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
        DECLARE THERMO$SETTING WORD;
        DECLARE (HUNDREDS, TENS, UNITS) BYTE;

        IN$PORT$HIGH = INPUT(HIGH);
        IN$PORT$LOW = INPUT(LOW);
        HUNDREDS = IN$PORT$HIGH AND 00001111B;
        TENS = SHR(IN$PORT$LOW, 4);
        UNITS = IN$PORT$LOW AND 00001111B;
        THERMO$SETTING = UNITS + 10*TENS + 100*HUNDREDS;

        RETURN THERMO$SETTING; /*this returns the temperature*/
END THERMOSTAT$SETTING$FROM$PORTS;

.
.
.
More procedures follow--see figure 5-2.
```

Figure 5-1. The PL/M-86 Typed Procedure THERMOSTAT$SETTING$FROM$PORTS (Cont'd.)

There are notable similarities between Pascal and PL/M. Most notable are the logical structures that can occur in both languages—both have the DO WHILE and IF-THEN-ELSE constructs. The languages are lexically and syntactically different in data declarations, procedure headings, and other constructs, but they are logically similar. By conforming to a logical structure, you make your program readable and easier to debug.

The data declarations in both languages are very similar. In both languages, you must declare your data identifiers to be of some type before using the identifiers. In Pascal, you define data types or use predefined Pascal data types. In PL/M, you are restricted to the acceptable PL/M data types, but they are loosely defined. A BYTE can be any value expressed in eight bits, and a WORD in PL/M-86 (ADDRESS in PL/M-80) can be any value expressed in sixteen bits; both types of values are treated as unsigned integers. PL/M-86 offers several more types: INTEGER (for signed integers), REAL (for floating point numbers), and POINTER (for 8086 and 8088 addressing).

PL/M-86 offers many features useful for system programming— arrays and structures, based variables, easy type conversion, built-in procedures for manipulating strings, setting and disabling interrupts, accessing the 8086 or 8088 hardware stack pointer and base registers, and performing bit shift and rotate operations. We use the AND and SHR operators in the THERMOSTAT$SETTING$FROM$PORTS procedure in figure 5-1.

We also use PL/M-86 procedures to obtain temperature data from thermocouple voltage data. The **GetData** procedure in our **GetData** module (written in Pascal-86) calls the PL/M-86 procedure TEMP$DATA$FROM$PORTS to obtain the temperatures **InsideTemp**, **CollectorWaterTemp**, **TankWaterTemp**, and **HeatedTankTemp** using these assignment statements:

```
InsideTemp:=TEMP$DATA$FROM$PORTS(InsidePort1,InsidePort2);
CollectorWaterTemp:=TEMP$DATA$FROM$PORTS(CollectPort1,CollectPort2);
TankWaterTemp:=TEMP$DATA$FROM$PORTS(TankPort1,TankPort2);
HeatedTankTemp:=TEMP$DATA$FROM$PORTS(HotTankPort1,HotTankPort2);
```

In all four assignments, the Pascal-86 identifiers on the left side of the := symbols receive the values from the PL/M-86 typed procedure TEMP$DATA$FROM$PORTS. Figure 5-2 shows TEMP$DATA$FROM$PORTS.

```
.
.
.
/*
The typed procedure TEMP$DATA$FROM$PORTS receives two port numbers:
HIGH and LOW. These ports are accessed for the binary ADC output
from a thermocouple device: HIGH gets the high-order 8 bits, and
LOW gets the low-order 8 bits. TEMP$DATA$FROM$PORTS then uses
the typed procedure INTERPOLATE, a routine that interpolates a
Celsius temperature from a thermocouple voltage using two tables.
TEMP$DATA$FROM$PORTS sends a WORD with the input voltage to
INTERPOLATE. INTERPOLATE returns a Celsius temperature, which
is returned to the GetData procedure (written in Pascal-86).

INPUTS
------

TEMP$DATA$FROM$PORTS:

Formal parameters HIGH and LOW receive port numbers as actual
parameters.

Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
Input port LOW: Binary ADC output of thermocouple, low-order 8 bits
```

Figure 5-2. The PL/M-86 Typed Procedures TEMP$DATA$FROM$PORTS and INTERPOLATE

```
OUTPUT
------

TEMP$DATA$FROM$PORTS: Return WORD with temperature in Celsius

*/

/* INTERPOLATE is declared first, then its calling procedure
   TEMP$DATA$FROM$PORTS is declared. */

/* INTERPOLATE is a typed procedure that receives thermocouple
   voltage and returns temperature in Celsius using an
   interpolation routine. */

INTERPOLATE:
   PROCEDURE(VOLT$IN) WORD;
      DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
      DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
      DECLARE (I, VOLT$IN, NUMERATOR) WORD;

      I = 0;
      IF VOLT$IN=0 THEN RETURN T$CEL(I);
      DO WHILE VOLT$IN > VOLTS(I);
         I = I + 1;
      END;
      /* Shift for rounding, and return Celsius temperature */
      NUMERATOR = SHL((VOLT$IN-VOLTS(I-1))*(T$CEL(I)- T$CEL(I-1)), 1);
      RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)- VOLTS(I-1))+1, 1);

END INTERPOLATE;

/*****************************************************************/
/* TEMP$DATA$FROM$PORTS */

TEMP$DATA$FROM$PORTS:
   PROCEDURE(HIGH,LOW) WORD;
      DECLARE (HIGH,LOW) WORD;
      DECLARE IN$PORT$HIGH WORD;
      DECLARE IN$PORT$LOW BYTE;
      DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

      IN$PORT$HIGH = INPUT(HIGH);
      IN$PORT$LOW = INPUT(LOW);
      THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
      TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
      RETURN TEMPERATURE;

END TEMP$DATA$FROM$PORTS;
```

Figure 5-2. The PLM-86 Typed Procedures TEMP$DATA$FROM$PORTS and INTERPOLATE (Cont'd.)

Let's look closely at the following statement, which appears in the procedure TEMP$DATA$FROM$PORTS shown in figure 5-2:

```
THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
```

The variable IN$PORT$HIGH was declared as a WORD with 16 bits, and the variable IN$PORT$LOW was declared as a BYTE with 8 bits. The thermocouple voltage data from an analog-to-digital converter can have up to 13 bits, but our procedure was originally written to access 8-bit ports. To assemble the 13 bits, we use the SHL (shift to the left) built-in procedure to shift the rightmost 8 bits of IN$PORT$HIGH to the left, and we OR this shifted value with IN$PORT$LOW.

The INTERPOLATE procedure uses a more complicated expression that includes both the SHL (shift to the left) and SHR (shift to the right) built-in procedures. The INTERPOLATE procedure also uses two tables, or *arrays*. They are declared in the following statements:

```
DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
```

In both declarations, the arrays VOLTS and T$CEL are assigned actual values through the use of DATA initializations. The DATA initialization allocates storage for the array and assigns the values specified in parentheses after the word DATA in a single step.

The values chosen for the VOLTS array are from the National Bureau of Standards; they represent the output (in millivolts) of type J thermocouples that corresponds to the Celsius temperature range assigned to the T$CEL array. For example, a thermocouple output of 102 millivolts should correspond roughly to 20 degrees Celsius. The INTERPOLATE procedure uses these tables to arrive at an approximate Celsius temperature value for a known thermocouple output value. Our calculations would be more accurate if the ranges between values within each table were smaller.

Compiling a PL/M-86 program is very similar to compiling a Pascal-86 program. We execute the PL/M-86 compiler in the 8086 execution environment by using the Series III RUN command. The PL/M-86 compiler is supplied in the file PLM86.86 on the PL/M-86 disk. We inserted a copy of this disk into drive 1 (we also put our source program on the same disk). In the following example, we execute the PLM86.86 using the RUN command. We do not have to supply the ".86" extension, since RUN already assumes that the file you specify has that extension. The following example shows the RUN command line for compiling the module PLMDATA, which is in a source file called PLMDAT.SRC (PLMDAT.SRC and the PL/M-86 compiler are both in directory :F1:):

```
-RUN :F1:PLM86 :F1:PLMDAT.SRC CODE LARGE<cr>
```

This compiler invocation produces two output files: :F1:PLMDAT.OBJ to hold the compiled object module, and :F1:PLMDAT.LST to hold the listing. The listing is shown in figure 5-3 (in the next section).

Two *compiler controls*, CODE and LARGE, were also specified. The CODE control tells the compiler to list the approximate assembly language instructions that would be necessary to implement the PL/M-86 statements—this is useful for many reasons, some of which are described in the next section. The LARGE control is needed here because Pascal-86 modules are compiled with a default size control that is equivalent to the PL/M-86 LARGE model. All modules of a program must conform to the same size control, so our PL/M-86 module must be compiled as a LARGE module to conform to the default size of Pascal-86 modules.

You need to know more about the PL/M-86 language and compiler than the brief introduction provided in this guide. Intel provides a manual for the PL/M-86 language and compiler (*PL/M-86 User's Guide for 8086-Based Development Systems*). Intel also supplies two manuals for PL/M-80 program development (*PL/M-80 Programming Manual* and *ISIS-II PL/M-80 Compiler Operator's Manual*) in the 8085 execution environment of the Series III. For tutorial information on the PL/M language, see *A Guide to PL/M Programming For Microcomputer Applications* by Daniel McCracken (listed in the Bibliography).

PROGRAMMING IN 8086/8087/8088 ASSEMBLY LANGUAGE

So far we have described high-level languages that are translated by compilers into machine code; namely, Pascal-86 and PL/M-86. Another high-level language not described in this book is FORTRAN, which is also translated by a compiler into machine code. Other high-level languages like BASIC-80 are translated by interpreters into machine code.

An *assembly language* program is translated into machine code by an *assembler*. Intel provides the 8086/8087/8088 Macro Assembler, which is described in this section, to translate 8086/8087/8088 Assembly Language programs. It is called a macro assembler because it will also translate *macro definitions* written in the Macro Processor Language, which is described with the 8086/8087/8088 Assembly Language.

The common denominator of all these languages is the *machine code*, which is the binary language of ones and zeros that only the processor can "speak" well. The following is an example of machine code, with comments to explain what action each code performs:

Memory Address (Hexadecimal)	Memory Contents (Binary)	Comments (English)
00000	11100101	Read word into reg. AX...
00001	00000101	...from input port 5.
00002	01000000	Increment contents of AX.
00003	11100111	Write word from reg. AX...
00004	00000010	...to output port 2.
00005	11101011	Repeat by jumping...
00006	11111001	...back seven bytes.
00007	...	

This machine code (sometimes called *object code*) is the code that the processor executes. All languages are eventually translated into this type of code.

A program written in assembly language is a symbolic representation of machine code. The relation between assembly language instructions and the resulting machine code is usually very obvious; the relation between statements in a high-level language and the resulting machine code is often not obvious (with some exceptions in the PL/M-86 language).

Assembly language gives you complete control over the resulting machine code and thereby allows you to generate very efficient machine code. There are times when this control is desirable, and other times when you want to be free of such details. Most high-level language compilers are efficient enough for microcomputer applications; in fact, some compilers are more efficient than most humans.

Assembly language is the closest language to machine code, but it does allow you to use symbolic names. Here is a rewrite of the machine code instructions shown before, using 8086/8087/8088 Assembly Language (comments follow the semicolons):

```
CYCLE:    IN      AX,5        ;Read word from port 5 into reg. AX.
          INC     AX          ;Increment the contents of reg. AX.
          OUT     2,AX        ;Write word from reg. AX to port 2.
          JMP     CYCLE       ;Jump to beginning and repeat.
```

This program fragment is simpler to read because it uses symbolic names like CYCLE instead of binary and hexadecimal numbers. The 8086/8087/8088 Assembly Language also provides sophisticated code and data structuring mechanisms usually found only in high-level languages. The assembler enforces some consistency in data types to prevent inadvertent errors, yet it also allows some deliberate ways to override data types.

The 8086/8087/8088 Macro Assembler also processes macro definitions. A *macro* is a shorthand function name for a string of instructions. First you define a macro, using the Macro Processing Language, to be a sequence of assembly language instructions. Once defined, you can specify the macro name in an assembly language program, and the macro assembler automatically replaces the macro name with the actual sequence of instructions from the definition. Using this facility you can create many macros and use them in many programs.

There are many times when an assembly language version of a routine runs faster and takes up less space than a high-level language version. Intel's compilers can produce a listing of the assembly language instructions that are approximately the ones you would use to implement the compiled high-level language routine in assembly language. For example, we used the CODE control in the previous section when we compiled the PL/M-86 program PLMDAT.SRC. The CODE control produced the listing shown in figure 5-3.

PROGRAMMING FOR THE SERIES III ENVIRONMENT

Assembly language and PL/M give you more direct control over the processor's operation; however, the Series III system gives you a set of operating system procedures called *primitives* (or *service routines*) that your programs can use to perform standard operations. By using these primitives, you save development time in two ways: first, you save time by not implementing the standard operations yourself; and second, you save time in the future by writing programs that will always be compatible with future Intel operating systems.

In Pascal-86, you are supplied with built-in procedures that automatically call these primitives. You link the run-time libraries that contain the primitives to your Pascal-86 modules, and you're all set to run the program on a Series III system. The program will also run on future Intel operating systems, since the only changes it would need would be contained in the run-time libraries; that is, you would have a different set of run-time libraries for each system, but your basic program modules would remain unchanged.

To have this modularity in PL/M or assembly language, you simply use the set of primitives described in the *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, and then link in the appropriate system libraries to your PL/M or assembly language modules (as described in the *iAPX 86,88 Family Utilities User's Guide*). For future Intel operating systems, you only need to use a different set of system libraries. Your basic program modules would remain unchanged.

The next chapter describes the linking and locating operations for our Pascal-86 modules.

```
PL/M-86 COMPILER    PLMDATA                                                  PAGE   1


ISIS-II PL/M-86 V121 COMPILATION OF MODULE PLMDATA
OBJECT MODULE PLACED IN PLMEX.OBJ
COMPILER INVOKED BY:  PLM86 PLMEX.SRC CODE


   1          PLMDATA: DO;

              /*
              INPUTS
              ------

              THERMOSTAT$SETTING$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                                   Port HIGH, bits 3-0: hundred's digit
                                   Port LOW, bits 7-4: ten's digit
                                   Port LOW, bits 3-0: unit's digit


              TEMP$DATA$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
              Input port LOW: Binary ADC output of thermocouple, low-order 8 bits


              OUTPUTS
              -------

              THERMOSTAT$SETTING$FROM$PORTS:  Return WORD with setting in Celsius

              TEMP$DATA$FROM$PORTS:  Return WORD with temperature in Celsius

              */


   2   1      THERMOSTAT$SETTING$FROM$PORTS:
                                                        ; STATEMENT # 2
                        THERMOSTATSETTINGFROMPORTS      PROC NEAR
          0000  55              PUSH    BP
          0001  8BEC            MOV     BP,SP
             PROCEDURE (HIGH, LOW) WORD;
   3   2         DECLARE (HIGH, LOW) WORD;
   4   2         DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
   5   2         DECLARE THERM0$SETTING WORD;
   6   2         DECLARE (HUNDREDS, TENS, UNITS) BYTE;

   7   2         IN$PORT$HIGH = INPUT(HIGH);
                                                        ; STATEMENT # 7
          0003  8B5606          MOV     DX,[BP].HIGH
          0006  EC              IN      DX
          0007  88060C00        MOV     INPORTHIGH,AL
   8   2         IN$PORT$LOW = INPUT(LOW);
                                                        ; STATEMENT # 8
          000B  8B5604          MOV     DX,[BP].LOW
          000E  EC              IN      DX
          000F  88060000        MOV     INPORTLOW,AL
   9   2         HUNDREDS = IN$PORT$HIGH AND 00001111B;
                                                        ; STATEMENT # 9
          0013  8A060C00        MOV     AL,INPORTHIGH
          0017  80E00F          AND     AL,0FH
          001A  88060E00        MOV     HUNDREDS,AL
  10   2         TENS = SHR(IN$PORT$LOW, 4);
                                                        ; STATEMENT # 10
          001E  8A060D00        MOV     AL,INPORTLOW
          0022  B104            MOV     CL,4H
          0024  D2E8            SHR     AL,CL
          0026  88060F00        MOV     TENS,AL
  11   2         UNITS = IN$PORT$LOW AND 00001111B;
                                                        ; STATEMENT # 11
          002A  8A060D00        MOV     AL,INPORTLOW
          002E  80E00F          AND     AL,0FH
          0031  88061000        MOV     UNITS,AL
```

Figure 5-3. Listing of PLMDATA with the CODE Control

```
12  2       THERMO$SETTING = UNITS + 10*TENS + 100*HUNDREDS;
                                                ; STATEMENT # 12

        0035  8A060F00          MOV     AL,TENS
        0039  B10A              MOV     CL,0AH
        0038  F6E1              MUL     CL
        003D  8A0E1000          MOV     CL,UNITS
        0041  B500              MOV     CH,0H
        0043  03C1              ADD     AX,CX
        0045  50                PUSH    AX      ; 1
        0046  8A060E00          MOV     AL,HUNDREDS
        004A  B164              MOV     CL,64H
        004C  F6E1              MUL     CL
        004E  59                POP     CX      ; 1
        004F  03C1              ADD     AX,CX
        0051  89060000          MOV     THERMOSETTING,AX
13  2       RETURN THERMO$SETTING;
                                                ; STATEMENT # 13
        0055  5D                POP     BP
        0056  C20400            RET     4H
14  2     END THERMOSTAT$SETTINGSFROM$PORTS;
                                                ; STATEMENT # 14
                    THERMOSTATSETTINGFROMPORTS      ENDP


          /* Another typed procedure to return temperature data, which
             uses the INTERPOLATE typed procedure. */

          /* INTERPOLATE is a typed procedure that receives thermocouple
             voltage and returns temperature in Celsius using an
             interpolation routine */

15  1     INTERPOLATE:
                                                ; STATEMENT # 15
                  INTERPOLATE       PROC NEAR
        0059  55                PUSH    BP
        005A  8BEC              MOV     BP,SP
      PROCEDURE(VOLT$IN) WORD;
16  2       DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
17  2       DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
18  2       DECLARE (I, VOLT$IN, NUMERATOR) WORD;

19  2       I = 0;
                                                ; STATEMENT # 19
        005C  C706020000000     MOV     I,0H
20  2       IF VOLT$IN<1 THEN RETURN T$CEL(I);
                                                ; STATEMENT # 20
        0062  817E040100        CMP     [BP].VOLTIN,1H
        0067  7203              JB      $+5H
        0069  E90000            JMP     @1
                                                ; STATEMENT # 21
        006C  BB0000            MOV     BX,0H
        006F  D1E3              SHL     BX,1
        0071  8B871000          MOV     AX,TCEL[BX]
        0075  5D                POP     BP
        0076  C20200            RET     2H
                    @1:
22  2       DO WHILE VOLT$IN > VOLTS(I);
                                                ; STATEMENT # 22
                    @2:
        0079  8B1E0200          MOV     BX,I
        007D  D1E3              SHL     BX,1
        007F  8B4604            MOV     AX,[BP].VOLTIN
        0082  3B870000          CMP     AX,VOLTS[BX]
        0086  7703              JA      $+5H
        0088  E90700            JMP     @3
23  3           I = I + 1;
                                                ; STATEMENT # 23
        0088  FF060200          INC     I
24  3       END;
                                                ; STATEMENT # 24
        008F  E9E7FF            JMP     @2
                    @3:

          /* Shift for rounding, and return Celsius temperature */
25  2       NUMERATOR = SHL((VOLT$IN-VOLTS(I-1))*(T$CEL(I)-T$CEL(I-1)), 1);
                                                ; STATEMENT # 25
        0092  8B1E0200          MOV     BX,I
        0096  4B                DEC     BX
        0097  D1E3              SHL     BX,1
        0099  8B870000          MOV     AX,VOLTS[BX]
        009D  8B4E04            MOV     CX,[BP].VOLTIN
        00A0  2BC8              SUB     CX,AX
        00A2  8B360200          MOV     SI,I
        00A6  D1E6              SHL     SI,1
```

Figure 5-3. Listing of PLMDATA with the CODE Control (Cont'd.)

```
          00A8  88971000          MOV     DX,TCEL[BX]
          00AC  8B9C1000          MOV     BX,TCEL[SI]
          00B0  2BDA              SUB     BX,DX
          00B2  50                PUSH    AX      ; 1
          00B3  89C8              MOV     AX,CX
          00B5  52                PUSH    DX      ; 2
          00B6  F7E3              MUL     BX
          00B8  D1E0              SHL     AX,1

          00BA  89060400          MOV     NUMERATOR,AX
  26   2          RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)-VOLTS(I-1))+1, 1);
                                                  ; STATEMENT # 26

          00BE  8B9C0000          MOV     BX,VOLTS[SI]
          00C2  5A                POP     DX      ; 2
          00C3  59                POP     CX      ; 1
          00C4  2BD9              SUB     BX,CX
          00C6  52                PUSH    DX      ; 1
          00C7  31D2              XOR     DX,DX
          00C9  F7F3              DIV     BX
          00CB  40                INC     AX
          00CC  D1E8              SHR     AX,1
          00CE  59                POP     CX      ; 1
          00CF  03C1              ADD     AX,CX
          00D1  5D                POP     BP
          00D2  C20200            RET     2H
  27   2      END INTERPOLATE;
                                                  ; STATEMENT # 27
                        INTERPOLATE       ENDP


  28   1      TEMP$DATA$FROM$PORTS:
                                                  ; STATEMENT # 28
                      TEMPDATAFROMPORTS       PROC NEAR
          00D5  55                PUSH    BP
          00D6  8BEC              MOV     BP,SP
             PROCEDURE(HIGH,LOW) WORD;
  29   2          DECLARE (HIGH,LOW) WORD;
  30   2          DECLARE IN$PORT$HIGH WORD; /* ADDRESS in McCracken's book */
  31   2          DECLARE IN$PORT$LOW BYTE;
  32   2          DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

  33   2          IN$PORT$HIGH = INPUT(HIGH);
                                                  ; STATEMENT # 33
          00D8  8B5606            MOV     DX,[BP].HIGH
          00DB  EC                IN      DX
          00DC  B400              MOV     AH,0H
          00DE  89060600          MOV     INPORTHIGH,AX
  34   2          IN$PORT$LOW = INPUT(LOW);
                                                  ; STATEMENT # 34
          00E2  8B5604            MOV     DX,[BP].LOW
          00E5  EC                IN      DX
          00E6  88061100          MOV     INPORTLOW,AL
  35   2          THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
                                                  ; STATEMENT # 35
          00EA  8B060600          MOV     AX,INPORTHIGH
          00EE  B108              MOV     CL,8H
          00F0  D3E0              SHL     AX,CL
          00F2  8A0E1100          MOV     CL,INPORTLOW
          00F6  B500              MOV     CH,0H
          00F8  0BC1              OR      AX,CX
          00FA  89060800          MOV     THERMOCOUPLEOUTPUT,AX
  36   2          TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
                                                  ; STATEMENT # 36
          00FE  50                PUSH    AX      ; 1
          00FF  E857FF            CALL    INTERPOLATE
          0102  89060A00          MOV     TEMPERATURE,AX
  37   2          RETURN TEMPERATURE;
                                                  ; STATEMENT # 37

          0106  8B060A00          MOV     AX,TEMPERATURE
          010A  5D                POP     BP
          010B  C20400            RET     4H
  38   2      END TEMP$DATA$FROM$PORTS;
                                                  ; STATEMENT # 38
                      TEMPDATAFROMPORTS       ENDP

  39   1      END PLMDATA;
                                                  ; STATEMENT # 39


MODULE INFORMATION:

     CODE AREA SIZE     = 010EH    2700
     CONSTANT AREA SIZE = 0020H     32D
     VARIABLE AREA SIZE = 0012H     18D
     MAXIMUM STACK SIZE = 0010H     16D
     95 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 5-3. Listing of PLMDATA with the CODE Control (Cont'd.)

CHAPTER 6
USING UTILITIES TO PREPARE EXECUTABLE PROGRAMS

"Three things are to be looked to in a building: that it stand on the right spot; that it be securely founded; that it be successfully executed."

—Johann Wolfgang Von Goethe

You must do three things to prepare and execute programs successfully: link program modules to resolve external references, locate the linked modules by binding them to memory addresses, and run the program in the appropriate operating environment.

These things are easy to do for most high-level language programs. Easy-to-use *utility programs* perform these operations for you. They are also flexible enough to allow you to perform more complicated linking and locating operations for programs that refer to physical memory addresses. The compilers for high-level languages usually produce programs that do not refer to physical memory addresses; these programs can be linked and located in one easy step.

The diagram in figure 6-1 shows the process of linking and locating (binding to addresses) modules to prepare a program that can be RUN on the Series III system (or debugged via DEBUG-86, described in Chapter 7).

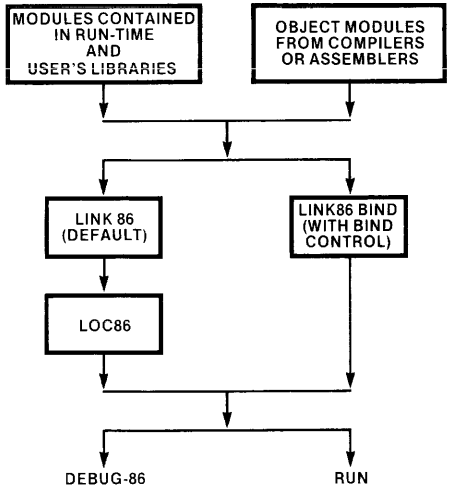


Figure 6-1. Using Utilities to Prepare Executable Programs 121632-6

PREPARING A LIBRARY OF PROGRAM MODULES

A *library* is any collection of public modules—modules that contain public procedures that can be used by programs. Some libraries are supplied by Intel; for example, the *run-time system* used with Pascal-86 programs is supplied as several library files. Pascal-86 has a number of built-in procedures that you can use in any Pascal-86 program—these procedures can be found in the supplied run-time libraries. To use any of the built-in procedures, you must link the run-time libraries to your program modules using the LINK86 utility.

You can also build your own library files using the LIB86 utility. With the LIB86 utility you can create a library, add modules from another library, add new modules, delete modules in a library, and list the names of modules in a library.

The following example shows a session with LIB86. We create a new library called TONY.LIB, and then we ADD to it some of the modules from a supplied run-time library called P86RN2.LIB. We then LIST the modules in TONY.LIB, and EXIT from the LIB86 utility:

```
-RUN LIB86<cr>
SERIES-III 8086 LIBRARIAN V1.0
*CREATE :F1:TONY.LIB<cr>
*ADD :F1:P86RN2.LIB(MOD1, MOD4, MOD7) TO :F1:TONY.LIB<cr>
*LIST :F1:TONY.LIB<cr>
TONY.LIB
     MOD1
     MOD4
     MOD7
*EXIT<cr>
-
```

The linker (LINK86) treats library files in a special way. As shown in the next section, you specify your program modules first in the LINK86 command line, then you specify the appropriate libraries. You must be sure to link these modules in the proper sequence.

Why? Remember that your program's main module refers to procedures that exist only in other modules—*external procedures*. The linker must be able to find the external procedures. LINK86 remembers the references to external procedures in the first module, and looks in the subsequent modules for those external procedures. If it cannot find the external procedures in subsequent modules (maybe because you erroneously specified the library *before* specifying the program modules in the LINK86 command line), LINK86 will generate an error message.

The built-in procedures supplied with Pascal-86 (READLN, WRITELN, etc.) are external procedures contained in the modules included with the run-time support libraries. LINK86 will first see the reference to these procedures in your program modules, and then it will look in the libraries for the modules that will satisfy those references. LINK86 will only link in those modules that are needed to satisfy external references; it will not link in the entire library of modules unless your program modules refer to all of the library modules.

Since you do not need to use LIB86 to handle run-time libraries supplied with Pascal-86, you only need LIB86 to handle your own libraries. Why would you set up your own libraries? To manage sets of repetitive modules. In many software development labs, modules useful to many different programs would either be lost or repeated. Libraries are sets of modules that are easily maintained through use of the LIB86 librarian. The LINK86 utility is capable of searching such a library and only checking out the modules needed for the linked program.

LINKING MODULES TO FORM A LOCATABLE PROGRAM

Most modular programs have a main module that calls procedures in other subordinate modules. Although a subordinate module can call a procedure in the main module, most calls are from the main module to a subordinate module, and the modular structure resembles an upside-down tree, as shown in figure 6-2. We included the run-time system modules in this tree, since the program modules rely on the built-in procedures and operating environment calls found in the run-time system.

When you link these modules together using the LINK86 utility, you allow LINK86 to see the main module first, because the main module is the most abstract; that is, it has the highest level of abstraction, and it calls procedures in lower levels to perform each activity. You should then allow LINK86 to see the next subordinate level of modules, and so on.

The *last* group of modules for LINK86 should be any run-time system libraries that are needed to perform the built-in procedures (READLN, WRITELN, etc.). The run-time system libraries contain modules that are at the lowest level of abstraction—these are the modules that call procedures in the operating environment of your system (the operating environment is usually invisible to you, but not to your Pascal program).

For an example, we will link together the modules needed to test our main program in the Series III environment. We start with our main module MAIN.OBJ, which holds the **MainControl** module. We link to it the test versions of the modules **GetData** found in DUMDAT.OBJ, and **Operation** found in DUMOP.OBJ. Finally, we link in the modules we need from the run-time system libraries P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB, and LARGE.LIB (we explain these libraries after the example):

```
-RUN LINK86 :F1:MAIN.OBJ,:F1:DUMDAT.OBJ,:F1:DUMOP.OBJ,&<cr>
>>:F1:P86RN0.LIB,:F1:P86RN1.LIB,:F1:P86RN2.LIB,&<cr>
>>:F1:P86RN 3.LIB,:F1:87NULL.LIB,:F1:LARGE.LIB&<cr>
>>TO :F1:PROGRM.86 BIND<cr>
```

Let's explain this example. We used the RUN command to run the LINK86.86 utility in the "8086 side" of the Series III. We specified the three object modules of our program, and then we specified all of the run-time libraries needed to run our program in the Series III environment.

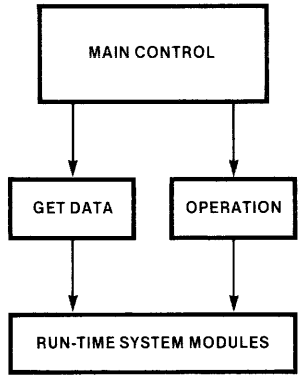


Figure 6-2. Main Module with Subordinate Modules 121632-7

We directed the output to a file called PROGRM.86. Finally, we specified the BIND control in order to make PROGRM.86 an LTL (load-time locatable) program. The BIND control is the easiest way to make a program locatable in the Series III environment (more on BIND in the next section). PROGRM.86 can now be loaded and executed (via the RUN command) in the Series III system.

The run-time system libraries P86RN0.LIB and P86RN1.LIB are required for any Pascal-86 program that needs run-time support. The libraries P86RN2.LIB and P86RN3.LIB are required for any Pascal-86 program that uses the file input/output procedures and other operations performed by the operating system. The 87NULL.LIB is needed for programs that do *not* use either the 8087 processor or emulator to perform real arithmetic. Since our program has no REAL data types, it does not perform real arithmetic; therefore, it needs the 87NULL.LIB library (if it did perform real arithmetic, it would need other libraries). Finally, to run any Pascal-86 program on the Series III system, you need to link the LARGE.LIB library to the program. The LARGE.LIB library contains the primitives (service routines) used to perform operations in the Series III environment.

The run-time system is separated into several libraries so that you can customize your run-time environment if you so wish. The libraries described above are the default libraries used to run programs on the Series III system, if your programs do not use REAL data types. If your programs use REAL data types, you would not use 87NULL.LIB; instead, you would use the 8087 processor with the library 8087.LIB, or the 8087 software emulator with the libraries E8087 and E8087.LIB. Consult the *Pascal-86 User's Guide* for specific information about the run-time system libraries.

LOCATING AND RUNNING PROGRAMS

A program must reside in actual memory before it can run. The *locating* process assigns actual (physical) memory addresses to a program. There are two ways to accomplish the locating process:

1. Using LINK86 with the BIND control to produce an LTL module (discussed in the next paragraph), which can be located, loaded, and executed automatically by the RUN command
2. Using LINK86 (without the BIND control) to produce a linked module, then LOC86 to locate the module in an area of memory you specify, and finally RUN to load and execute the program

The simplest locating operation involves using the LINK86 utility with the BIND control, as shown in the previous example. This simpler process, called *binding*, binds modules to logical segments, which can be located in actual memory by the RUN command in one fast step. Modules produced by the LINK86 utility with the BIND control are called *load-time locatable* modules (LTL modules). An LTL module is a module that can be located almost anywhere in memory, and so the RUN loader can easily locate it in the Series III environment for you. The DEBUG-86 debugger can also locate an LTL module for you, as we will show in the next chapter.

For example, our LINK86 example in the previous section bound the modules properly to form the program PROGRM.86. Now, in one step, you can locate this bound program in actual memory, load it into memory, and execute it in the "8086 side" (8086 execution mode) of the Series III system by using the RUN command:

```
-RUN :F1:PROGRM<cr>
```

Note that we did not type PROGRM.86, only PROGRM. The RUN loader looks for the ".86" extension automatically, unless you specify another extension or a period at the end of the name (the period signifies no extension).

High-level language programmers usually do not burden themselves with more details about locating programs. However, assembly language programs and some PL/M programs frequently refer to physical addresses rather than symbolic (logical) addresses. These program modules are called *absolute* modules because they use absolute physical addresses. Absolute modules cannot be located automatically by the RUN command—they must be relocated first by the LOC86 utility.

There is a case when even the simplest program must be located by LOC86: if you intend to debug your program using an ICE-86 or ICE-88 emulator, you must locate the program with LOC86 to make it an absolute module. The ICE (In-Circuit Emulation) loaders can only load absolute modules.

For an example, we will link our new PL/M-86 module PLMDAT.OBJ to our program, along with another version of our **GetData** module in DATA.OBJ, and produce the linked module MAIN.LNK:

```
-RUN LINK86 :F1:MAIN.OBJ,:F1:DATA.OBJ,:F1:PLMDAT.OBJ,&<cr>
>>:F1:DUMOP.OBJ,:F1:P86RN0.LIB,:F1:P86RN1.LIB,:F1:P86RN2.LIB,&< cr>
>>:F1:P86RN3.LIB,:F1:87NULL.LIB,:F1:LARGE.LIB<cr>
```

Since we did not specify a new filename with a "TO" clause, the LINK86 utility directed the linked output to the file :F1:MAIN.LNK (LINK86 takes the name of the first object module MAIN.OBJ, and changes its extension to LNK to make MAIN.LNK). Now we are ready to locate :F1:MAIN.LNK with the LOC86 utility:

```
-RUN LOC86 :F1:MAIN.LNK TO :F1:PROGRM.86 RESERVE(200H TO 77FFH)<cr>
```

We used the RESERVE control with LOC86 to reserve an area of memory for the Series III operating system. LOC86 will not locate any program segments in the area between addresses 200H and 77FFH ("H" is for hexadecimal). You must leave room for the Series III operating system to execute programs in the Series III environment.

The LINK86, LOC86, and LIB86 utilities are described in detail in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems*. This manual describes all of the utilities for iAPX 86 or iAPX 88 applications development. These utilities are designed to be used in 8086-based development systems like the Series III.

We can now RUN our program on the Series III system:

```
-RUN PROGRM<cr>

Climate system is now on.
-------------------------
Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:21<cr>

Type the temperature of the collector water in Celsius:50<cr>

Type the temperature of the tank water in Celsius:60<cr>
```

```
Type the temperature of the heated tank water in Celsius:70<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:25<cr>

Type the amount of sunlight, any integer will do for now:1<cr>

---------------------------
CURRENT DATA IS AS FOLLOWS:
---------------------------
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1

===============================================
The Climate System is now operating.

The time is 13:25
The inside temperature is 21C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector to Exchanger
===============================================

:::::::::::::::::::::::::::::::::::::::::::::::::
No panic occurred, ABNORMAL shutdown.
:::::::::::::::::::::::::::::::::::::::::::::::::
Last chosen heating method was: Solar Collector to Exchanger
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Goodnight, Irene...
-
```

What happened!? Our program stopped with an abnormal shutdown! It had no trouble picking the right heating method; however, something caused the variable **Operating** to become FALSE, without setting **Panic** to TRUE. We will have to debug the program in the next chapter...

CHAPTER 7
DEBUGGING AND EXECUTING PROGRAMS

"Appearances are often deceiving." —Aesop

We have written program modules and they compiled correctly. We used the LINK86 utility with BIND to link the modules into a program and prepare it for execution. By all appearances, it looks like it will work as planned.

However, when we loaded and executed (via the RUN command) the program, something mysterious happened—the "hidden glitch" struck!

We now have to *debug* our program—find all of the "hidden glitches" and fix them. Debugging is so essential to programming that in most software development efforts, more time is alotted to debugging than to any other activity.

Debuggers are programs that monitor a program's execution and allow you to stop execution and check details. Usually a debugger is part of an *execution vehicle*—a piece of hardware your program executes on, or a piece of software your program executes in. The debugger monitors the activity in the execution environment.

For example, the Series III provides two execution environments or execution vehicles: the 8085 execution mode and the 8086 execution mode. To debug programs that run in the 8085 execution mode, you type the DEBUG command on the "8085 side" (8085 execution mode) to start the Monitor. To debug programs that run in the 8086 execution mode, you RUN the DEBUG command on the "8086 side" (8086 execution mode) to start the DEBUG-86 debugger. Both debuggers reside in ROM.

With a debugger you can load and execute a program (rather than using RUN), and stop the execution to check the values of variables, the contents of registers, and other details. You can also change such values, and perform other activities that monitor a program's execution. You can even execute your program one step at a time.

Since the Monitor (for 8085 execution mode) is described in several Intel documents (McCracken's *Guide to Intellec Microcomputer Development Systems* and the *Intellec Series III Microcomputer Development System Console Operating Instructions*), we will devote this chapter to describing DEBUG-86 and the ICE-88 emulator (a version of the ICE-86 emulator) for debugging programs in the 8086 execution mode for iAPX 86, 88 applications.

USING DEBUG-86 FOR SYMBOLIC DEBUGGING

Debuggers are useful because they load and execute a program for you, and they allow you to stop execution at any point you specify. If you are an assembly language programmer, your program directly uses registers and memory locations; you would want to check the contents of these registers and memory locations. If you are a high-level language programmer (PL/M, Pascal, FORTRAN, etc.), you do not directly refer to registers and memory locations; you would rather check the values of your variables, or *symbols*.

For example, we know we have a problem with our program: the variable **Operating** should always have the boolean value of TRUE as long as **Panic** is FALSE. If **Panic** is set to TRUE, **Operating** should then become FALSE; however, **Operating** is somehow set to FALSE while **Panic** remains FALSE. During the execution of our program, we should be able to stop execution and check the value of **Operating**.

One way to do this is to find the address of **Operating** and check its contents. A better way would be to simply ask for the value of **Operating** in a command such as:

```
*BOOLEAN BYTE ..MAINCONTROL.OPERATING<cr>
FALSE
```

In DEBUG-86, you can execute our program and stop its execution anywhere. You can then use the above command to find the boolean (true or false) value of **Operating** without knowing where **Operating** is located.

In our program there is only one symbol named **Operating**. If there were more than one, we would specify the symbol by specifying both the module name and the symbol name (e.g., ..MAINCONTROL.OPERATING), as we did in the above example. We would have obtained the same result by specifying only ".OPERATING". You use two periods as a prefix to module names, and one period as a prefix to symbol names.

To do symbolic debugging using the symbols in your program, you must compile your program using the DEBUG compiler control (see Chapter 4 for the Pascal-86 compiler, and Chapter 5 for the PL/M-86 compiler). The DEBUG compiler control produces a symbol table for your program, which can be loaded by DEBUG-86. If you do not use the DEBUG compiler control while compiling, you can still do symbolic debugging by defining all of your symbols from within DEBUG-86.

To invoke DEBUG-86, use the following command:

```
-RUN DEBUG<cr>
DEBUG 8086, V1.0
*
```

DEBUG-86 is now in control, as shown by the asterisk (*) prompt. You can now type DEBUG-86 commands. To load PROGRM.86 and its symbol table, type the following command:

```
*LOAD :F1:PROGRM.86<cr>
```

You can check the symbol table by typing the SYMBOLS command:

```
*SYMBOLS<cr>
.
.
.
```

The symbol table contains each module name and symbol name, with their addresses. You can find the address of a single symbol by typing the name of the symbol. For example, we want to know the address of the call (in module **MainControl**) to the procedure **OperateSystem** (which is in the module **Operation**):

```
*..MAINCONTROL.OPERATESYSTEM<cr>
..MAINCONTROL.OPERATESYSTEM=0481:06CCH
```

DEBUG-86 knows where to start executing the program by looking at the contents of the CS and IP (CS:IP) registers. The CS register holds the starting address of the program's code segment, and the IP register holds the address within the code segment where the program should start execution. CS and IP change as the program executes; therefore, to preserve the starting location of the program, we create a new symbol called .START to hold the starting address:

```
*DEFINE .START=CS:IP<cr>
```

We could also check the contents of CS:IP:

```
*CS<cr>
CS=0481H
*IP<cr>
IP=0FD6H
```

You should familiarize yourself with addresses. The 8086 and 8088 processors use 20-bit physical addresses separated into two words: the segment base address and the offset value. The CS register holds the code segment base address, and the IP register holds the offset value. All addresses are in hexadecimal notation (the "H" stands for hexadecimal). All addresses are displayed with the segment base, followed by a colon, followed by the offset.

The EVALUATE (abbreviated "EVA") command is useful for evaluating numeric and character values and addresses. For example, the number 4142H can be represented in several ways:

```
*EVA 4142<cr>
1000001010000Y  40502Q   16706T  4142H  'AB'
```

When you type a number by itself, DEBUG-86 assumes the number is in hexadecimal notation. The number 4142H is equivalent to the decimal number 16706 ("T" denotes decimal notation), the octal number 40502Q ("Q" denotes octal notation), the binary number 1000001010000Y ("Y" denotes binary notation), and the ASCII characters "AB" (41H is the ASCII code for "A" and 42H is the ASCII code for "B").

The EVA command will also find the closest symbol that has the address you specify. The keyword SYM tells the EVA command to evaluate the address symbolically:

```
*EVA 481:6CC SYM<cr>
..MAINCONTROL.OPERATESYSTEM
```

We will now execute our program and stop its execution before it executes the **OperateSystem** procedure. The GO command will start executing at the beginning of the program, which it knows by looking at the CS and IP registers (CS:IP). You could specify an actual address with GO to start from, or the line number of a program statement, or a statement label; however, we do not use statement labels in our program. We want our program to stop while it is executing the main (WHILE **Operating**) loop, *not* at the start of the **OperateSystem** proced: e in the

Operation module (since procedure definitions do not actually execute). Our program will GO until it reaches the starting address, in the **MainControl** module, of the call to the **OperateSystem** procedure:

```
*GO TILL ..MAINCONTROL.OPERATESYSTEM<cr>
Climate System is now on.
-------------------------
.
.                    During execution, we type in the
.                    temperatures and other data. The program
                     stops and DEBUG-86 displays the next
                     instruction to be executed (in assembly
                     language form).

0481:06CCH          PUSH        BP
**
```

At this point, we can check the boolean values of both **Panic** and **Operating**:

```
**BOOL BYTE .PANIC<cr>
FALSE
*BOOL BYTE .OPERATING<cr>
FALSE
```

To make sure the program stopped at the right place, we evaluate the CS and IP registers symbolically:

```
*EVA CS:IP SYM<cr>
..MAINCONTROL.OPERATESYSTEM
```

Another useful DEBUG-86 command is the STEP command. From any point of program execution, you can execute the program step by step—one machine instruction at a time. The STEP command also displays the next machine instruction to be executed. This display is in "disassembled" form—the machine instruction is translated back into assembly language:

```
*STEP<cr>
0481:06CDH                MOV       BP,SP
*
```

STEP executes one machine instruction and displays the next one disassembled. Another STEP would execute the displayed instruction.

We will now change the value of **Operating** to be TRUE. A boolean variable is TRUE if its numeric value is odd; FALSE if its numeric value is even. Why? Because a boolean is TRUE if its rightmost bit (in a binary representation) is 1, and FALSE if its rightmost bit is 0. All even values written in binary form end with a 0 in the rightmost bit, and all odd values written in binary form end with a 1.

To change the value of **Operating**, we must also specify BYTE, since it is only one byte long:

```
*BYTE .OPERATING=1<cr>
*BOOL BYTE .OPERATING<cr>
TRUE
```

We can now continue execution with a GO command:

```
*GO<cr>
================================================
The Climate System is now operating.
.
.
.
```

Execution continues correctly, and the program loops back to the **GetData** *procedure to obtain more data. As we continue execution, the program once again erroneously changes* **Operating** *to FALSE.*

We can use our .START symbol to start execution once again at the beginning of the program:

```
*GO FROM .START TILL ..MAINCONTROL.OPERATESYSTEM<cr>
```

After repeating these debugging operations, we are sure that our error occurred at the statement where **Operating** is set to FALSE while **Panic** is set to TRUE.

Refer to the listing in Chapter 4. The Pascal-86 statement numbers are in the "STMT" column of the listing. The "LINE" column shows the source file line numbers (you can use the line numbers with CREDIT to display, edit, move, or copy particular lines).

Between statements 37 and 40 (source lines 68 and 70), we have an ELSE clause that should only execute if the temperatures are not greater than the minimums necessary to heat the building. **Panic** should be set to TRUE and **Operating** should be set to FALSE, if this ELSE clause executes.

However, the "ELSE **Panic**:=TRUE" is not executing, but the "**Operating**:=FALSE" is *always* executing! The error is one of omission: to have two statements execute as part of an ELSE clause, they must begin with BEGIN and end with END, as shown:

```
ELSE BEGIN
       Panic:=TRUE; Operating:=FALSE;
     END;
```

To make this change, we must re-edit MAIN.SRC, re-compile MAIN.SRC to obtain a new MAIN.OBJ, and re-link the modules using LINK86 (with BIND, as shown in Chapter 6). Figure 7-1 shows the revised listing, with the corrected ELSE clause shaded, and a sample run of the program.

```
SERIES-III Pascal-86, x031                                  09/01/80        PAGE   1
                                                                            MAINCONTROL


Source File: :F1:MAIN.SRC
Object File: :F1:MAIN.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:MAIN.SRC
   1    1  0  0         MODULE MainControl;
   2    2  0  0
                       (* Interface specification common to all modules *)

                       $INCLUDE(:F1:INSPEC.SRC)
              =1       PUBLIC MainControl; (*section of interface specification*)
```

Figure 7-1. Climate Control Program Listing and Sample Run

```
 3    2  0  0 =1
                =1          CONST (*declarations declared publicly in this module*)
                =1
                =1            MinimumForExchanger   =  35;(*degrees Celsius*)
 4    6  0  0 =1            MinimumForHeatPump    =  13;
 5    7  0  0 =1
                =1          TYPE (*definitions publicly defined in this module*)
                =1
                =1            AirTemperature   =-20..120;(*degrees in Celsius*)
 6   11  0  0 =1            WaterTemperature =0..120;
 7   12  0  0 =1            HeatingMethods   =(CollectorToExchanger,
                =1                               CollectorToHeatPump,
                =1                               TankToExchanger,
                =1                               TankToHeatPump,
                =1                               HeatedTankToHeatPump,
                =1                               NoMethod);
 8   18  0  0 =1
                =1            SystemData       = RECORD
 8   20  0  1 =1                                   ChosenMethod          : HeatingMethods;
 9   21  0  1 =1                                   InsideTemp,
                =1                                 ThermostatSetting     : AirTemperature;
10   23  0  1 =1                                   CollectorWaterTemp,
                =1                                 TankWaterTemp,
                =1                                 HeatedTankTemp        : WaterTemperature;
11   26  0  1 =1                                   AmountOfSunlight      : Integer;
12   27  0  1 =1                                   Hour                  : 00..24;
13   28  0  1 =1                                   Minute                : 00..59;
14   29  0  1 =1                                 END (*SystemData*);
15   30  0  0 =1
                =1          VAR (*variables publicly defined in this module.*)
                =1
                =1            CurrentData                 : SystemData;
16   34  0  0 =1            Operating, Panic            : BOOLEAN;
17   35  0  0 =1
                =1        PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
18   37  0  0 =1
                =1           PROCEDURE GetData(VAR CurrentData:SystemData);
19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
20   40  0  0 =1
                =1        PUBLIC Operation; (*Operation Module containing OperateSystem,
21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                =1           PROCEDURE StartUpSystem;
22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
24   46  0  0 =1
                          PROGRAM MainControl(INPUT,OUTPUT);
25    8  0  0
                          (* end of interface specification *)


                          PROCEDURE DetermineMethod(VAR CurrentData : SystemData);
26   13  1  0                BEGIN
26   14  1  1                  WITH CurrentData DO
27   15  1  1                        BEGIN
27   16  1  2                         IF InsideTemp<ThermostatSetting THEN
28   17  1  2                            BEGIN
28   18  1  3                               IF CollectorWaterTemp>MinimumForExchanger THEN
29   19  1  3                                  ChosenMethod:=CollectorToExchanger
                                          ELSE IF CollectorWaterTemp>MinimumForHeatPump THEN
31   21  1  3                                  ChosenMethod:=CollectorToHeatPump
                                          ELSE IF TankWaterTemp>MinimumForExchanger THEN
33   23  1  3                                  ChosenMethod:=TankToExchanger
                                          ELSE IF TankWaterTemp>MinimumForHeatPump THEN
35   25  1  3                                  ChosenMethod:=TankToHeatPump
                                          ELSE IF HeatedTankTemp>MinimumForHeatPump THEN
37   27  1  3                                  ChosenMethod:=HeatedTankToHeatPump
                                          ELSE BEGIN
38   29  1  4                                  Panic:=TRUE; Operating:=FALSE;
40   30  1  4                                  END
41   31  1  3                            END
42   32  1  2                        ELSE (*no heating request*)ChosenMethod:=NoMethod
43   33  1  2                    END; (*With CurrentData*)
45   34  1  1                END; (*DetermineMethod)
46   35  0  0
                          (************** MAIN PROGRAM ***********************************)

                          BEGIN
46   39  0  1                StartUpSystem;
47   40  0  1                Operating:=TRUE;
48   41  0  1                Panic:=FALSE;
49   42  0  1                WHILE Operating DO (*while system is operating, do:*)
50   43  0  1                   BEGIN
50   44  0  2                      GetData(CurrentData); (*Get the temps, time, etc.*)
51   45  0  2                      StoreData(CurrentData); (*Store this data as record*)
52   46  0  2                      DetermineMethod(CurrentData); (*this detects a panic*)
53   47  0  2                      OperateSystem(CurrentData);
54   48  0  2                   END; (*while operating*)
56   49  0  1                ShutDownSystem(CurrentData);
57   50  0  1          END. (*Main Control Algorithm*)
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
Summary Information:

PROCEDURE                          OFFSET    CODE SIZE      DATA SIZE    STACK SIZE
DETERMINEMETHOD                    0011H  003FH   143D                  0006H    6D

Total                                     0147H   327D  0016H    22D  004CH   76D

  96 Lines Read.
   0 Errors Detected.
 33% Utilization of Memory.



SERIES-III Pascal-86, X031                                          09/01/80        PAGE  1


Source File: :F1:DUMDAT.SRC
Object File: :F1:DUMDAT.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMDAT.SRC
   1    1  0  0         (*This is a dummy GetData module, with dummy GetData
                        and StoreData procedures, for use with MainControl
                        module in testing phases.  It only performs console
                        input to get Celsius temperatures, the time of day,
                        and the amount of sunlight (insolation) for the
                        solar collector.  Use PLM86$DATA module for real
                        application.*)

                        MODULE GetData;
   2   10  0  0
                        (* Interface specification common to all modules *)

                        $INCLUDE(:F1:INSPEC.SRC)
                    =1  PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                    =1     CONST (*declarations declared publicly in this module*)
                    =1
                    =1       MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1           MinimumForHeatPump    =  13;
   5    7  0  0 =1
                    =1     TYPE (*definitions publicly defined in this module*)
                    =1
                    =1       AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1           WaterTemperature =0..120;
   7   12  0  0 =1           HeatingMethods   =(CollectorToExchanger,
                    =1                         CollectorToHeatPump,
                    =1                         TankToExchanger,
                    =1                         TankToHeatPump,
                    =1                         HeatedTankToHeatPump,
                    =1                         NoMethod);
   8   18  0  0 =1
                    =1       SystemData       = RECORD
   8   20  0  1 =1                               ChosenMethod          : HeatingMethods;
   9   21  0  1 =1                               InsideTemp,
                    =1                         ThermostatSetting     : AirTemperature;
  10   23  0  1 =1                               CollectorWaterTemp,
                    =1                         TankWaterTemp,
                    =1                         HeatedTankTemp        : WaterTemperature;
  11   26  0  1 =1                               AmountOfSunlight      : Integer;
  12   27  0  1 =1                               Hour                  : 00..24;
  13   28  0  1 =1                               Minute                : 00..59;
  14   29  0  1 =1                             END (*SystemData*);
  15   30  0  0 =1
                    =1     VAR (*variables publicly defined in this module.*)
                    =1
                    =1       CurrentData                : SystemData;
  16   34  0  0 =1           Operating, Panic           : BOOLEAN;
  17   35  0  0 =1
                    =1   PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
  18   37  0  0 =1
                    =1      PROCEDURE GetData(VAR CurrentData:SystemData);
  19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
  20   40  0  0 =1
                    =1   PUBLIC Operation; (*Operation Module containing OperateSystem,
  21   42  0  0 =1                           StartUpSystem and ShutDownSystem*)
                    =1      PROCEDURE StartUpSystem;
  22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
  23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
  24   46  0  0 =1


                        PRIVATE GetData;
  25   16  0  0
                        (* end of interface specification *)
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
                                PROCEDURE GetData(VAR CurrentData:SystemData);
   26    21  1  0               BEGIN
   26    22  1  1                  WITH CurrentData DO BEGIN
   27    23  1  2                     WRITE('Type the thermostat setting in degrees Celsius:');
   28    24  1  2                     READLN(ThermostatSetting); WRITELN;
   30    25  1  2                     WRITE('Type the inside temperature reading in Celsius:');
   31    26  1  2                     READLN(InsideTemp); WRITELN;
   33    27  1  2                     WRITE('Type the temperature of the collector water in Celsius:');
   34    28  1  2                     READLN(CollectorWaterTemp); WRITELN;
   36    29  1  2                     WRITE('Type the temperature of the tank water in Celsius:');
   37    30  1  2                     READLN(TankWaterTemp); WRITELN;
   39    31  1  2                     WRITE('Type the temperature of the heated tank water in Celsius:');
   40    32  1  2                     READLN(HeatedTankTemp); WRITELN;
   42    33  1  2                     WRITE('Type the hour of day, as in 04 or 24:');
   43    34  1  2                     READLN(Hour); WRITELN;
   45    35  1  2                     WRITE('Type the minute of the hour, as in 01 or 59: ');
   46    36  1  2                     READLN(Minute); WRITELN;
   48    37  1  2                     WRITE('Type the amount of sunlight, any integer will do for now:');
   49    38  1  2                     READLN(AmountOfSunlight); WRITELN;
   51    39  1  2                  END;  (*with CurrentData*)
   53    40  1  1               END;
   54    41  0  0
                                PROCEDURE StoreData(VAR CurrentData:SystemData);
   55    43  1  0               BEGIN
   55    44  1  1               (*Dummy procedure, eventually will store CurrentData in a file*)
                                   WITH CurrentData DO BEGIN
   56    46  1  2                     WRITELN('----------------------------');
   57    47  1  2                     WRITELN('CURRENT DATA IS AS FOLLOWS:');
   58    48  1  2                     WRITELN('----------------------------');
   59    49  1  2                     WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
   60    50  1  2                     WRITELN('Inside temperature is ',InsideTemp,'C');
   61    51  1  2                     WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
   62    52  1  2                     WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
   63    53  1  2                     WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
   64    54  1  2                     WRITELN('Time of day is ',Hour,':',Minute);
   65    55  1  2                     WRITELN('Amount of sunlight is ',AmountOfSunlight);
   66    56  1  2                     WRITELN; (*a blank line*)
   67    57  1  2                  END; (*with CurrentData*)
   69    58  1  1               END
***WARNING, input: "END "
***was repaired to "END ; "
   70    58  0  0

Summary Information:

PROCEDURE                       OFFSET    CODE SIZE     DATA SIZE     STACK SIZE
STOREDATA                        0405H  0222H   546D                  0010H    16D
GETDATA                          0294H  0241H   577D                  0010H    16D

Total                                   06F7H  1783D  0000H      0D  0020H    32D

 104 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.


SERIES-III Pascal-86, X031                                        09/01/80          PAGE   1


Source File: :F1:DUMOP.SRC
Object File: :F1:DUMOP.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: :F1:DUMOP.SRC
   1    1  0  0         (*This is a dummy Operation module, with dummy StartUpSystem,
                       ShutDownSystem, and OperateSystem procedures,
                       for use with MainControl module in testing phases.
                       *)

                       MODULE Operation;
   2    7  0  0
                       (* Interface specification common to all modules *)

                       $INCLUDE(:F1:INSPEC.SRC)
                 =1    PUBLIC MainControl; (*section of interface specification*)
   3    2  0  0 =1
                 =1       CONST (*declarations declared publicly in this module*)
                 =1
                 =1         MinimumForExchanger   =  35;(*degrees Celsius*)
   4    6  0  0 =1          MinimumForHeatPump    =  13;
   5    7  0  0 =1
                 =1       TYPE (*definitions publicly defined in this module*)
                 =1
                 =1         AirTemperature   =-20..120;(*degrees in Celsius*)
   6   11  0  0 =1          WaterTemperature =0..120;
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
 7   12  0  0 =1          HeatingMethods   =(CollectorToExchanger,
                =1                            CollectorToHeatPump,
                =1                            TankToExchanger,
                =1                            TankToHeatPump,
                =1                            HeatedTankToHeatPump,
                =1                            NoMethod);
 8   18  0  0 =1
                =1          SystemData       = RECORD
 8   20  0  1 =1                               ChosenMethod          : HeatingMethods;
 9   21  0  1 =1                               InsideTemp,
                =1                            ThermostatSetting     : AirTemperature;
10   23  0  1 =1                               CollectorWaterTemp,
                =1                            TankWaterTemp,
                =1                            HeatedTankTemp        : WaterTemperature;
11   26  0  1 =1                               AmountOfSunlight      : Integer;
12   27  0  1 =1                               Hour                  : 00..24;
13   28  0  1 =1                               Minute                : 00..59;
14   29  0  1 =1                             END (*SystemData*);
15   30  0  0 =1
                =1        VAR (*variables publicly defined in this module.*)
                =1
                =1          CurrentData                 : SystemData;
16   34  0  0 =1            Operating, Panic            : BOOLEAN;
17   35  0  0 =1
                =1      PUBLIC GetData; (*GetData Module containing GetData & StoreData*)
18   37  0  0 =1
                =1        PROCEDURE GetData(VAR CurrentData:SystemData);
19   39  0  0 =1           PROCEDURE StoreData(VAR CurrentData:SystemData);
20   40  0  0 =1
                =1      PUBLIC Operation; (*Operation Module containing OperateSystem,
21   42  0  0 =1                          StartUpSystem and ShutDownSystem*)
                =1        PROCEDURE StartUpSystem;
22   44  0  0 =1           PROCEDURE OperateSystem(VAR CurrentData:SystemData);
23   45  0  0 =1           PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
24   46  0  0 =1


                        PRIVATE Operation;
25   13  0  0
                        (* end of interface specification *)


                        PROCEDURE StartUpSystem;
26   18  1  0           BEGIN
26   19  1  1                WRITELN ('Climate system is now on.');
27   20  1  1                WRITELN ('--------------------------');
28   21  1  1                WRITELN;
29   22  1  1           END;
30   23  0  0


                        PROCEDURE OperateSystem(VAR CurrentData:SystemData);
31   26  1  0           BEGIN
31   27  1  1             WITH CurrentData DO BEGIN
32   28  1  2               WRITELN('================================================');
33   29  1  2               WRITELN('The Climate System is now operating.');
34   30  1  2               WRITELN;
35   31  1  2               WRITELN('The Time is ',Hour,':',Minute);
36   32  1  2               WRITELN('The inside temperature is ',InsideTemp,'C');
37   33  1  2               WRITELN('The thermostat setting is ',ThermostatSetting,'C');
38   34  1  2               WRITE('Method chosen to heat the building: ');
39   35  1  2               CASE ChosenMethod OF
40   36  1  3                  CollectorToExchanger: WRITE('Solar Collector to Exchanger');
41   37  1  3                  CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
42   38  1  3                  TankToExchanger     : WRITE('Tank to Exchanger');
43   39  1  3                  TankToHeatPump      : WRITE('Tank to Heat Pump');
44   40  1  3                  HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
45   41  1  3                  NoMethod            : WRITE('No heat required');
46   42  1  3               END;
48   43  1  2               WRITELN;
49   44  1  2               WRITELN('================================================');
50   45  1  2               WRITELN; (*write a blank line.*)
51   46  1  2            END;
53   47  1  1           END;(*OperateSystem*)
54   48  0  0


                        PROCEDURE ShutDownSystem(VAR CurrentData:SystemData);
55   51  1  0           BEGIN
55   52  1  1           WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
56   53  1  1              IF Panic THEN WRITELN('PANIC occurred, NORMAL shutdown.')
57   54  1  1                       ELSE WRITELN('No panic occurred, ABNORMAL shutdown.');
59   55  1  1              WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
60   56  1  1              WITH CurrentData DO BEGIN
61   57  1  2                 WRITE('Last chosen heating method was: ');
62   58  1  2                 CASE ChosenMethod OF
63   59  1  3                    CollectorToExchanger: WRITE('Solar Collector to Exchanger');
64   60  1  3                    CollectorToHeatPump : WRITE('Solar Collector to Heat Pump');
65   61  1  3                    TankToExchanger     : WRITE('Tank to Exchanger');
66   62  1  3                    TankToHeatPump      : WRITE('Tank to Heat Pump');
67   63  1  3                    HeatedTankToHeatPump: WRITE('Heated Tank to Heat Pump');
68   64  1  3                    NoMethod            : WRITE('No heat required');
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
  69   65  1  3              END;
  71   66  1  2              WRITELN;
  72   67  1  2              WRITELN('Thermostat Setting is ',ThermostatSetting,'C');
  73   68  1  2              WRITELN('Inside temperature is ',InsideTemp,'C');
  74   69  1  2              WRITELN('Temperature of collector water is ',CollectorWaterTemp,'C');
  75   70  1  2              WRITELN('Temperature of tank water is ',TankWaterTemp,'C');
  76   71  1  2              WRITELN('Temperature of the heated tank water is ',HeatedTankTemp,'C');
  77   72  1  2              WRITELN('Time of day is ',Hour,':',Minute);
  78   73  1  2              WRITELN('Amount of sunlight is ',AmountOfSunlight);
  79   74  1  2            END;(*with CurrentData*)
  81   75  1  1            WRITELN('::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::');
  82   76  1  1            WRITELN;
  83   77  1  1            WRITELN('Goodnight, Irene...');
  84   78  1  1          END
***WARNING, input: "END "
***was repaired to "END ; "
  85   78  0  0            .(*ShutDownSystem*)


Summary Information:

PROCEDURE                      OFFSET   CODE SIZE     DATA SIZE    STACK SIZE
SHUTDOWNSYSTEM                  0698H  0398H   920D                 0010H   16D
OPERATESYSTEM                   0440H  0258H   600D                 0010H   16D
STARTUPSYSTEM                   03E2H  005EH    94D                 0010H   16D

Total                                  0A30H  2608D  0000H     0D  0030H   48D

 125 Lines Read.
   1 Error Detected.
 33% Utilization of Memory.
```

```
-RUN PROGRM<cr>

Climate system is now on.
-------------------------
Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:21<cr>

Type the temperature of the collector water in Celsius:60<cr>

Type the temperature of the tank water in Celsius:60<cr>

Type the temperature of the heated tank water in Celsius:70<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:25<cr>

Type the amount of sunlight, any integer will do for now:1<cr>

---------------------------
CURRENT DATA IS AS FOLLOWS:
---------------------------
Thermostat Setting is 24C
Inside temperature is 21C
Temperature of collector water is 60C
Temperature of tank water is 60C
Temperature of the heated tank water is 70C
Time of day is 13:25
Amount of sunlight is 1
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

```
================================================
The Climate System is now operating.

The time is 13:25
The inside temperature is 21C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector to Exchanger
================================================

Type the thermostat setting in degrees Celsius:24<cr>

Type the inside temperature reading in Celsius:22<cr>

Type the temperature of the collector water in Celsius:10<cr>

Type the temperature of the tank water in Celsius:10<cr>

Type the temperature of the heated tank water in Celsius:10<cr>

Type the hour of day, as in 04 or 24:13<cr>

Type the minute of the hour, as in 01 or 59:30<cr>

Type the amount of sunlight, any integer will do for now:0<cr>
---------------------------
CURRENT DATA IS AS FOLLOWS:
---------------------------
Thermostat Setting is 24C
Inside temperature is 22C
Temperature of collector water is 10C
Temperature of tank water is 10C
Temperature of the heated tank water is 10C
Time of day is 13:30
Amount of sunlight is 0
================================================
The Climate System is now operating.

The time is 13:30
The inside temperature is 22C
The thermostat setting is 24C
Method chosen to heat the building: Solar Collector To Exchanger
================================================
::::::::::::::::::::::::::::::::::::::::::::::::::
PANIC occurred, NORMAL shutdown.
::::::::::::::::::::::::::::::::::::::::::::::::::
Last chosen heating method was: Solar Collector to Exchanger
Thermostat Setting is 24C
Inside temperature is 22C
Temperature of collector water is 10C
Temperature of tank water is 10C
Temperature of the heated tank water is 10C
Time of day is 13:30
Amount of sunlight is 0
::::::::::::::::::::::::::::::::::::::::::::::::::

Goodnight, Irene...
-
```

Figure 7-1. Climate Control Program Listing and Sample Run (Cont'd.)

Obviously this book only introduces DEBUG-86. For a complete description, see the *Intellec Series III Microcomputer Development System Console Operating Instructions*.

USING ICE-88™, AN IN-CIRCUIT EMULATOR

The ICE-88 emulator consists of three circuit boards, a cable, a buffer box, and software on disk. The three circuit boards fit inside your Intellec Series III development system, and the cable and buffer box assembly connects your Series III system to your prototype hardware. A forty-pin plug at the end of the cable plugs into your prototype system in place of your prototype's CPU, allowing the in-circuit emulator hardware to emulate all functions of your prototype's CPU. If you do not have any prototype hardware, keep the plug's protector closed to prevent damage to its pins.

In-circuit emulation shortens your development time in two ways. It provides symbolic debugging capabilities and diagnostic hardware debugging capabilities, and it lets you borrow resources (like memory) from your Series III system until your prototype system is complete.

Your program can be loaded into borrowed memory, prototype memory, or any combination of the two, and run as if it were resident in the prototype. You can emulate program execution at real-time speed or in single- or multiple-instruction steps. You can also stop emulation manually at any time to examine system status, or you can specify breakpoints as you can with DEBUG-86.

We will debug the **PLMDATA** module (first shown in Chapter 5) to see if it performs its port input and interpolations correctly. We added statements (shaded in figure 7-2) to the module that will execute the procedures. Figure 7-2 shows the listing of the compiled **PLMDATA** module.

```
PL/M-86 COMPILER     PLMDATA                                                    PAGE   1

ISIS-II PL/M-86 V121 COMPILATION OF MODULE PLMDATA
OBJECT MODULE PLACED IN PLMDEB.OBJ
COMPILER INVOKED BY:  PLM86 PLMDEB.SRC DEBUG


   1          PLMDATA: DO;

   2   1      DECLARE START LABEL PUBLIC;
   3   1      DECLARE (SETTING, TEMPERATURE) WORD;

              /*
              INPUTS
              ------

              THERMOSTAT$SETTING$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input ports HIGH and LOW: Three BCD digits for the thermostat setting:

                                  Port HIGH, bits 3-0: hundred's digit
                                  Port LOW, bits 7-4: ten's digit
                                  Port LOW, bits 3-0: unit's digit


              TEMP$DATA$FROM$PORTS:

              Formal parameters HIGH and LOW receive port numbers as actual parameters.

              Input port HIGH: Binary ADC output of thermocouple, high-order 8 bits
              Input port LOW: Binary ADC output of thermocouple, low-order 8 bits


              OUTPUTS
              -------

              THERMOSTAT$SETTING$FROM$PORTS:  Return WORD with setting in Celsius

              TEMP$DATA$FROM$PORTS:  Return WORD with temperature in Celsius

              */
```

Figure 7-2. Listing of the Modified PLMDATA Module

```
  4   1     THERMOSTAT$SETTING$FROM$PORTS:
            PROCEDURE (HIGH, LOW) WORD;
  5   2        DECLARE (HIGH, LOW) WORD;
  6   2        DECLARE (IN$PORT$HIGH, IN$PORT$LOW) BYTE;
  7   2        DECLARE THERMO$SETTING WORD;
  8   2        DECLARE (HUNDREDS, TENS, UNITS) BYTE;

  9   2        IN$PORT$HIGH = INPUT(HIGH);
 10   2        IN$PORT$LOW = INPUT(LOW);
 11   2        HUNDREDS = IN$PORT$HIGH AND 00001111B;
 12   2        TENS = SHR(IN$PORT$LOW, 4);
 13   2        UNITS = IN$PORT$LOW AND 00001111B;
 14   2        THERMO$SETTING = UNITS + 10*TENS + 100*HUNDREDS;
 15   2        RETURN THERMO$SETTING;
 16   2     END THERMOSTAT$SETTINGS$FROM$PORTS;


            /* Another typed procedure to return temperature data, which
               uses the INTERPOLATE typed procedure. */

            /* INTERPOLATE is a typed procedure that receives thermocouple
               voltage and returns temperature in Celsius using an
               interpolation routine */

 17   1     INTERPOLATE:
            PROCEDURE(VOLT$IN) WORD;
 18   2        DECLARE VOLTS(*) WORD DATA(0,51,102,154,206,258,365,472);
 19   2        DECLARE T$CEL(*) WORD DATA(0,10,20,30,40,50,70,90);
 20   2        DECLARE (I, VOLT$IN, NUMERATOR) WORD;

 21   2        I = 0;
 22   2        DO WHILE VOLT$IN > VOLTS(I);
 23   3           I = I + 1;
 24   3        END;

               /* Shift for rounding, and return Celsius temperature */
 25   2        NUMERATOR = SHL((VOLT$IN-VOLTS(I-1))*(T$CEL(I)-T$CEL(I-1)), 1);
 26   2        RETURN T$CEL(I-1) + SHR(NUMERATOR/(VOLTS(I)-VOLTS(I-1))+1, 1);

 27   2     END INTERPOLATE;


 28   1     TEMP$DATA$FROM$PORTS:
            PROCEDURE(HIGH,LOW) WORD;
 29   2        DECLARE (HIGH,LOW) WORD;
 30   2        DECLARE IN$PORT$HIGH WORD; /* ADDRESS in McCracken's book */
 31   2        DECLARE IN$PORT$LOW BYTE;
 32   2        DECLARE (THERMOCOUPLE$OUTPUT, TEMPERATURE) WORD;

 33   2        IN$PORT$HIGH = INPUT(HIGH);
 34   2        IN$PORT$LOW = INPUT(LOW);
 35   2        THERMOCOUPLE$OUTPUT = SHL(IN$PORT$HIGH, 8) OR IN$PORT$LOW;
 36   2        TEMPERATURE = INTERPOLATE(THERMOCOUPLE$OUTPUT);
 37   2        RETURN TEMPERATURE;

 38   2     END TEMP$DATA$FROM$PORTS;

 39   1     START: DO;
 40   2              SETTING = THERMOSTAT$SETTING$FROM$PORTS(2000,1000);
 41   2              TEMPERATURE = TEMP$DATA$FROM$PORTS(200,100);
 42   2              END START;
 43   1     END PLMDATA;


MODULE INFORMATION:

     CODE AREA SIZE     = 012CH    300D
     CONSTANT AREA SIZE = 0020H     32D
     VARIABLE AREA SIZE = 0016H     22D
     MAXIMUM STACK SIZE = 0012H     18D
     101 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 7-2. Listing of the Modified PLMDATA Module (Cont'd.)

We also have to link the LARGE.LIB run-time library to this module, and locate the final module:

```
-RUN LINK86 :F1:PLMDAT.OBJ, :F1:LARGE.LIB TO :F1:PLMDAT.LNK<cr>
     .
     .
     .
-RUN LOC86 :F1:PLMDAT.LNK RESERVE(200H TO 77FFH) TO :F1:PLMDAT.86<cr>
     .
     .
     .
```

To use the ICE-88 emulator, copy the ICE88 program from its disk onto a disk in your system (for example, the system disk in drive 0), and type the following command:

```
-ICE88<cr>
```

You can now use the ICE-88 LOAD command to load the module and its symbols:

```
*LOAD :F1:PLMDAT.86<cr>
*SYMBOLS<cr>
```

The SYMBOLS command displays the entire symbol table. We can start emulation with the GO command, specifying both a starting location and a breakpoint. We can use the .START label as a starting location, and specify line #8 as a breakpoint:

```
*GO FROM .START TILL #8 EXECUTED<cr>
EMULATION BEGUN
.
.
.
EMULATION TERMINATED, CS:IP=0781:0040H
```

At this point, we want to change the contents of the 8088 ports so that the THERMOSTAT$SETTING$FROM$PORTS procedure can pick them up:

```
*PORT 2000=0  ;Hundred's digit of BCD thermostat setting.<cr>
*PORT 1000=24 ;Tens and units of BCD setting.<cr>
*GO FROM CS:IP TILL .SETTING WRITTEN<cr>
EMULATION BEGUN
.
.
EMULATION TERMINATED, CS:IP=0781:002CH
```

Note that a semicolon and comment is allowed on an ICE-88 command line. We assigned 24H to port 1000 to obtain a 2 as the tens digit and a 4 as the units digit (to represent a thermostat setting of 24 degrees Celsius). Then we resumed emulation from the program counter (PC) until a value was obtained for the .SETTING variable. At this point, we can check the contents of .SETTING:

```
*WORD .SETTING<cr>
WOR 0794:0024H = 0018H
                        18 hexadecimal is 24 in decimal.
```

Since .SETTING is correct, we can set the contents of the other two ports and continue emulation:

```
*PORT 200=0  ;High order 8 bits are zero.<cr>
*PORT 100=66 ;Low order equal 66H, or 102 in decimal.<cr>
*GO FROM CS:IP TILL #43 EXECUTED<cr>
EMULATION BEGUN
.
.
.
EMULATION TERMINATED, CS:IP=0781:003BH
*WORD .TEMPERATURE<cr>
WOR 0794:0026H = 0014H
*
                          14 hexadecimal is 20 in decimal.
```

We have a temperature reading of 20 degrees Celsius.

Obviously this session only introduces the ICE-88 emulator. The *ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users* contains both tutorial and reference information on the ICE-88 emulator. The similarities between DEBUG-86 and in-circuit emulators enhance their usefulness in software development efforts, since both provide symbolic debugging. In-circuit emulation also lets you emulate all of your prototype CPU functions, even though your prototype CPU is not installed, and even if your prototype has not been built. It is a powerful debugging and diagnostic tool for both the hardware and software of your final product.

EXECUTION ENVIRONMENTS

The Intellec Series III system provides an 8086 execution environment and operating system support—the support your program needs to be able to access devices and files. When you link the run-time support libraries to your Pascal-86 program, you are providing the software your program needs to "talk" to the Series III operating system.

You can also run Pascal-86 programs in other systems, or in dedicated application environments, as long as you provide the run-time support software. For example, you could transfer your program to RAM on an SDK-86 (System Design Kit with an 8086), or to RAM on an iSBC 86/12A Single Board Computer system, by first using the OH86 utility described in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems* to convert the program to hexadecimal object format, and then using an appropriate tool to load the program into your execution board (the ICE-86 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957 Interface and Execution Package).

You could also transfer your program to ROM on an SDK-86 kit, iSBC Single Board Computer system, or your own custom-designed hardware, by using the Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software.

Figure 7-3 shows possible execution paths for Pascal-86 programs.

The Series III operating system has a standard set of primitives (service routines) that any program can use. Intel supplies *run-time support libraries* that act as an interface between your Pascal-86 program and the Series III system. By replacing this interface with your own custom-designed interface, you can use the same Pascal-86 programs on other non-Intel systems. With each future Intel system, Intel will supply the appropriate run-time interface so that your present programs will also run in future Intel systems.

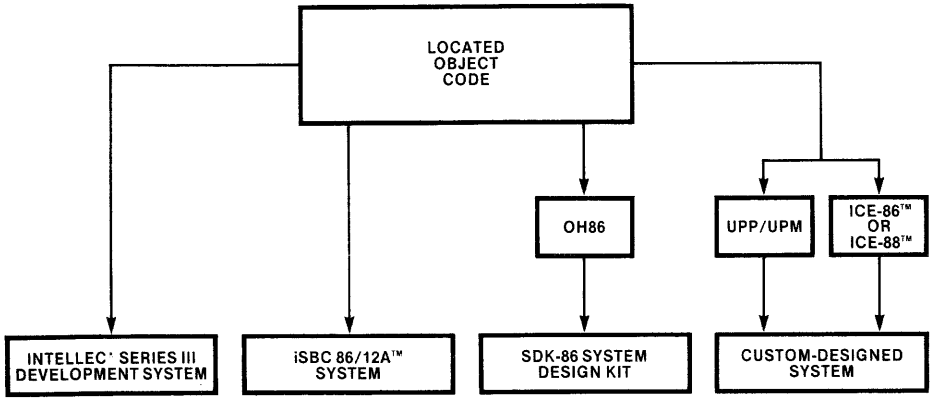


Figure 7-3. Possible Execution Paths for Pascal-86 Programs 121632-8

A system library is also supplied for PL/M and assembly language programs; this library (or set of libraries) also acts as an interface between your programs and the Series III operating system. By supplying your own interface, you can also use these programs on other systems.

The Series III system was designed in this modular fashion to provide operating system support without necessarily binding programs to that particular system. It was designed to be *used* as part of your application (as the operating environment), but it was also designed to be useful for the development of applications that do not need full-blown operating system support. With several layers of interfacing between the system and your program, you can choose exactly how much system you want in your final application, and you can preserve your software investment with an eye to the future.

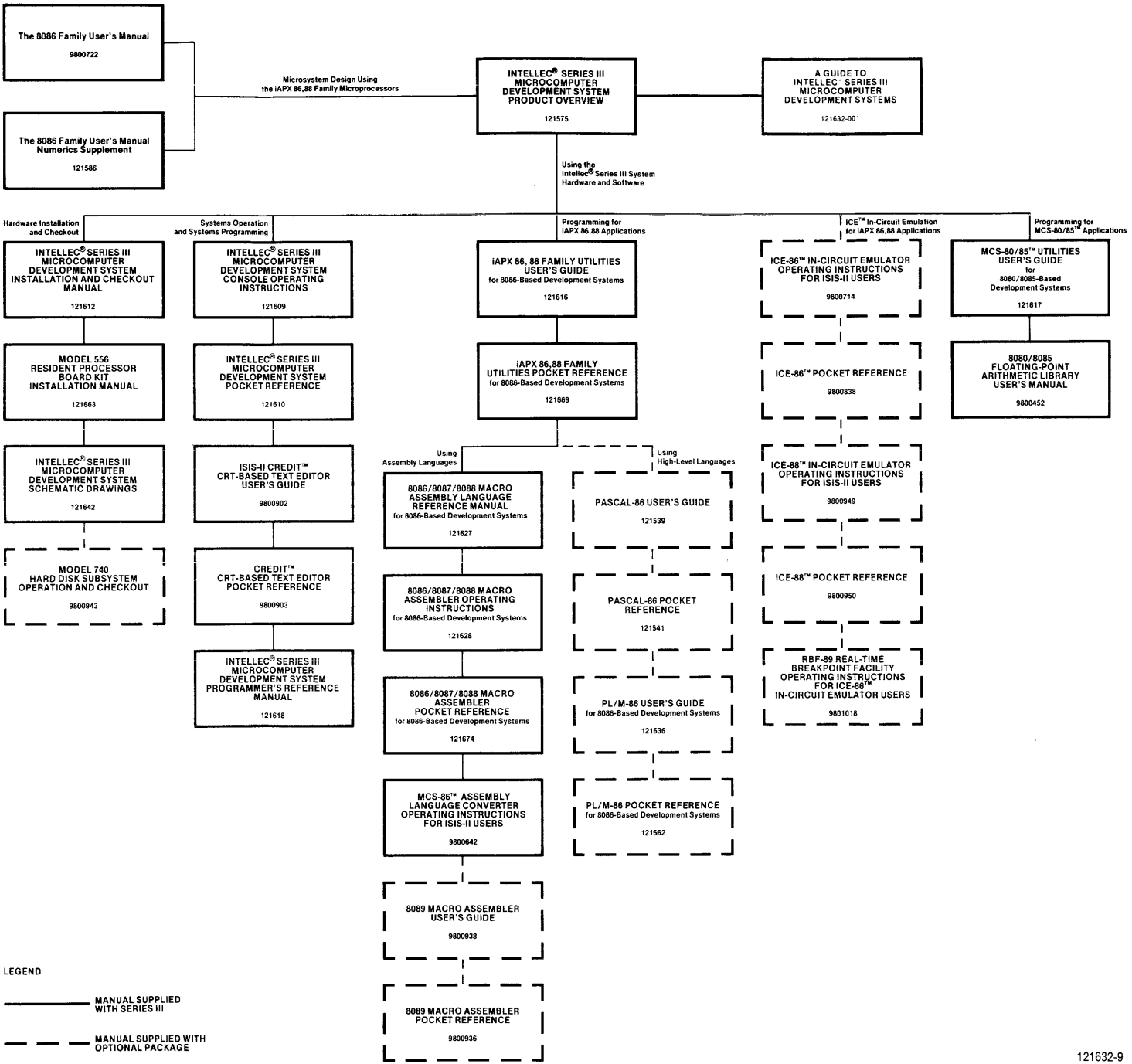
BIBLIOGRAPHY

1. Brooks, Frederick P., *The Mythical Man-Month: Essays on Software Engineering*; Addison-Wesley Pub. Co., Phillipines, 1975.
2. Brown, P.J., "Programming and Documenting Software Projects," *Computing Surveys*, Dec. 1974, pp. 213-220.
3. Dahl, O. J., Dijkstra, E. W., and Hoare, C.A.R., *Structured Programming*; Academic Press, New York, 1972.
4. Dijkstra, E. W., "The Humble Programmer," 1972 Turing Award Lecture, *Communications of the ACM*; Vol. 15, No. 10, Oct. 1972.
5. Floyd, Robert W., "The Paradigms of Programming," 1978 Turing Award Lecture, *Communications of the ACM*; Vol. 22, No. 8, Aug. 1979.
6. Hueras, J. F., Ledgard, H. F., and Nagin, P. A., *Pascal With Style (Programming Proverbs Series)*; Hayden Book Co., Rochelle Park, NJ., 1979.
7. Jensen, Kathleen, and Wirth, Nicklaus, *PASCAL User Manual and Report*; Springer-Verlag, New York, Heidelberg, and Berlin, 1975.
8. Kernighan and Plauger, *The Elements of Programming Style* (2nd Edition); Bell Telephone Labs, Inc., Murray Hill, NJ, and McGraw-Hill, New York, 1978.
9. Kernighan and Plauger, *Software Tools*; Bell Telephone Labs, Inc., New Jersey, Yourdon inc., New York, and Addison-Wesley, Cambridge MA, 1976.
10. Ledgard, Henry F., *Programming Proverbs*; Hayden Book Co., Rochelle Park, NJ., 1975.
11. McCracken, D. D., *A Guide to Intellec Microcomputer Development Systems*; Intel Corp., 1978 (No. 9800558B).
12. McCracken, D. D., *A Guide to PL/M Programming For Microcomputer Applications*; Addison-Wesley Pub. Co., Menlo Park, CA., 1978.
13. Morse, Stephen P., *The 8086 Primer: An Introduction to Its Architecture, System Design, and Programming*; Hayden Book Co., Rochelle Park, New Jersey, 1980.
14. Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*; Vol. 15, No. 12, Dec. 1972.
15. Polya, G., *How To Solve It* (2nd Ed.); Princeton Univ. Press and Doubleday Anchor Books, Doubleday & Company, Inc., Garden City, NY, 1957.
16. Strunk, William, Jr., and White, E. B., *The Elements of Style* (3rd Ed.); Macmillan, New York, 1979.
17. Weinberg, Gerald M., *PL/I Programming Primer*; McGraw-Hill, Inc., New York, 1966.
18. Wirth, Nicklaus, "Program Development by Stepwise Refinement," *Communications of the ACM*; Vol. 14, No. 4, April 1971.
19. Wirth, Nicklaus, *Systematic Programming: An Introduction*; Prentice-Hall, Englewood Cliffs, NJ, 1973.

INDEX

Within this index, *f* or *ff* after a page number means *and the following page* (or *pages*).

The Intellec® Series III Publications Library


The 8086 Family User's Manual
9800722
The 8086 Family User's Manual Numerics Supplement
121586
Microsystem Design Using the iAPX 86,88 Family Microprocessors
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM PRODUCT OVERVIEW
121575
A GUIDE TO INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEMS
121632-001
Using the Intellec® Series III System Hardware and Software
Hardware Installation and Checkout
Systems Operation and Systems Programming
Programming for iAPX 86,88 Applications
iCE™ In-Circuit Emulation for iAPX 86,88 Applications
Programming for MCS-80/85™ Applications
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM INSTALLATION AND CHECKOUT MANUAL
121612
MODEL 556 RESIDENT PROCESSOR BOARD KIT INSTALLATION MANUAL
121663
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM SCHEMATIC DRAWINGS
121642
MODEL 740 HARD DISK SUBSYSTEM OPERATION AND CHECKOUT
9800943
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS
121609
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM POCKET REFERENCE
121610
ISIS-II CREDIT™ CRT-BASED TEXT EDITOR USER'S GUIDE
9800902
CREDIT™ CRT-BASED TEXT EDITOR POCKET REFERENCE
9800903
INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM PROGRAMMER'S REFERENCE MANUAL
121618
iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE for 8086-Based Development Systems
121616
iAPX 86,88 FAMILY UTILITIES POCKET REFERENCE for 8086-Based Development Systems
121669
Using Assembly Languages
Using High-Level Languages
8086/8087/8088 MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL for 8086-Based Development Systems
121627
8086/8087/8088 MACRO ASSEMBLER OPERATING INSTRUCTIONS for 8086-Based Development Systems
121628
8086/8087/8088 MACRO ASSEMBLER POCKET REFERENCE for 8086-Based Development Systems
121674
MCS-86™ ASSEMBLY LANGUAGE CONVERTER OPERATING INSTRUCTIONS FOR ISIS-II USERS
9800642
8089 MACRO ASSEMBLER USER'S GUIDE
9800938
8089 MACRO ASSEMBLER POCKET REFERENCE
9800936
PASCAL-86 USER'S GUIDE
121539
PASCAL-86 POCKET REFERENCE
121541
PL/M-86 USER'S GUIDE for 8086-Based Development Systems
121636
PL/M-86 POCKET REFERENCE for 8086-Based Development Systems
121662
ICE-86™ IN-CIRCUIT EMULATOR OPERATING INSTRUCTIONS FOR ISIS-II USERS
9800714
ICE-86™ POCKET REFERENCE
9800838
ICE-88™ IN-CIRCUIT EMULATOR OPERATING INSTRUCTIONS FOR ISIS-II USERS
9800949
ICE-88™ POCKET REFERENCE
9800950
RBF-86 REAL-TIME BREAKPOINT FACILITY OPERATING INSTRUCTIONS FOR ICE-86™ IN-CIRCUIT EMULATOR USERS
9801018
MCS-80/85™ UTILITIES USER'S GUIDE for 8080/8085-Based Development Systems
121617
8080/8085 FLOATING-POINT ARITHMETIC LIBRARY USER'S MANUAL
9800452
LEGEND
MANUAL SUPPLIED WITH SERIES III
MANUAL SUPPLIED WITH OPTIONAL PACKAGE

121632-9