Coronado Enterprises Modula-2 TUTOR

March 16, 1987

CORONADO ENTERPRISES MODULA-2 TUTORIAL - Version 1.00
(Generic Version)

Coronado Enterprises - 12501 Coronado Ave NE - Albuquerque, New Mexico 87122

Contents

Introduction to the Modula-2 Tutorial

Welcome to the programming language Modula-2, a very complete, high level language with many advanced features. Modula-2 was designed by Niklaus Wirth, the designer of Pascal. Based on experience with Pascal, Modula-2 was designed to make up for many of the deficiencies noted by programmers worldwide, and the changes make it a very powerful language with few limitations. In spite of its power, Modula-2 retains the simplicity of Pascal and can be used for small applications as easy as Pascal could be used.

MODULA-2 TUTORIAL - PART I

Even though there are many similarities between the two languages, the differences are significant. This tutorial was written considering both the similarities and the differences between the languages. The first part of this tutorial is composed of those features that are common to Pascal and Modula-2 and are also of a fundamental nature. You will need to study all of Part I in order to write meaningful Modula-2 programs. If you are already a fairly experienced Pascal programmer, you will absorb this material very fast. Be sure to go through it all at least once because there are many small differences between the languages that you must consider.

MODULA-2 TUTORIAL - PART II

The topics taught in Part II of this tutorial are those advanced features that are also available in Pascal. Some of these topics are pointers, dynamic allocation, records, and linked lists. They are very powerful tools that can be used to great advantage but are quite often overlooked by many Pascal programmers that I have talked to. These are the tools that give Pascal and Modula-2 an advantage in flexibility over such languages as BASIC and FORTRAN. They do require a bit of deep concentration to understand, but you will be greatly rewarded if you take the time to understand and use them.

MODULA-2 TUTORIAL - PART III

Part III of this tutorial covers those aspects of Modula-2 that are not included in Pascal in any way. Some of the topics are independent compilation, the entire topic of modules, and concurrent processing. These are advanced topics and some of these topics may be the reasons that you selected Modula-2 as a programming language. The material covered in Part I in conjunction with that covered in Part III can lead to some very powerful programming techniques. To efficiently use this tutorial, you must carefully study all of the material in Part I, then you can do a lot of jumping around in Parts II and III and still cover the material in a meaningful manner. You may also choose to only study some chapters of the last two parts in order to learn the needed material for the programming problem at hand. I would like to emphasize that it is important that you cover the material in Part I very carefully and in order since so much depends on what was taught before. When you get to the last two parts, comments at the beginning of each chapter will tell you what parts need to be reviewed in order to effectively use the material in the chapter.

FOR THE BEGINNING PROGRAMMER

If you are a novice to computer programming, this course is for you because it is assumed that you know nothing about programming. Many sections, especially in the early chapters, will cover very basic topics for your benefit. The biggest problem you will have will be setting up your compiler for use, since this can be a very difficult task. Possibly you know someone with experience that would be willing to help you get started.

FOR ALL PROGRAMMERS

There are, at this time, a very limited number of Modula-2 compilers available, but it would not be possible to include notes on every compiler about how to install it for your computer. The COMPILER.DOC file on your distribution disk contains notes on all of the compilers we have had access to and some of the difficulties in setting them up for the IBM-PC or near compatibles. In addition, all compilers do not implement all functions defined in Niklaus Wirth's definition of the language. As many of these as we have found are listed in the same file. Finally, all of the problems in compiling the files on this disk are noted in the COMPILER.DOC file. It would be worth your effort to print out this file and keep the hardcopy handy while you are working your way through the lessons.

Modula-2, as defined by Niklaus Wirth, contains no provisions for input or output because they are so hardware dependent. It is up to each compiler writer to provide you with supplemental procedures for handling I/O (input/output) and a few other machine dependent features. Niklaus Wirth did recommend a library of I/O routines that should be available and most compilers contain at least those facilities, and usually provide many more. The COMPILER.DOC file will contain notes about differences in these facilities for those compilers which we have access to. The COMPILER.DOC file will be updated anytime new information is available.

SIMPLE EXAMPLES WILL BE USED

All of the instructional programs are purposely kept simple and small to illustrate the point intended. It is of little value to you to present you with a large complex program to illustrate what can be illustrated in a small program better. In addition, every program is complete, and can be compiled and run. Program fragments frequently pose as many questions as they answer.

Because it would be a disservice to you to teach you a lot of simple techniques and never show you how they go together in a significant program, chapters 9 and 16 contain several larger example programs. A relatively small amount of description is given about these programs because you will have already covered the details and only need a quick overview of how to put the various constructs together. You will find some of these programs useful and will have the ability to modify and enhance them for your use since you have the source code.

SOME VARIABLE NAMES SEEM SILLY, WHY IS THAT?

I have seen example programs with the same name everywhere, and had a hard time deciding what names were required and what could be changed to something more meaningful. For example a "SORT" program is in a file named "SORT", the program name is "SORT", the input file is named "SORT", and variables were named "SORT1", SORT2", etc. This was no help to myself, a novice sorter, and would be no help to you. For that reason the first program is in a file named "PUPPYDOG.MOD" and the module name is "PuppyDog". It should be obvious to even the newest programmer that the name of a module can be anything if it is allowed to be "PuppyDog". You will learn later that well selected names can be a great aid in understanding a program. This will be evident in some of the early programs when variable names are chosen to indicate what type of variable they are.

Some compilers require that the module name be the same as the file name, and all require them to agree when you get to global modules because of the way "Type checking" is accomplished. It would be best for you to get into the habit of naming them both the same now. For that reason, all of the example programs use the same name for the file name and for the module name. Your compiler may allow you to use different names. It will be left up to you to study your manual and see if this is so for your compiler.

WHAT IS A COMPILER?

There are two primary methods used in running any computer program that is written in a readable form of English. The first method is an interpreter. An interpreter is a program that looks at each line of the "english" program, decides what the "english" on that line means, and does what it says to do. If one of the lines is executed repeatedly, it must be scanned and analyzed each time, greatly slowing down the solution of the problem at hand.

A compiler on the other hand, is a program that looks at each statement one time and converts it into a code that the computer understands directly. When the compiled program is actually run, the computer does not have to figure out what each statement means, it is already in a form the computer can run directly, hence a much faster execution of the program. Due to the nature of Modula-2, there will be few, if any, interpreters.

WHAT ABOUT THE PROGRAMMING EXERCISES?

The programming exercises at the end of each chapter are a very important part of the tutorial. If you do them, you will embed the principles taught in each chapter more firmly in your mind than if you ignore them. If you choose to ignore them, you will be somewhat adept at reading Modula-2 programs but very ineffectual at writing them. By doing the exercises, you will also gain considerable experience in using your editor and compiler.

It will be assumed that you know how to use your compiler and that you have some kind of an editor for use with the example files. With the above in mind, you are now ready to begin your tour of Modula-2.

A sample program is included in this chapter for you to try with your compiler. It is left as an exercise for you to compile and run FIRSTEX.MOD. When you can successfully compile and run this program, you are ready to begin the tutorial on Modula-2 programming. Do not worry about what the statements mean in FIRSTEX.MOD, you will have a complete understanding of this program by the time you complete chapter 4.

```
MODULE FirstEx;

FROM InOut IMPORT WriteLn, WriteString, WriteCard;

VAR Index : CARDINAL;

BEGIN

   WriteString("This is our first example program");
   WriteLn;
   WriteLn;
   FOR Index := 1 TO 12 DO
      WriteString("The value of the index is now ");
      WriteCard(Index,3);
      WriteLn;
   END

END FirstEx.
```

Chapter 1 – What is a computer program?

If you are a complete novice to computers, you will find the information in this chapter useful. If you have some experience in computer use, and especially programming, you can completely ignore this chapter. It will deal with a few of the most fundamental topics of computers and will have nothing to do with the Modula-2 programming language.

WHAT IS A COMPUTER PROGRAM?

A computer is nothing but a very dumb machine that has the ability to perform mathematical operations very rapidly and very accurately, but it can do nothing without the aid of a program written by a human being. Moreover, if the human being writes a program that turns good data into garbage, the computer will very obediently, and very rapidly turn good data into garbage. It is possible to write a large program with one small error that will do just that. In some cases the error will be obvious, but if the error is subtle, the answers may appear to be right, and the error will go unnoticed. It is up to you, the human programmer, to write a correct program to tell the computer what to do. You can think of the computer as your very obedient slave ready to do your every whim. It is up to you to tell your slave what you want it to do.

A computer program is a "recipe" which the computer will use on the input data to derive the desired output data. It is similar to the recipe for baking a cake. The input data is comparable to the ingredients, including the heat supplied by the oven. The program is comparable to the recipe instructions to mix, stir, wait, heat, cool, and all other possible operations on the ingredients. The output of the computer program can be compared to the final cake sitting on the counter ready to be cut and served. A computer then is composed of two parts, the data upon which the program operates, and the data. The data and program are inseparable as implied by the last sentence.

WHAT ARE CONSTANTS?

Nearly any computer program requires some numbers that never change throughout the program. They can be defined once and used as often as needed during the operation of the program. To return to the recipe analogy, once you have defined how big a tablespoon is, you can use the same tablespoon without regard to what you are measuring with it. When writing a computer program, you can define the value of PI = 3.141592, and continue to use it wherever it makes sense knowing that it is available, and correct.

WHAT ARE VARIABLES?

In addition to constants, nearly any computer program uses some numbers that change in value throughout the program. They can be defined as variables, then changed to any values that make sense to the proper operation of the program. An example would be the number of eggs in the above recipe. If a single layer of cake required 2 eggs, then a triple layer cake would require 6 eggs. The number of eggs would therefore be a variable.

HOW DO WE DEFINE CONSTANTS OR VARIABLES?

All constants and variables have a name and a value. In the last example, the name of the variable was "eggs", and the value was either 2 or 6 depending on when we looked at the stored value. In a computer program the constants and variables are given names in much the same manner, after which they can store any value within the defined range. Any computer language has a means by which constants and variables can be first named, then assigned a value. The means of doing this in Modula-2 will be given throughout the remainder of this tutorial.

WHAT IS SO GOOD ABOUT MODULA-2?

Some computer languages allow the programmer to define constants and variables in a very haphazard manner and then combine data in an even more haphazard manner. For example, if you added the number of eggs, in the above recipe, to the number of cups of flour, you would arrive at a valid mathematical addition, but a totally meaningless number. Some programming languages would allow you to do just such an addition and obediently print out the meaningless answer. Since Modula-2 requires you to set up your constants and variables in a very precise manner, the possibility of such a meaningless answer in minimized. A well written Modula-2 program has many cross checks to minimize the possibility of a completely scrambled and meaningless output.

Notice however, in the last statement, that a "well written" Modula-2 program was under discussion. It is still up to the programmer to define the data structure in such a way that the program can prevent garbage generation. In the end, the program will be no better than the analysis that went into the program design.

If you are a novice programmer, do not be intimidated by any of the above statements. Modula-2 is a well designed

tool that has been used successfully by many computer novices and professionals. With these few warnings, you are ready to begin.

Chapter 2 – Getting started in Modula-2

OUR FIRST MODULA-2 PROGRAM

We are ready to look at our first instructional program in Modula-2. Assuming that you have a full screen editor of some type, load the program PUPPYDOG.MOD and display it on your screen. It is an example of the minimum Modula-2 program. There is nothing that can be left out of this program and still have a compileable, executable program.

```
MODULE PuppyDog;

BEGIN

END PuppyDog.
```

The first word in the program, "MODULE", is the name that identifies a module, and it must be written as given here, in all capital letters. During the entire first part of this tutorial, we will use only this type of a module. There are other types but we will not look at any of them until we get to part III of this tutorial. Modula-2 requires us to name our module so we give it a name, "PuppyDog". We could have used any name that qualifies as an identifier but we have chosen a name that has nothing to do with computers as an illustration that any name could be used. In a practical program, you would probably use a name that was descriptive of the program in some way.

WHAT IS AN IDENTIFIER?

An identifier is a combination of letters and numbers that Modula-2 uses to identify a variable, program name, procedure name, and several other quantities. In Modula-2, an identifier is composed of any number of characters. The characters may be any mix of alphabetic and numeric characters, but the first character must be an alphabetic character. The case of the alphabetic character is significant such that "IdentNumber1", "IDENTNUMBER1", and "IdEnTnUmBeR1" are all different identifiers. No spaces or any other special characters are allowed.

BACK TO THE PROGRAM UNDER CONSIDERATION

The "header" line is terminated with a semicolon according to the formal definition of Modula-2. A semicolon is a statement separator and many will be used in large programs. Following the semicolon, we come to the program itself. The program statements are enclosed between the two words "BEGIN" and "END". In this case there are no statements, but if there were some, they would be placed between the two indicated words. Finally, the module name is repeated after the "END" and it is followed by a period. The module name is repeated in order to make the program easier to understand by clearly marking its limits. In this case it really doesn't add to the clarity of the program, but in a large program it can be of significant help. The period marks the end of the listing and can be thought of as the period that marks the end of a sentence.

The three words, MODULE, BEGIN, and END, are special words in Modula-2. They are "reserved words" because they are used for a specific purpose and cannot be used for any other purpose. They are not available for your use in any way except for the defined purpose. The reserved words in Modula-2 are always capitalized or the compiler will not consider them as reserved words. Remember that alphabetic characters must have the correct case in Modula-2. Some other languages, most notably Pascal, allow you to use either case anywhere and it converts them internally so that they are the same. It would be permissible for you to use words such as "Begin" or "End" as variables in a Modula-2 program, but it would be very poor programming practice and should be avoided. We will come across many other reserved words in these lessons. There are 40 reserved words in Modula-2.

You should have learned how to use your compiler by now so you can compile and run this program. It will do nothing, but that is significant in itself, because it should at least return to the operating system after it finishes doing nothing. That may sound a little silly, but it does take a considerable amount of effort to load, transfer control to the program, and set up linkage back to your Disk Operating System.

It should be noted at this time that the Modula-2 compiler doesn't care about extra blanks or linefeeds and the careful programmer will insert extra blanks and linefeeds as desired in order to make the program easier to read. As you continue to program in Modula-2, you will no doubt develop a style of your own and hopefully your programs can be read easily by other programmers.

A PROGRAM THAT DOES SOMETHING

Load and display the program named WRITESM.MOD for an example of a Modula-2 program that does something. First you should notice that the elements of the first program are still here as they will be in every Modula-2 program. The same three reserved words are used here as before, but now there are some added statements.

```
MODULE WriteSm;

FROM InOut IMPORT WriteLn, WriteString;

BEGIN

   WriteString("This line will be displayed on the monitor.");
   WriteLn;
   WriteString('This line will be displayed too.');
   WriteLn;
   WriteString("This will all be ");
   WriteString('on one line.');
   WriteLn;
   WriteString('She said, "I ');
   WriteString("don't ");
   WriteString('like dogs."');
   WriteLn;

END WriteSm.
```

The line near the beginning that begins with the reserved word "FROM" is a special line that must be used in any program that accesses external procedures. We will not try to define this line at this time. We will only say that every external call in Modula-2 requires a definition of where to find the procedure. The module named "InOut" is a collection of input and output routines that are available for our use and this line in the program tells the system to look in the "InOut" collection for the procedures named "WriteLn" and WriteString". When the program needs these particular functions to do what we ask it to do, it knows where to find them. We will cover the IMPORT list in detail later in this tutorial. Until then, simply use the example programs as a guide when you wish to write a practice program.

OUR FIRST PROGRAM STATEMENTS

Between the BEGIN and END statements, which we defined previously as the place where the actual program is placed, we have a series of "WriteString" and WriteLn" statements. These statements are almost self explanatory, but we will say a few words about them anyway. Each line is a call to a "procedure" which is a very important feature of Modula-2. A "procedure" is an external servant that does a certain job for us in a well defined way. In the case of the "WriteString", it looks at the string of characters supplied to it and displays the string of characters on the monitor at the current cursor position. In the case of the "WriteLn" procedure, it serves us by moving the cursor down one line on the monitor and moving it to the left side of the screen.

The parentheses are required for the WriteString because it has data following it. The data within the parentheses is data supplied to our slave or helper. It gets the string of characters between the quotation marks or the apostrophes and displays the string on the monitor. You have a choice of delimiters so that you can output the delimiters themselves. If you desire to output a quotation mark to the monitor, use apostrophes for delimiters, and if you wish to output apostrophes, use quotation marks. If you wish to output both, break the line up and output it piecemeal as in the last example line.

This program should be very clear to you by now. First we tell the system where to get the procedures, then we list the procedures in the order required to produce the desired results. It should be apparent that the lines of the program between the reserved words BEGIN and END are simply executed in order. Compile and run the program and observe the output on your monitor. It should be mentioned at this point that it is possible to redirect the output to the printer or to a disk file but we will not be doing that for quite some time. We will stay with the basic syntax of Modula-2 for now.

MODULA-2 COMMENTS

No program is complete without a few comments embedded in the program as notes to the programmer describing the reasons for doing some particular thing. The notes are particularly helpful to another programmer who needs to modify the program some day. It is not necessary for the computer to understand the notes and in fact, you don't want the computer to try to understand the notes, so you tell the compiler to ignore the notes completely. How to do this is the object of our next program named MODCOMS.MOD which you should load and display on your monitor.

```
(* This is a comment that is prior to the start of the
   program itself. *)

MODULE ModComs;

FROM InOut IMPORT WriteLn, WriteString;

                     (* This is a block
                        of comments that
                        illustrate one way
                        to format some of
                        your comments.    *)

BEGIN  (* This is the beginning of the main program *)

   WriteString("This is a comments demo program.");
   WriteLn;
(* WriteString("This will not be output.");
   WriteString("Nor will this."); *)

END ModComs.  (* This is the end of the main program *)

(* This is a comment after the end of the program *)
```

In Modula-2, comments are enclosed in pairs of double characters. The comment is started with the "(*", and ended with the "*)". The program on your monitor has several examples of comments in it. If the comments were completely removed, the program would be very similar to the last one but a lot shorter. Notice that comments can go nearly anywhere in a program, even before the header statement or after the ending period. Comments can be used to remove a section of program from consideration by the compiler so that a particularly troublesome section of code can be "commented out" until you solve some of the other problems in program debugging. It is important to remember that comments can be "nested" in Modula-2 so that a section of code can be "commented out" even if it contains other comments.

This particular program is not meant to be an example of good commenting. It is really a sloppy looking program that would need some work to put it into a good style, but it does illustrate where it is possible to put comments.

GOOD PROGRAMMING STYLE

Load and display the program named GOODFORM.MOD for an example of a well formatted program. Since Modula-2 allows you to use extra spaces and blank lines freely, you should use them in any way you can to make your programs easy to understand, and therefore easy to debug and modify. Special care has been given to style in this program and it payed off in a very easy to understand program. Even with your very limited knowledge of Modula-2 programming you can very quickly decipher what it does. It is so well formatted that comments are not needed and they would probably detract from its readability. No further comment is needed or will be given. Compile and run this program to see if it does what you think it will do.

```
MODULE GoodForm;

FROM InOut IMPORT WriteLn, WriteString;

BEGIN

   WriteString("Programming style ");
   WriteString                 ("is a matter of ");
   WriteString                               ("personal choice.");
   WriteLn;
   WriteString("Each person ");
   WriteString            ("can choose ");
   WriteString                       ("his own style.");
   WriteLn;
   WriteString("He can be ");
   WriteString          ("very clear, or extremely messy.");
   WriteLn;

END GoodForm.
```

REALLY BAD FORMATTING

Load and display UGLYFORM.MOD for an excellent example of bad formatting. If you can see at a glance what this program does you deserve the Nobel Prize for understanding software if such a thing exists. The syntax for this program follows all of the rules of Modula-2 programming except for good style. Without saying anything else about this mess, I would suggest that you try to compile and run it. You may be surprised to find that it does compile and run, and in fact it is identical to the last program. Keep in mind that you can add extra blanks and linefeeds anyplace you desire in a program to improve its readability.

```
MODULE UglyForm
;FROM
                                                              InOut
IMPORT WriteLn, WriteString;BEGIN
WriteString("Programming style ");WriteString("is a matter of ");
WriteString("personal choice.")                 ;WriteLn;WriteString
("Each person ")
                                                       ;WriteString
("can choose ");
                           WriteString
("his own style.");WriteLn;WriteString("He can be ");WriteString
("very clear, or extremely messy.")
;WriteLn;END UglyForm
.
```

Hopefully, the last two programs will be an indication to you that good programming style is important and can be a tremendous aid in understanding what a program is supposed to do. You will develop your own programming style as time goes by. It is good for you to spend some effort in making your program look good, but don't get too excited about it yet. Initially, you should expend your effort in learning how to program in Modula-2 with reasonable style and strive to improve your style as you go along. It would be good for now if you simply tried to copy the style given in these lessons.

PROGRAMMING EXERCISES

1. Write a program that will display your name on the monitor.

2. Write a program that will display your name on the monitor along with your address, city and state in 3 separate lines.

3. Add a comment to the MODCOMS.MOD file between the words "IMPORT" and "WriteLn" to see if the compiler will allow you to put a comment within a statement.

Chapter 3 – The simple Modula-2 data types.

The material in this chapter is extremely important to you as you strive to become a good Modula-2 programmer, but you may also find it to be somewhat tedious because it contains so many facts. This material is needed in order to develop the topics in the next few chapters, but all of the details are not necessarily required. For that reason, you may wish to go through it rather rapidly picking up the high points and come back to this chapter for the details later when they will be much more meaningful. Do not completely pass over this material at this time or the next few chapters will be meaningless unless you are already highly experienced in other programming languages.

A PROGRAM WITH VARIABLES

Load and display the program named INTVAR.MOD for our first program with some variables in it. This program begins in the usual way since it has a MODULE header and the IMPORT list. Next we come to a new reserved word, VAR. This word is used to indicate to the compiler that we wish to define one or more variables. In Modula-2, there is a rule that says you can use nothing until it is defined. If we wish to use a variable in the program, we must first define that it will exist, and what kind of a variable it is. After that, it can be used in the program to do what needs to be done.

```
(* Chapter 3 - Program 1 *)
MODULE IntVar;

FROM InOut IMPORT WriteLn, WriteString, WriteInt;

VAR Count : INTEGER;  (* The sum of two variables *)
    x,y   : INTEGER;  (* The two variables to add *)

BEGIN

   x := 12;
   y := 13;
   Count := x + y;

           (* Assignments complete, now display the results *)

   WriteString("The value of x  =");
   WriteInt(x,3);
   WriteLn;
   WriteString("The value of y  =");
   WriteInt(y,4);
   WriteLn;
   WriteString("The sum of them =");
   WriteInt(Count,6);
   WriteLn;

   x := 0FFH;    (* This is the way to assign a hexadecimal number *)
   y := 177B;    (* This is the way to assign an octal number      *)

END IntVar.
```

Following the reserved word VAR, we have the variable named "Count" defined. The reserved word INTEGER following the colon states that the variable "Count" will be of type INTEGER. This means that it can store any whole number between -32768 to 32767. Don't worry too much about this yet, the next program will completely define what an INTEGER type variable is. It is important to recognize that after we have defined the variable "Count", it still doesn't have a value stored in it, that will come later.

The next line has two more variables defined, namely "x", and "y". They are also INTEGER type variables and do not have a value stored in them yet. You can think of the three variables as three empty boxes, each capable of storing a number but with no number in them yet. It would be perfectly permissible to put all three variables on one line, or to have separated them such that each was on a separate line. At this point, the program doesn't know that there is any difference between them, because there isn't any. The fact that one will contain the sum of the other two has no meaning yet, the comments are only for us, not the computer.

USING VARIABLES IN A PROGRAM

Now we will go to the program itself. The first line sets the variable "x" equal to 12, in effect putting the number 12 in the box mentioned earlier. The sign := is the Modula-2 symbol for assignment. It is most meaningful to read the symbol "gets the value of" since it is not really stating a mathematical equality but is saying in effect, "assign the value of this to the variable at the left." The entire line can be read as "x gets the value of 12." There is now a value assigned to the variable "x" declared in the header. The next statement assigns the value of 13 to the variable "y". Finally the value of the data stored in the variable "x" is added to the value of the data stored in the variable "y", and the sum is stored in the variable "Count". We have therefore done our first calculations in Modula-2 but we will do many more before this tutorial is completed.

Notice that each statement is terminated with a semicolon, a Modula-2 requirement.

The three variables are then displayed on the monitor with appropriate prose to identify them. The only new statement here is the "WriteInt" procedure that needs a little explanation. This procedure is used to output an INTEGER type variable to the monitor or whatever device is being used. By definition, it contains two quantities within the parentheses, the variable name and the number of columns it should fill. If there are not enough columns to output the data, more will be used so that no digits will be truncated. If all are not needed, leading blanks will be output. If the variable "x" had the value of 1234 when we came to program line 18, all four digits would be output in spite of the request for three. Since "x" has the value of 12, only two columns will be used and one leading blank will be output. In like manner, "y" is allotted 4 columns and "Count" is to be output in 6 columns.

The last two lines of the program assign new values to two of the variables. The variable "x" is assigned the value of FF hexadecimal which is 256 decimal, and "y" is assigned the value of 177 octal which is 127 decimal. This is only done as an illustration to you of how it is done. If you don't understand these two numbering systems, simply ignore this until you have a need for it.

Compile and run this program to see if it does what you expect it to do. The important thing to notice in this program is the variable definition in the definition part of the module and the variable assignment in the program part. It should be obvious, but it would be well to mention that

the definition part of the module extends from the module name to the reserved word "BEGIN" and is where all definitions are put. Likewise, the program part of the module includes all statements from the "BEGIN" to the "END".

SIMPLE VARIABLE TYPES

Modula-2 has several predefined data types that you can use in your programs. You also have the ability to define any number of complex types built up from the simple types but we will not discuss this until we get to chapter 6 and beyond. The simple types are INTEGER, CARDINAL, REAL, BOOLEAN, and CHAR. Each has its own purpose and its own peculiarities and we will cover each type one at a time.

THE SIMPLE VARIABLE - INTEGER

Load and display the program named INTMATH.MOD for an example of INTEGER math. In the declaration part of the program (the part prior to the BEGIN) we have 7 INTEGER type variables defined for use in the program. We will use them to illustrate

INTEGER arithmetic.

```
(* Chapter 3 - Program 2 *)
MODULE IntMath;

FROM InOut IMPORT WriteLn, WriteString, WriteInt;

VAR IntSum, IntDif, IntMul, IntDiv, IntRem : INTEGER;
    A, B                                   : INTEGER;

BEGIN
   A := 9;             (* Simple assignment                *)
   B := A + 4;         (* Addition                         *)
   IntSum := A + B;    (* Addition                         *)
   IntDif := A - B;    (* Subtraction                      *)
   IntMul := A * B;    (* Multiplication                   *)
   IntDiv := B DIV A;  (* Integer division, the result is a
                          truncated integer number.        *)
   IntRem := B MOD A;  (* d is the remainder of the integer
                          division.                        *)
   A := (A + B) DIV (3*B + 7);  (* Composite math statement  *)

   WriteString("The integer values are ");
   WriteInt(IntSum,6);
   WriteInt(IntDif,6);
   WriteInt(IntMul,6);
   WriteInt(IntDiv,6);
   WriteInt(IntRem,6);
   WriteLn;

   INC(A);        (* This increments the value of A   *)
   DEC(A);        (* This decrements the value of A   *)
   INC(A,3);      (* This adds 3 to the value of A    *)
   DEC(A,7);      (* This reduces the value of A by 7 *)
   INC(A,B*2+4); (* A composite incrementing amount  *)

   A := MIN(INTEGER);  (* This produces the smallest INTEGER *)
   B := MAX(INTEGER);  (* This produces the largest INTEGER  *)

END IntMath.
```

An INTEGER variable, by definition, can store any whole number between -32768 and 32767. An attempt to store any other value in an INTEGER type variable should produce an error by your compiler but it may produce some other result. Some compilers may store a -32769, which is one count too low, as a 32767 which is at the top end of the range. This is due to the two's complement arithmetic that you don't need to understand at this point. It will be left to you to determine what your compiler does in such a case.

The first line in the program is nothing new to you, it simply assigns the variable "A" the value of 9. The second line adds 4 to the value stored in the variable "A" and the result, 13, is stored in the variable "B". Next the values stored in the variables "A" and "B" are added together and the sum, which is 9 + 13, is stored in the variable "IntSum". Continuing in the same manner, the difference and the product are calculated and stored. When we come to INTEGER division, we are breaking new ground because the result is truncated to the largest whole number resulting from the division. Thus 13 DIV 9 results in 1 because the remainder is simply dropped. The next construct, B MOD A results in the remainder of the division, in this case 4. You will find these operations very useful as you progress as a Modula-2 programmer.

The intent of the next line is to illustrate that you can do several math operations in a statement if you are careful to put the parentheses in the proper places. There are definite rules about operator precedence but I recommend that you use lots of parentheses to remove all doubt as to what the results will be.

The results of the operations are displayed in 6 columns and we move on to several new operations. The first new operation is the "INC" which is short for "increment". This simply increments the variable contained within the parentheses and if a second argument is given, the variable is incremented by the value of that variable. In like manner the "DEC" procedure decrements the variable in the parentheses by one unless a second argument is given in which case the variable is decremented by the value of that variable.

It may not be clear at this point, but the second variable itself may be another variable name or even a composite of several as long as it results in an INTEGER type variable. This is illustrated in the program.

Finally, we come to the last two procedures in this program, the "MIN" and the "MAX". These two procedures will return the value of the smallest possible INTEGER, -32768 and the largest possible INTEGER, 32767. These are the values returned for a 16 bit microcomputer which is what you are probably using since that is what this tutorial is intended for. It would be well to add that not all Modula-2 compilers implement these functions so you may need to comment out these two lines in order to compile and run this program.

Compile and run this program and observe the output. If your compiler results in errors, you may have to make some changes in order to compile it. Refer to the COMPILER.DOC file on the distribution disk for notes on some of the more popular Modula-2 compilers.

THE SIMPLE VARIABLE - CARDINAL

Load and display the file named CARDMATH.MOD for an example of CARDINAL mathematics and output. In this file, 7 variables are defined as CARDINAL and one more as INTEGER. A CARDINAL variable can store any whole number from 0 to 65535 in a 16 bit microcomputer, although the range may be different if you are using an unusual computer or compiler.

```
(* Chapter 3 - Program 3 *)
MODULE CardMath;

FROM InOut IMPORT WriteLn, WriteString, WriteCard;

VAR CardSum, CardDif, CardMul, CardDiv : CARDINAL;
    A, B, CardRem                      : CARDINAL;
    IntVar                             : INTEGER;

BEGIN
   A := 9;              (* Simple assignment                  *)
   B := A + 4;          (* Addition                           *)
   CardSum := A + B;    (* Addition                           *)
   CardDif := B - A;    (* Subtraction                        *)
   CardMul := A * B;    (* Multiplication                     *)
   CardDiv := B DIV A;  (* Integer division, the result is a
                           truncated integer number.         *)
   CardRem := B MOD A;  (* d is the remainder of the integer
                           division.                         *)
   A := (A + B) DIV (3*B + 7);  (* Composite math statement  *)

   WriteString("The cardinal values are ");
   WriteCard(CardSum,6);
   WriteCard(CardDif,6);
   WriteCard(CardMul,6);
   WriteCard(CardDiv,6);
   WriteCard(CardRem,6);
   WriteLn;

   IntVar := A;       (* INTEGER and CARDINAL are assignment  *)
   B := IntVar + 27;  (* compatible, but cannot be mixed in   *)
                      (* any expression.                      *)

   A := 125;          (* CARDINAL assignment                  *)
   B := A - 112;      (* CARDINAL math                        *)
(* B := 125 + (-112);    Illegal CARDINAL Math - see text     *)

   IntVar := 125 + (-112);       (* INTEGER math, OK here     *)

   INC(A);       (* This increments the value of A       *)
   DEC(A);       (* This decrements the value of A       *)
   INC(A,4);     (* This adds 4 to the value of A        *)
   DEC(A,6);     (* THis subtracts 6 from the value of A *)
   A := MIN(CARDINAL);   (* This produces the minimum CARDINAL *)
   B := MAX(CARDINAL);   (* This produces the maximum CARDINAL *)

END CardMath.
```

The first few lines are the same as the last program so very little needs to be said about them except for the subtraction example. In this case, the result of the subtraction would be negative if it were carried out as in the last program so "A" is subtracted from "B". It is an error to attempt to store a negative number in a CARDINAL type variable. For that reason, a CARDINAL should not be used if there is any chance that it will be required to go negative. Programming experience will be the best teacher when it comes to deciding what variables to use in each situation.

In this program the variables are once again displayed, but now the procedure named "WriteCard" is used for output because the variables to output are CARDINAL.

The next two statements indicate that INTEGER and CARDINAL variables are "assignment compatible" meaning that they can be assigned to each other with the := operator. They cannot however, be mixed in calculations. Constants in an expression are assumed to be of the same type as the variables in the expression and they must agree. For that reason, the expression in line 36 is invalid because (-112) is required to be a CARDINAL constant but it is negative and therefore not CARDINAL. In the prior line it is permissible to subtract 112 from the value of "A" as long as the result is still positive. As an exercise, change line 34 such that a number less than 112 is assigned to "A". The program will compile without error but when you run it, you should get a runtime error because the CARDINAL assignment is out of range. Notice that the constant value of -112 is allright for use an an INTEGER variable.

The remaining statements in the program are the same as the last program so additional explanation is unneeded. It would be good to point out that in the case of CARDINAL, the "MIN" and "MAX" procedures will return values of 0 and 65535 for most 16 bit implementations.

Compile and run this program remembering that it may be necessary to comment out the "MIN" and "MAX" statements to get a successful compilation.

THE SIMPLE VARIABLE - REAL

Load and display the program named REALMATH.MOD for a demonstration of the data type REAL. The definition part of this program is similar to the last with some additions to the IMPORT list. Your compiler may use different names for some of the procedures here, so if you get a compile error you will need to modify these. We will study the IMPORT (and EXPORT) list in detail later, so be patient.

```
(* Chapter 3 - Program 4 *)
MODULE RealMath;

FROM InOut IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteReal;
FROM MathLib0 IMPORT sin, cos; (* Your system may use a different
                                  name for either the module or the
                                  trig functions.                  *)

VAR Sum, Diff, Product, Div : REAL;
    A, B    : REAL;
    Inumber : INTEGER;
    Cnumber : CARDINAL;

BEGIN
   A := 3.234;                    (* Assigns a value          *)
   B := A + 1.0123;               (* Add a constant           *)
   Sum := A + B;                  (* Add two variables        *)
   Product := A * B;              (* Multiplication           *)
   Div := A / B;                  (* Division                 *)
   Diff := A - B;                 (* Subtraction              *)
   A := (A + B)/(12.345 * A - B); (* Multiple math expression *)
   B := sin(A)*cos(B);

   WriteString("The REAL values are");
   WriteReal(Sum,12);
   WriteString("  ");
   WriteReal(Diff,12);
   WriteString("  ");
   WriteReal(Product,12);
   WriteString("  ");
   WriteReal(Div,12);
   WriteLn;

         (* Conversion between data types - illustration  *)

   Inumber := 15;          (* This is an INTEGER              *)
   Cnumber := 333;         (* This is a CARDINAL              *)
   A := FLOAT(Inumber);    (* INTEGER to REAL                 *)
   B := FLOAT(Cnumber);    (* CARDINAL to REAL                *)
   Inumber := TRUNC(Sum);  (* REAL to INTEGER                 *)
   Cnumber := TRUNC(Sum);  (* REAL to CARDINAL                *)

   A := MIN(REAL);   (* This produces the smallest REAL *)
   A := MAX(REAL);   (* This produces the largest REAL  *)

END RealMath.
```

Several REAL variables and one each of the INTEGER and CARDINAL types are defined for use in the program. The REAL type variable can contain numbers in a wide range and with fractional parts included. The exact range, and the accuracy will vary widely depending on your implementation. It will be up to you to check your reference manual for the limits on your computer and compiler. A REAL type number is defined as one with a decimal point. The mathematics are the same as with the other two except that the division symbol is the slash (/). There is no "MOD" for REAL type numbers because there is theoretically no remainder, since a fractional part is computed as part of the calculation.

The four results are displayed on the monitor with 12 columns allowed for each result and two extra blanks displayed between each number. Unfortunately, we have no control over how many digits will be displayed following the decimal point. This would be nice for outputting data in a financial model where we would like to have two digits following the decimal point. When we get to the advanced part of this tutorial, we will write our own procedure for doing that in such a way that we can call it from any program just like we call these output procedures.

CONVERSION BETWEEN DATA TYPES

Beginning in line 37, we assign the INTEGER and CARDINAL variables some values and convert the values to type REAL by using the procedure "FLOAT". We then convert the variable "Sum" to INTEGER and CARDINAL by use of the procedure "TRUNC". The fractional part, if any, will simply be thrown away. These procedures will be very useful in many of your programs.

The last two lines return the value of the largest possible REAL number and the smallest REAL number for your implementation. Once again, your compiler may not support these two functions and they may have to be commented out in order to compile.

Compile and run this program.

THE SIMPLE VARIABLE - BOOLEAN

Load and display the file named BOOLMATH.MOD on your monitor for an example of BOOLEAN variables. A BOOLEAN variable can only have one of two possible values, TRUE or FALSE. These variables cannot be printed directly but can be used to control other print statements to print out a representation of their value. We will see how later.

```
(* Chapter 3 - Program 5 *)
MODULE BoolMath;
```

```
VAR IsIt, WillIt, What : BOOLEAN;
    A, B, C            : INTEGER;

BEGIN

   A := 22;        (* Assign some values to work with *)
   B := 12;
   C := -12;

   IsIt := A = 22;      (* TRUE    - equal to                   *)
   IsIt := A = 23;      (* FALSE   - equal to                   *)
   WillIt := A > B;     (* TRUE    - greater than               *)
   WillIt := A < B;     (* FALSE   - less than                  *)
   What := WillIt;      (* FALSE   - assignment                 *)
   IsIt := B <= 12;     (* TRUE    - less than or equal         *)
   IsIt := B >= 4;      (* TRUE    - greater than or equal      *)
   IsIt := A # B;       (* FALSE   - not equal                  *)
   IsIt := A <> B;      (* TRUE    - not equal                  *)

   IsIt := TRUE;
   What := FALSE;
   WillIt := IsIt AND What;      (* FALSE because What is FALSE *)
   WillIt := IsIt AND NOT What;  (* TRUE                        *)
   WillIt := IsIt OR What;       (* TRUE because one is TRUE    *)
   WillIt := NOT IsIt OR What;   (* FALSE                       *)
   IsIt := (A = B) OR (B = C) OR (A = 22);
   IsIt := ((A < B) AND (B < C)) OR NOT (B > C);

   (* We have not studied a way to print out representations of *)
   (* BOOLEAN variables so it will have to wait.                 *)

END BoolMath.
```

We define 3 BOOLEAN variables and 3 INTEGER variables and assign values to the 3 INTEGER variables in the program for use in these illustrations. In line 13 the BOOLEAN expression "A = 22" is TRUE, therefore the BOOLEAN variable "IsIt" is assigned the value TRUE. The variable "IsIt" could be used later in the program to make a decision, by a yet undefined method, to do something or bypass it. In like manner, the next statement assigns "IsIt" the value FALSE because A is not equal to 23. The remainder of the allowed BOOLEAN expressions are defined in the next few lines and are left for your inspection and study.

Beginning in line 25, composite BOOLEAN expressions are illustrated. As many BOOLEAN expressions as desired can be combined with AND and OR operators. If two or more BOOLEAN expressions are combined with the AND, the result is TRUE if all expressions are TRUE. If two or more BOOLEAN expressions are combined with the OR, the result is true if any of them are TRUE. The NOT operator inverts the sense of what it modifies, it turns a TRUE to FALSE and vice-versa. Finally a couple of composite BOOLEAN expressions are given for illustration of the amount of complexity that is allowed, although there is no real limit as to how far you can go with the complexity. Good programming practice would dictate that you keep it simple and understandable.

TWO RULES FOR BOOLEAN EVALUATION

First it is important that you use the same type of variables within a BOOLEAN expression. REAL's can be compared to REAL's and INTEGER's to INTEGERs, but REAL's cannot be compared to INTEGER's. CARDINAL and CHAR types can also be compared to their own types, but none of the four can be compared directly to each other.

Secondly, Modula-2 uses a shortcut evaluation technique for BOOLEAN evaluation. Evaluation proceeds from left to right and if it finds a result which will positively determine the outcome, evaluation stops. For example, if it is evaluating a string of 5 comparisons all combined with an AND, and it finds that the second term is FALSE, evaluation stops there. Since all terms must be TRUE for the result to be TRUE, it makes no difference what values the last three are, the result will be FALSE because of the second term.

THE SIMPLE VARIABLE - CHAR

Load and display the program named CHARDEMO.MOD for an illustration of the last simple variable type, CHAR. Text data is stored in a computer in a format utilizing the CHAR data type. Although there are exceptions, such as when text is stored in some form of a packed mode, this is nearly always true. This tutorial was written with a word processor that uses a CHAR type for text storage, and few word processors use any other method.

```
(* Chapter 3 - Program 6 *)
MODULE CharDemo;

FROM InOut IMPORT WriteLn, Write, WriteString;

VAR Char1, Char2, Dog3, Cat4 : CHAR;
    Index                    : INTEGER;

BEGIN

   Char1 := 'A';                     (* This is a capitol A      *)
   Char2 := "T";                     (* This is a capitol T      *)
   Index := ORD(Char1) + 2;          (* The numerical value of A
                                        plus 2 = the value of C  *)
   Dog3 := CHR(Index);                (* The letter C             *)
   Cat4 := '"';                       (* The quotation mark       *)

   WriteString("The characters can spell ");
   Write(Cat4);
   Write(Dog3);
   Write(Char1);
   Write(Char2);
   Char1 := "S";                      (* Change to the letter S   *)
   Write(Char1);
   Write(Cat4);
   WriteLn;

   Char1 := 65C;                        (* This sets Char1 to 'A' *)
   Char1 := 'a';                        (* This sets Char1 to 'a' *)
   Char2 := CAP(Char1);                 (* This sets Char2 to 'A' *)

END CharDemo.
```

Although there are many different ways to store text, only two are used to any level of significance, EBCDIC and ASCII. ASCII is used almost exclusively in micro computers. I have never heard of an implementation that used EBCDIC in a microcomputer, so we will limit our discussion to ASCII. This merely refers to the way the characters of the alphabet and all other characters are represented in the computer. The ASCII standard defines that the value of 65 will be the letter 'A', 66 will be the letter 'B', etc. If everyone uses the same standard, transfer of data from one computer to another is greatly simplified.

The program named CHARDEMO has the usual header with 4 CHAR type variables defined for use in the program. An INTEGER is also defined. In the program itself, we begin by assigning 2 of the variables some CHAR data. Since a CHAR variable is capable of storing one letter, numeral, or special character, each variable is assigned one letter. The single or double quotes are used as an indication to the compiler that you intend for it to use the letter as a CHAR type variable rather than as another variable name. Of course if you wanted to use "A" as a variable name, you would have to define it in the definition part of the module.

TWO SPECIAL CHAR PROCEDURES

The next instruction gets the ordinal value of the letter "A", adds two to it, and assigns that value to the variable "Index", which must be an INTEGER (although it could have been defined as a CARDINAL). Refer to the documentation that came with your computer and you will find an ASCII table that will define the letter "A" as 65. Finally, the CHAR type variable "Dog3" is assigned the character value of "Index". Your ASCII table should define 67 as the letter "C". It is important to understand that the CHAR variable "Dog3" contains the character representation of the letter "C", and the

INTEGER variable "Index" contains the numerical value of the ASCII representation of the letter "C". It would be perfectly allright to use the variable "Index" for any desired numerical calculations, but not to display the letter "C". On the other hand, it would be allright to use the variable "Dog3" to display the letter "C" on the monitor but it could not be used for any calculations. The purpose therefore, of the two procedures "ORD" and "CHR", is to translate from one type to the other.

The variable "Cat4" is assigned the double quote by enclosing it in the single quotes, and the characters are output in a funny order to spell "CAT. The variable "Char1" is assigned the value "S", and the word is completed resulting in the full word "CATS" on the monitor after the program is compiled and run.

If this were the only way to use the CHAR type variable, it would be very tedious and frustrating, but there are other methods to use the CHAR type that are far more useful as you will see.

Next, an additional means of assigning a CHAR type variable is given. By assigning the CHAR variable "65C", it is the same as writing CHR(65), resulting in the variable having the internal value "A". A number less than 256 followed by a "C" is defined by Modula-2 as a CHAR type constant.

Finally, the variable "Char1" is assigned the letter "a" and it is converted to upper case "A" with the procedure CAP. This procedure will convert the argument to its upper case equivalent if it is a lower case letter. If its argument is an upper case letter or any other character, it will do nothing to it.

Compile and run this program and observe the output.

USING THE TRANSFER PROCEDURES

Load and display the file named TRANSFER.MOD for several examples of transferring data between the various simple data types. The transfer functions given here may not seem too important at this time, but some time spent here will help to reduce the frustration later when you get seemingly wrong errors that say you are using incompatible types in a statement. All of the program will not be discussed, only those statements that use some of the more unusual capabilities of Modula-2.

```
(* Chapter 3 - Program 7 *)
MODULE Transfer;
```

```
VAR Int1,  Int2  : INTEGER;
    Card1, Card2 : CARDINAL;
    Real1, Real2 : REAL;
    Char1, Char2 : CHAR;

BEGIN

   Int1  := 14;
   Int2  := 35;
   Card1 := Int1 + Int2 + 23;            (* assignment compatible *)
   Card2 := Card1 - 13 * 2 + CARDINAL(Int1);   (* mixed types     *)
   Card2 := Card1 - 13 * 2 + CARDINAL(Int1);   (* assignment comp *)
   Int1  := Int2 * INTEGER(Card1);

   Real1 := 12.0;
   Real2 := Real1 + FLOAT(Card2) * 1.112;     (* CARDINAL to REAL *)
   Real2 := Real2 + FLOAT(CARDINAL(Int1));    (* INTEGER to REAL  *)

(* Int1  := TRUNC(Real1) + Int2 * 3;         Incompatible error 1 *)
   Int1  := TRUNC(Real1) + CARDINAL(Int2) * 3;     (* error fixed *)
   Int1  := INTEGER(TRUNC(Real1)) + Int2 * 3;      (* error fixed *)

(* Card1 := TRUNC(Real1) + Int2 * 3;         Incompatible error 2 *)
   Card1 := INTEGER(TRUNC(Real1)) + Int2 * 3;      (* error fixed *)
   Card1 := TRUNC(Real1) + CARDINAL(Int2) *3;      (* error fixed *)

   Char1 := "A";
(* Int1  := ORD(Char1) + Int2;               Incompatible error 3 *)
   Int1  := INTEGER(ORD(Char1)) + Int2;            (* error fixed *)
   Int1  := ORD(Char1) + CARDINAL(Int2);           (* error fixed *)

   Card1 := ORD(Char1) + Card1;
   Real2 := FLOAT(ORD(Char1)) + 1.2345;

   Char1 := CHR(TRUNC(FLOAT(ORD(Char1))));      (* Sheer Nonsense *)

END Transfer.
```

In line 13, the calculations are done in INTEGER format, but due to the assignment compatibility of INTEGER and CARDINAL, the result is converted to CARDINAL across the ":=". Line 16 is an illustration of mixed mathematics using the transfer procedure INTEGER. Line 20 is the first example of "nested" procedures which must be done because FLOAT only uses a CARDINAL for an argument.

The expression in line 22 is an error because TRUNC results in a CARDINAL which cannot be added to an INTEGER. Either of the next two lines fix the problem by making the addition type-compatible then making use of the assignment compatibility between INTEGER and CARDINAL for line number 23. The same error occurs in line 26 and is fixed the same way in either of the next two lines. Once again, in line 31, the incompatible type error occurs and is fixed in either of two ways in the next two lines.

Lines 35 and 36 illustrate converting CHAR data to first CARDINAL then REAL which requires nested procedure calls. The last line of the program is a nest of procedures which converts a character from CHAR to CARDINAL, then to REAL, back to CARDINAL, and finally back to the original CHAR variable. It does nothing except act as a good illustration to you of what can be done.

Conversion between types is very important. You will use these techniques often so it is important to know how they work. A very simple yet helpful memory aid is to remember that any simple type can be converted to CARDINAL and CARDINAL can be converted to any type. Most other conversions require two steps to get from one to the other.

Chapter 14 will readdress this topic with even more extensive type transfer procedures.

PROGRAMMING PROBLEMS

1. Write a program in which you define, using the CHAR type variable, the letters "a" and "z", and the numbers "0" and "9". Convert them to CARDINAL, and display the four characters and their ASCII (numerical) values.

2. Write a program that you can easily modify to experiment with conversions between the various types that result in incorrect conversions to see the results on your compiler. For example, convert a -1 as an INTEGER to a CARDINAL.

Chapter 4 - Modula-2 Loops and Control Structures

Loops are some of the most important and most used constructs in computer programming and in all parts of your life. You use loops all the time for many things. Walking is a repetition of putting one foot in front of the other until you get where you are going. Eating a sandwich involves a loop of eating, chewing, swallowing, etc. In this chapter we will first cover all of the possible loops you can define in Modula-2, then go on to the control structures, the decision makers.

Load and display the program LOOPDEMO.MOD. This is a rather large program compared to the ones we have seen so far, but I felt it would be better to cover all of the loops in one file than have you compile and run 4 different files.

```
(* Chapter 4 - Program 1 *)
MODULE LoopDemo;

FROM InOut IMPORT WriteString, WriteInt, WriteLn, Write;

CONST Where = 11;

VAR  Index   : INTEGER;
     What    : INTEGER;
     Letter  : CHAR;

BEGIN

   WriteString("REPEAT loop     = ");
   Index := 0;
   REPEAT
     Index := Index + 1;
     WriteInt(Index,5);
   UNTIL Index = 5;          (* This can be any BOOLEAN expression *)
   WriteLn;

   WriteString("WHILE loop      = ");
   Index := 0;
   WHILE Index < 5 DO        (* This can be any BOOLEAN expression *)
      Index := Index + 1;
      WriteInt(Index,5);
   END;
   WriteLn;

   WriteString("First FOR loop  = ");
   FOR Index := 1 TO 5 DO
      WriteInt(Index,5);
   END;
   WriteLn;

   WriteString("Second FOR loop = ");
   FOR Index := 5 TO 25 BY 4 DO
      WriteInt(Index,5);
   END;
   WriteLn;

(* Note - The four loops above could use a CARDINAL type variable
          in place of the INTEGER type variable Index.  The next 2
          examples must use an INTEGER type variable because it
          must be capable of storing a negative value.            *)

   WriteString("Third FOR loop  = ");
   FOR Index := 5 TO -35 BY -7 DO
      WriteInt(Index,5);
   END;
   WriteLn;

   What := 16;
   FOR Index := (What - 21) TO (What * 2) BY Where DO
      WriteString("Fourth FOR loop = ");
      WriteInt(Index,5);
      WriteLn;
   END;

(* Note - The next two loops are demonstrations of using a CHAR
          type variable to index a FOR loop.                      *)

   FOR Letter := "A" TO 'Z' DO
      Write(Letter);
   END;
   WriteLn;

   FOR Letter := 'z' TO 'a' BY -1 DO
      Write(Letter);
   END;
   WriteLn;

(* Note - The following loop contains an EXIT which is a way to get
          out of the loop in the middle.                           *)

   Index := 1;
   LOOP
      WriteString("In the EXIT loop ");
      WriteInt(Index,5);
      IF Index = 5 THEN
         WriteLn;
         EXIT;
      END;
      WriteString("  We are still in the loop.");
      WriteLn;
      Index := Index + 1;
   END;

END LoopDemo.
```

REPEAT ... UNTIL LOOP

Ignoring the declaration part of the listing and going straight to the program itself, we first come to the REPEAT loop which does just what it says it will do. It will repeat until it is told to stop. The REPEAT in line 16 and the UNTIL go together, and everything between them will be executed until the condition following the UNTIL becomes TRUE. The condition can be any expression that will evaluate to a BOOLEAN answer, TRUE or FALSE. It can even be a composite expression with AND's, OR's, and NOT's like we studied in the last chapter. It can be composed of any of the simple types discussed so far as long as the terms are all compatible and it evaluates to a BOOLEAN value. In this case we have a very simple expression, "Index = 5". Since "Index" is initialized to 0 and is incremented each time we go through the loop, it will eventually reach a value of 5 and the loop will terminate, after which time the expressions following it will be executed.

We are not quite finished with the REPEAT loop yet, we will have more to say about it when we complete the WHILE loop.

WHILE LOOP

The WHILE loop is very much like the REPEAT loop except that the condition is tested at the beginning of the loop and when the condition becomes FALSE, the loop is terminated. Once again, the condition can be as complex as desired but in this case it is the very simple "Index < 5". When Index reaches 5, the loop is terminated and the statements following the loop are executed.

The biggest difference between the REPEAT and the WHILE loops is concerned with when the test is made. In the WHILE loop, the test is made at the beginning, so it is possible that the statements inside the loop will not be executed even once. In the REPEAT loop, the test is made at the end of the loop, so the statements in the loop will always be executed at least once. It is also good to keep in mind that the REPEAT stops when its condition goes TRUE, and the WHILE stops when its condition goes FALSE.

There is another loop that we can use in which we exit from the center using any test we can devise. It will be covered after we complete the FOR loop.

THE FOR LOOP

The FOR loop exists in one form or another in nearly every programming language and you will use it repeatedly because it is so useful. It uses the reserved words FOR, TO, BY, DO, and END. It uses any simple variable type except REAL, and counts loops depending on what counts you put in for beginning and ending points. The first example on line 31 says for the computer to start "Index" at 1 and count to 5, going through the loop once for each value of "Index". The count advances by 1 each time because nothing else is specified and 1 is the default. The end of the loop is specified by the reserved word END, and as many statements as desired can be within the body of the loop.

The next loop starts in line 37 and this time counts from 5 to 25 but incrementing by 4 each time because of the "BY 4" part of the line. The loop will continue until the second limit is going to be exceeded, at which time the loop will stop. The beginning and ending limits can themselves be some kind of a calculated value or a constant, the only provision being that they must be of the same type as the loop indexing variable. In fact they can be negative and the increment value can be negative. This is illustrated in the next loop that starts in line 48 where we count by -7 until we go from 5 to -35. No further explanation should be required for this loop.

The next loop, starting in line 54, uses calculated limits to determine its starting and ending points and it uses the name "Where" for its incrementing value. The value of "Where" is established in the definition part of this program as a constant. It is simply used here and will be explained in a future lesson when we get to it. "Where" is a constant with a value of 11, and the incrementing value must always be a constant.

The next two FOR loops use a CHAR type variable and simply "count" from "A" to "Z", or backwards in the case of the second one.

Several things should be pointed out about the FOR loop for you. The three values must agree in type, that is the index, the starting point, and the ending point. The index must not be changed by any logic within the loop or the results will be unpredictable. The value of the index must be assumed to be undefined after the loop terminates. You may discover that it is predictable on your compiler, but it may not be on some other compiler, and you may want to transfer your program to another system someday.

THE INFINITE LOOP

The fourth and final loop is an infinite loop, it never terminates by itself. It is up to you the programmer to see to it that some means of terminating it is available, the most usual is through use of the EXIT statement. Anyplace in the loop you can set up some conditions for exiting based on whatever you desire. Executing the EXIT procedure will cause the program control to leave the loop and begin executing the statements following the loop.

Now you have been exposed to the four loops available in Modula-2, the REPEAT, WHILE, FOR, and LOOP. Spend some time studying this program, then compile and run it to see if it does what you expect it to do. Loops are very important. You will do the vast majority of your logical control in loops and IF statements.

WHAT IS AN IF STATEMENT?

Load the program IFDEMO.MOD and display it on your monitor for an example of some IF statements. Ignoring the header we notice that the program is composed of one big loop in order to have some changing variables. Within the loop are 3 IF statements, the most used conditional statement in Modula-2.

```
(* Chapter 4 - Program 2 *)
MODULE IfDemo;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Index1 : INTEGER;

BEGIN

   FOR Index1 := 1 TO 8 DO
      IF Index1 < 4 THEN                   (* Simple IF statement *)
         WriteString("Index1 is less than 4.");
         WriteInt(Index1,4);
         WriteLn;
      END;  (* end of first IF statement *)

      IF Index1 = 5 THEN                   (* two way IF statement *)
         WriteString("Index1 is 5");
      ELSE
         WriteString("Index1 is not 5");
      END;  (* end of second IF statement *)
      WriteLn;

      IF Index1 = 2 THEN              (* multiple way IF statement *)
         WriteString("Index1 is 2");
      ELSIF Index1 = 6 THEN
         WriteString("Index1 is 6");
      ELSE
         WriteString("I really don't care what Index1 is");
      END;  (* end of third IF statement *)
      WriteLn;
   END;  (* of big FOR loop *)

END IfDemo.
```

The first IF statement is given in line 11. It simply says "if the value of Index1 is less than 4, then" do everything from the reserved word THEN to the reserved word END which is associated with it. If the value of Index1 is not less than 4, then all of these statements are ignored and the next statement to be executed will be the one following the reserved word END. In a nutshell, that is all there is to the simple IF statement. Once again, the condition can be any expression that will evaluate to a BOOLEAN result, and it can be composed of any of the simple types of data elements.

THE "ELSE" CLAUSE

The second IF statement, beginning in line 17 has an added feature, the ELSE clause. If the BOOLEAN expression does not evaluate to TRUE, then instead of the expressions following the THEN being executed, the group following the ELSE will be. Thus, if it is TRUE, everything from the THEN to the ELSE is executed, but if it is FALSE, everything from the ELSE to the END is executed. The END statement is therefore the terminator for the effect of the IF statement.

WHAT CAN GO IN THE IF STATEMENTS?

You may be wondering what is allowed to go into the group of executable statements between the THEN and the ELSE or some other place. The answer is, anything you want to put there. You can put other IF statements, loops, input or output statements, calculations, just about anything. If you indent the statements properly, you will even be able to read and understand what you put in there and why you put it there. Of course, if you put a loop in there, for example, you can put other constructs within the loop including other IF statements, etc. Thus you can go as far as you desire in building up a program.

THE ELSIF CLAUSE

The third and last kind of IF statement is given in the third example starting on line 24. In this case, if the expression within the IF statement is found to be FALSE, the statements following the THEN are skipped and the next construct is found, the ELSIF. If program control comes here, it has a further expression to evaluate, which if TRUE, will cause the statements immediately following its THEN to be executed. If this expression is found to be FALSE, the statements following the ELSE will be executed. The net result is that, one and only one of the 3 groups of instructions will be executed each time through the loop. It is permissible to add as many ELSIF cases as desired to this construct, leading to a "many way" branch. In addition, the ELSE is entirely optional regardless of whether or not the ELSIF's are used.

After studying this program, compile and run it and compare the results with what you expected.

LOOP's IN IF's IN LOOP's

Load and display the next example program LOOPIF.MOD for an example of some of the latest topics being combined. This program makes nonsense data but is valuable because it is small enough to understand quickly to see how LOOP's and IF's can be nested together. The entire program is a FOR loop containing an IF statement. Each part of the IF statement has a loop nested within it. There is no reason why this process could not be continued if there were a need to. Study this program then compile and run it.

```
(* Chapter 4 - Program 3 *)
MODULE LoopIf;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Index, Count, Dog : INTEGER;

BEGIN
   FOR Index := 1 TO 10 DO
      WriteString("Major loop");
      WriteInt(Index,3);
      IF Index < 7 THEN
         FOR Count := 15 TO (15 + Index) DO
            WriteString(' XXX');
         END;
         WriteLn;
      ELSE
         WriteString(' How many dogs?');
         FOR Dog := 1 TO 10 - Index DO
            WriteString("  too many");
         END;
         WriteLn;
      END;  (* ELSE part of IF statement *)

   END (* Major FOR loop *)
END LoopIf.
```

FINALLY, A MEANINGFUL PROGRAM

Load and display the program named TEMPCONV.MOD for your first look at a program that really does do something useful. This is a program that generates a list of temperatures in centigrade, converts the list to farenheit, and displays the list along with a note in the table at the freezing point and boiling point of water. You should have no difficulty understanding this program, so the fine points will be left up to you.

```
(* Chapter 4 - Program 4 *)

(* This program is a good example of proper formatting, it is   *)
(* easy to read and very easy to understand.  It should be a    *)
(* snap to update a program that is well written like this. You *)
(* should begin to develop good formatting practice early in    *)
(* your programming career.                                     *)

MODULE TempConv;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Count      : INTEGER;   (* a variable used for counting     *)
    Centigrade : INTEGER;   (* the temperature in centigrade    *)
    Farenheit  : INTEGER;   (* the temperature in farenheit     *)

BEGIN

   WriteString("Farenheit to Centigrade temperature table");
   WriteLn;
   WriteLn;

   FOR Count := -2 TO 12 DO
      Centigrade := 10 * Count;
      Farenheit := 32 + Centigrade *9 DIV 5;
      WriteString("   C =");
      WriteInt(Centigrade,5);
      WriteString("     F =");
      WriteInt(Farenheit,5);
      IF Centigrade = 0 THEN
         WriteString("   Freezing point of water");
      END;
      IF Centigrade = 100 THEN
         WriteString("   Boiling point of water");
      END;
      WriteLn;
   END; (* of main loop *)

END TempConv.
```

A few comments on good formatting is in order at this point. Notice the temperature conversion program and how well it is formatted. It is simple to follow the flow of control, and the program itself needs no comments because of the judicious choice of variable names. The block header at the top of the page is a good example of how you should get used to defining your programs. A simple block header of that variety goes a long way toward making a program maintainable and useful later. Take notice also of the way the variables are each defined in a comment. A program as simple as this probably doesn't need this much attention, but it would be good for you to get into practice early. It would be good for you to think of each of your programs as a work of art and strive to make them look good.

After spending some time studying this program, compile and run it to see what it does. Load and study the next program named DUMBCONV.MOD to see if you can figure out what it does. If you are really sharp, you will see that it is the same program as the last one but without all of the extra effort to put it into a neat, easy to follow format. Compile and run this program and you will see that they both do the same thing. They are identical as far as the computer is concerned. But there is a world of difference in the way they can be understood by a human being.

```
(* Chapter 4 - Program 5 *)
MODULE DumbConv;
FROM InOut IMPORT WriteString, WriteInt, WriteLn;
VAR a, b, c : INTEGER;
BEGIN
   WriteString("Farenheit to Centigrade temperature table");
   WriteLn;
   WriteLn;
   FOR a := -2 TO 12 DO
      b := 10 * a;
      c := 32 + b *9 DIV 5;
      WriteString("   C =");
      WriteInt(b,5);
      WriteString("     F =");
      WriteInt(c,5);
      IF b = 0 THEN
         WriteString("   Freezing point of water");
      END;
      IF b = 100 THEN
         WriteString("   Boiling point of water");
      END;
      WriteLn;
   END; (* of main loop *)

END DumbConv.
```

THE CASE STATEMENT

Load and display the program named CASEDEMO.MOD for an example of the last decision making construct in Modula-2, the CASE statement. A CASE statement is a "many-way" branch based on some simple variable. In this program we have a loop which sets the variable "Dummy" to the values from 1 to 25 successively. Each time it comes to the CASE statement, one of the branches is taken. The first branch is taken if the value is from 1 to 5, the second branch is taken if the value is from 6 to 9, the third is taken if it is either a 10 or 11, etc. Finally, if the value is not found in any of the branches, the ELSE path is taken as would be the case of a 12, a 13, or a few others. The important point is that one and only one of the many paths are taken each time the CASE construct is entered. The CASE variable can be any of the simple types except for the REAL type. For each path, as many statements can be executed as desired before the "|" is put in to end that path. The CASE statement is a powerful statement when you need it but you will not use it nearly as often as you will use the IF statement and the various loops.

```
(* Chapter 4 - Program 6 *)
MODULE CaseDemo;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Dummy : INTEGER;

BEGIN

   FOR Dummy := 1 TO 25 DO
      WriteInt(Dummy,4);
      WriteString("  ");
      CASE Dummy OF
         1..5        : WriteString("the number is small"); |
         6..9        : WriteString("it is a little bigger"); |
         10,11       : WriteString("it is 10 or 11"); |
         14..17      : WriteString("it is midrange"); |
         18,20,22,24 : WriteString("it is big and even"); |
         19,21,23    : WriteString("it is big and odd");
      ELSE
         WriteString("The number didn't make the list");
      END;  (* of CASE *)
      WriteLn;
   END;  (* of FOR loop *)

END CaseDemo.
```

PROGRAMMING EXERCISES

1. Write a program that will put your name on the monitor 10 times using a loop.

2. Write a program that lists the numbers from 1 to 12 on the monitor and prints a special message beside the number that represents the month of your birthday.

3. Write a program that calculates and lists the numbers from 1 to 8 along with the factorial of each. This will require use of a loop within a loop. A factorial is the number obtained by multiplying each number less than and up to the number in question. For example, factorial 4 = 1 * 2 * 3 * 4. Use a CARDINAL type var- iable for the result, then change it to an INTEGER to see the difference in output due to the range of the two different variable types. This is a good illustra- tion of the fact that careful choice of variable type is sometimes very important.

Chapter 5 - Modula-2 Procedures

In order to define the procedure, we will need to lay some groundwork in the form of a few definitions.

Program Heading - This is the easiest part since it is only one line, at least it has been in all of our programs up to this point. It is simply the MODULE line, and it never needs to be any more involved than it has been up to this point, except for one small addition which we will cover in a later chapter.

Declaration Part - This is the part of the Modula-2 source code in which all constants, variables, and other user defined auxiliary operations are defined. In some of the programs we have examined, there have been one or more VAR declarations and in one case a constant was declared. These are the only components of the declaration part we have used up to this time. There are actually four components in the declaration part, and the procedures make up the fourth part. We will cover the others in the next chapter.

Statement Part - This is the last part of any Modula-2 program, and it is what we have been calling the main program. It always exists bounded by the reserved words BEGIN and END just as it has in all of our examples to this point.

It is very important that you grasp the above definitions because we will be referring to them constantly during this chapter, and throughout the remainder of this tutorial. With that introduction, let us look at our first Modula-2 program with a procedure in it. It will, in fact, have three procedures.

WHAT IS A PROCEDURE?

A Procedure is a group of statements, either predefined by the compiler writer, or defined by you, that can be called upon to do a specific job. In this chapter we will see how to write and use a procedure. During your programming in the future, you will use many procedures. In fact, you have already used some because the "WriteString", "WriteLn", etc procedures you have been using are predefined procedures.

Load and display the program named PROCED1.MOD for your first look at a user defined procedure. In this program, we have the usual header with one variable defined. Ignore the header and move down to the main program beginning with line 26. We will come back to all of the statements prior to the main program in a few minutes.

```
(* Chapter 5 - Program 1 *)
MODULE Proced1;

FROM InOut IMPORT WriteString, WriteLn;

VAR Count : INTEGER;

PROCEDURE WriteHeader;
BEGIN
   WriteString("This is the header");
   WriteLn;
END WriteHeader;

PROCEDURE WriteMessage;
BEGIN
   WriteString("This is the message");
   WriteLn;
END WriteMessage;

PROCEDURE WriteEnding;
BEGIN
   WriteString("This is the end");
   WriteLn;
END WriteEnding;

BEGIN        (* Main program *)
   WriteHeader;
   FOR Count := 1 TO 8 DO
      WriteMessage;
   END;
   WriteEnding;
END Proced1.
```

The main program is very easy to understand based on all of your past experience with Modula-2. First we somehow write a header (WriteHeader), then write a message out 8 times (WriteMessage), and finally we write an ending out (WriteEnding). Notice that with the long names for the parts, no comments are needed, the program is self documenting. The only problem we have is, how does the computer actually do the three steps we have asked for. That is the purpose for the 3 procedures defined earlier starting in lines 8, 14, and 20. Modula-2 requires that nothing can be used until it has been defined, so the procedures are required to be defined prior to the main program. This may seem a bit backward to you if you are experienced in some other languages like FORTRAN, BASIC, or C, but it will make sense eventually.

HOW DO WE DEFINE A PROCEDURE?

We will begin with the PROCEDURE at line 8. First we must use the reserved word PROCEDURE followed by the name we have chosen for our procedure, in this case "WriteHeader" which is required to follow all of the rules for naming an identifier. Following the PROCEDURE line, we can include more IMPORT lists, define variables, or any of several other things. We will go into a complete definition of this part of the program in the next chapter. I just wanted to mention that other quantities could be inserted here. We finally come to the procedure body which contains the actual instructions we wish to execute in the procedure. In this case, the procedure body is very simple, containing only a "WriteString" and a "WriteLn" instruction, but it could have been as complex as we needed to make it.

At the end of the procedure, we once again use the reserved word END followed by the same name as we defined for the procedure name. In the case of a procedure, the final name is followed by a semicolon instead of a period. Other than this small change, a procedure definition is identical to that of the program itself.

When the main program comes to the "WriteHeader" statement, it knows that it is not part of its standard list of executable instructions, so it looks for the user defined procedure by that name. When it finds it, it transfers control of the program sequence to there, and begins executing those instructions. When it executes all of the instructions in the procedure, it finds the END statement of the procedure and returns to the next statement in the main program. When the main program finally runs out of things to do, it finds the END statement and terminates.

As the program executes the FOR loop, it is required to call the "WriteMessage" procedure 8 times, each time writing its message on the monitor, and finally it finds and executes the "WriteEnding" procedure. This should be very straightforward and should pose no real problem for you to understand. When you think you understand what it should do, compile and run it to see if it does.

NOW FOR A PROCEDURE THAT USES SOME DATA

The last program was interesting to show you how a procedure works but if you would like to see how to get some data into the procedure, load and display the program named PROCED2.MOD. We will once again go straight to the program starting in line number 25. We immediately notice that the program is nothing more than one big FOR loop which we go through 3 times. Each time through the loop we call several procedures, some that are system defined, and some that are user defined. This time instead of the simple procedure name, we have a variable in the parentheses behind the variable name. In these procedures, we will take some data with us to the procedures, when we call them, just like we have been doing with the "WriteString" and "WriteInt" procedures.

```
(* Chapter 5 - Program 2 *)
MODULE Proced2;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Stuff : INTEGER;
    Thing : INTEGER;

PROCEDURE PrintDataOut(Puppy : INTEGER);
BEGIN
   WriteString("The value of Puppy is     ");
   WriteInt(Puppy,5);
   WriteLn;
   Puppy := 12;
END PrintDataOut;

PROCEDURE PrintAndModify(VAR Cat : INTEGER);
BEGIN
   WriteString("The value of Cat is       ");
   WriteInt(Cat,5);
   WriteLn;
   Cat := 37;
END PrintAndModify;

BEGIN        (* Main program *)
   FOR Stuff := 3 TO 5 DO
      Thing := Stuff;
      PrintDataOut(Thing);
         WriteString("Back from print, data is  ");
         WriteInt(Thing,5);
         WriteLn;
      PrintAndModify(Thing);
         WriteString("Back from modify, data is ");
         WriteInt(Thing,5);
         WriteLn;
      PrintDataOut(Thing);
         WriteString("Back from print, data is  ");
         WriteInt(Thing,5);
         WriteLn;
         WriteLn;
   END;
END Proced2.
```

We will take some data to the procedure named "PrintDataOut" where it will be printed. The procedure "PrintDataOut" starting in line 9 also contains a pair of parentheses with a variable named "Puppy" which is of type INTEGER. This says that it is expecting a variable to be passed to it from the calling program and it expects the variable to be of type INTEGER. Back to the main program we see, on line 28, that the program did make the call to the procedure with a variable named "Thing" which is an INTEGER type variable, so everything is fine. The procedure prefers to call the variable passed to it "Puppy" but that is perfectly acceptable, it is the same variable. The procedure writes the value of "Puppy", which is really the variable "Thing" in the main program, in a line with an identifying string, then changes the value of "Puppy" before returning to the main program.

Upon returning to the main program, we print out another line with three separate parts (notice the indenting and the way it makes the program more readable), then calls the next procedure "PrintAndModify" which appears to do the same thing as the last one. Indeed, studying the procedure itself leads you to believe they are the same, except for the fact that this one prefers to use the name "Cat" for a variable name. There is one subtle difference in this procedure, the reserved word VAR in the header, line 17.

CALL BY VALUE & CALL BY REFERENCE

In the first procedure, the word VAR was omitted. This is a signal to the compiler that this procedure will not actually receive the variable reference, instead it will receive a local copy of the variable which it can use in whatever way it needs to. When it is finished, however, it can not return any changes in the variable to the main program because it can only work with its copy of the variable. This is therefore a one-way variable, it can only pass data to the procedure. This is sometimes called a "call by value" or a "value parameter" in literature about Modula-2.

In the second procedure, the word VAR was included. This signals the compiler that the variable in this procedure is meant to be actually passed to the procedure, and not just the value of the variable. The procedure can use this variable in any way it desires, and since it has access to the variable in the main program, it can alter it if it so desires. This is therefore a two-way variable, it can pass data from the main program to the procedure and back again. This is sometimes called a "call by reference" or a "variable parameter" in literature about Modula-2.

WHICH SHOULD BE USED?

It is up to you to decide which of the two parameter passing schemes you should use for each application. The "two-way" scheme seems to give the greatest flexibility, so your first thought is to simply use it everywhere. But that is not a good idea because it gives every procedure the ability to corrupt your main program variables. In addition, if you use a "call by value" in the procedure definition, you have the ability to call the procedure with a constant in that part of the call. A good example is given in lines 12, 20, 30, 34, and 38 of the present program. If "WriteInt" were defined with a "call by reference", we could not use a constant here, but instead would have to set up a variable, assign it the desired value, then use the variable name instead of the 5. There are other considerations but they are beyond the level of our study at this point.

BACK TO THE PROGRAM ON DISPLAY, PROCED2

We have already mentioned that both of the procedures modify their respective local variables, but due to the difference in "call by value" in the first, and "call by reference" in the second, only the second can actually get the modified data back to the calling program. This is why they are named the way they are. One other thing should be mentioned. Since it is not good practice to modify the variable used to control the FOR loop, (and downright erroneous in many cases) we make a copy of it and call it "Thing" for use in the calls to the procedures. Based on all we have said above, you should be able to figure out what the program will do, then compile and run it.

SEVERAL PARAMETERS PASSED AT ONCE

Load and display the program named PROCED3.MOD for an example of a procedure definition with more than one variable being passed to it. In this case four parameters are passed to this procedure. Three of the parameters are one-way and one is a two-way parameter. In this case we simply add the three numbers and return it to the main program. Good programming practice would dictate the placement of the single "call by reference" by itself and the others grouped together, but it is more important to demonstrate to you that they can be in any order you desire. This is a very straightforward example that should pose no problem to you. Compile and run it.

```
(* Chapter 5 - Program 3 *)
MODULE Proced3;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR Apple, Orange, Pear, Fruit : CARDINAL;

PROCEDURE AddTheFruit (Value1,Value2 : CARDINAL;  (* One-way *)
                       VAR Total     : CARDINAL;  (* Two-way *)
                       Value3        : CARDINAL); (* One-way *)
BEGIN
   Total := Value1 + Value2 + Value3;
END AddTheFruit;

BEGIN  (* Main Program *)
   Apple := 4;
   Orange := 7;
   Pear := 5;
   AddTheFruit(Apple,Pear,Fruit,Orange);
   WriteString("The total number of fruits is ");
   WriteCard(Fruit,5);
   WriteLn;
END Proced3.
```

SCOPE OF VARIABLES

Load and display the program PROCED4.MOD for a program which can be used to define scope of variables or where variables can be used in a program. The three variables defined in lines 6, 7, and 8, are of course available in the main program because they are defined prior to it. The two variables defined in the procedure are available within the procedure because that is where they are defined. However, because the variable "Count" is defined both places, it is two completely separate variables. The main program can never use the variable "Count" defined in the procedure, and the procedure can never use the variable "Count" defined in the main program. They are two completely separate and unique variables with no ties between them. This is useful because when your programs grow, you can define a variable in a procedure, use it in whatever way you wish, and not have to worry that you are corrupting some other "global" variable. The variables in the main program are called "global variables" because they are available everywhere.

```
(* Chapter 5 - Program 4 *)
MODULE Proced4;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR Count : CARDINAL;
    Index : CARDINAL;
    Other : CARDINAL;

PROCEDURE PrintSomeData;
VAR Count : CARDINAL;
    Apple : CARDINAL;
BEGIN
   Count := 7;
   Other := 12;
   Apple := 32;
   WriteString("In PrintSomeData the variables are");
   WriteCard(Index,5);
   WriteCard(Count,5);
   WriteCard(Other,5);
   WriteCard(Apple,5);
   WriteLn;
END PrintSomeData;

BEGIN    (* Main program *)
   FOR Index := 1 TO 3 DO
      Count := Index;
      Other := Index;
         WriteString("In Main Program the variables are ");
         WriteCard(Index,5);
         WriteCard(Count,5);
         WriteCard(Other,5);
         WriteLn;
      PrintSomeData;
         WriteString("In Main Program the variables are ");
         WriteCard(Index,5);
         WriteCard(Count,5);
         WriteCard(Other,5);
         WriteLn;
      WriteLn;
   END;  (* of FOR loop *)
END Proced4.
```

In addition to the above scope rules, the variable named "Apple" in the procedure, is not available to the main program. Since it is defined in the procedure it can only be used in the procedure. The procedure effectively builds a wall around the variable "Apple" and its own "Count" so that neither is available outside of the procedure. We will see in the next chapter that procedures can be "nested" leading to further hiding of variables. This program is intended to illustrate the scope of variables, and it would be good for you to study it, then compile and run it.

A PROCEDURE CAN CALL ANOTHER PROCEDURE

Load and display the program named PROCED5.MOD for an example of procedures that call other procedures. Study of this program will reveal that procedure "Three" starting on line 19 calls procedure "Two" which in turn calls procedure "One". The main program calls all three, one at a time, and the result is a succession of calls which should be rather easy for you to follow. The general rule is, "any program or procedure can call any other procedure that has been previously defined, and is visible to it." (We will say more about visibility later.) Study this program then compile and run it.

```
(* Chapter 5 - Program 5 *)
MODULE Proced5;

FROM InOut IMPORT WriteString, WriteLn;
```

```
PROCEDURE One;
BEGIN
   WriteString("This is procedure One.");
   WriteLn;
END One;

PROCEDURE Two;
BEGIN
   One;
   WriteString("This is procedure Two.");
   WriteLn;
END Two;

PROCEDURE Three;
BEGIN
   Two;
   WriteString("This is procedure Three.");
   WriteLn;
END Three;

BEGIN   (* Main program *)
   One;
   WriteLn;
   Two;
   WriteLn;
   Three;
   WriteLn;
END Proced5.
```

A FUNCTION PROCEDURE

Load and display the program named FUNCTION.MOD for an example of a "Function Procedure". This contains a procedure very much like the ones we have seen so far with one difference. In the procedure heading, line 6, there is an added ": INTEGER" at the end of the argument list. This is a signal to the system that this procedure is a "function procedure" and it therefore returns a value to the calling program in a way other than that provided for by parameter references as we have used before. In fact, this program returns a single data value that will be of type INTEGER.

```
(* Chapter 5 - Program 6 *)
MODULE Function;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

PROCEDURE QuadOfSum(Number1, Number2 : INTEGER) : INTEGER;
BEGIN
   RETURN(4*(Number1 + Number2));
END QuadOfSum;

VAR Dogs, Cats, Feet : INTEGER;
```

```
BEGIN  (* Main program *)
   Dogs := 4;
   Cats := 3;
   Feet := QuadOfSum(Dogs,Cats);
   WriteString("There are a total of");
   WriteInt(Feet,3);
   WriteString(" paws.");
   WriteLn;
END Function.
```

In line 16 of the calling program, we find the call to the procedure which looks like the others we have used except that it is used in an assignment statement as though it is an INTEGER type variable. This is exactly what it is and when the call is completed, the "QuadOfSum(Dogs,Cats)" will be replaced by the answer and then assigned to the variable "Feet". The entire call can therefore be used anyplace in a program where it is legal to use an INTEGER type variable. This is therefore a single value return and can be very useful in the right situation. In one of the earlier program, we used the "sin" and "cos" function procedures and this is exactly what they were.

One additional point must be made here. If a function procedure does not require any parameters, the call to it must include empty parentheses, and the definition of the procedure must include empty parentheses also. This is by definition of the Modula-2 language.

In the procedure, we had to do one thing slightly different in order to get the return value and that was to use the RETURN reserved word. Whenever we have completed the desired calculations or whatever we need to do, we put the result that is to be returned to the main program in the parentheses following the RETURN and the procedure will terminate, return to the calling program, and take the value with it as the answer. Due to decision making, we may have several RETURN statements in the procedure but only one will be exercised with each call. It is an error to come to the END statement of a function procedure since that would constitute a return without the benefit of the RETURN statement, and no value would be returned to the calling program.

WHAT IS RECURSION?

Recursion is simply a procedure calling itself. If you have never been introduced to recursion before, that definition sounds too simple but that is exactly what it is. You have probably seen a picture containing a picture of itself. The picture in the picture also contains a picture of itself, the end result being an infinity of pictures. Load the file named RECURSON.MOD for an example of a program with recursion.

```
(* Chapter 5 - Program 7 *)
MODULE Recurson;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR Count : INTEGER;

PROCEDURE PrintAndDecrement(Index : INTEGER);
BEGIN
   WriteString("The value of the Index is");
   WriteInt(Index,5);
   WriteLn;
   Index := Index - 1;
   IF Index > 0 THEN
      PrintAndDecrement(Index);
   END;
END PrintAndDecrement;

BEGIN    (* Main program *)
   Count := 7;
   PrintAndDecrement(Count);
END Recurson.
```

In the main program, "Count" is set to 7 and the procedure is called taking along "Count" as a parameter. In the procedure, we display a line containing the value of the variable, now called "Index", and decrement it. If the variable is greater than zero, we call the same procedure again, this time entering it with the value of 6. It would be reasonably correct to think of the system as creating another copy of the procedure for this call. The variable "Index" would be reduced to 5, and another copy of the procedure would be called. Finally, the variable would be reduced to zero and the return path from procedure to procedure would be taken until the main program would be reached, where the program would terminate.

Rather than making a complete new copy of the procedure for each recursive call, the same code would be run each time through and all of the data would be stored away on the "stack" each time through. You have no need to worry about this because it is all taken care of for you by the system. You simply call the same procedure as though it were any other procedure and the system will worry about all of the details except for one. It is up to you to see that there is some mechanism by which the process will terminate. If there were no decrementing statement in the procedure, this program would never reach an end and the stack would overflow, resulting in an error message and termination of the program. It would be worth your time to remove the decrementing statement and observe this, after you compile and run it the way it is now.

Recursion can be very useful for those problems that warrant its use. This example is a very stupid use of recursion, but is an excellent method for giving an example of a program with recursion that is simple and easy to understand.

DIRECT AND INDIRECT RECURSION

This example uses direct recursion because the procedure calls itself directly. It is also possible to use indirect recursion where procedure "A" calls "B", "B" calls "A", etc. Either method is available and useful depending on the particular circumstances.

PROGRAMMING EXERCISES

1. Write a program to write your name, address, and phone number on the monitor with each line in a different procedure.

2. Add a statement to the procedure in RECURSON to display the value of "Index" after the call to itself so you can see the value increasing as the recurring calls are returned to the next higher level.

3. Rewrite TEMPCONV from chapter 4 putting the centigrade to farenheit formula in a function procedure.

Chapter 6 - Arrays, Types, and Constants

Load the program named ARRAYS.MOD and we will go right to our first example of an array. An array is simply a list made up of several of the same type of element. Notice the VAR definition in the sample program and specifically the variable named "Automobiles". The reserved word ARRAY followed by the square brackets with a range of numbers contained within them is the proper way to define an array of, in this case, CARDINAL type variables. This defines 12 different CARDINAL type variables, each of which is capable of storing one CARDINAL number. The names of the twelve variables are given by Automobiles[1], Automobiles[2], ... Automobiles[12]. The variable name is "Automobiles" and the array subscripts are the numbers 1 through 12. The variables are true CARDINAL type variables and can be assigned values, or they can be used in calculations or in nearly anyplace in a program where it is legal to use a CARDINAL type variable. One place they cannot be used is as the index for a FOR loop since a simple variable type is required there.

```
(* Chapter 6 - Program 1 *)
MODULE Arrays;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR Index       : CARDINAL;
    Automobiles : ARRAY [1..12] OF CARDINAL;

BEGIN   (* main program *)
   FOR Index := 1 TO 12 DO
      Automobiles[Index] := Index + 10;
   END;
   Automobiles[7] := 54;  (* example, change one value of array *)
   WriteString("This is the first program with an array.");
   WriteLn;
   WriteLn;                       (* end of data initialization *)

   FOR Index := 1 TO 12 DO              (* display the data now *)
      WriteString("Automobile number");
      WriteCard(Index,3);
      WriteString(" has the value of");
      WriteCard(Automobiles[Index],3);
      WriteLn;
   END;
END Arrays.
```

WHAT GOOD ARE ARRAYS?

Notice lines 10 through 12 of the program. In these lines, each of the 12 variables is assigned a value. When "Index" is 1, then "Automobiles[1]" is assigned 11, then when "Index" is 2, "Automobiles[2]" is assigned 12, etc.

If the 12 variables were defined as 12 separate variables of whatever names we chose for them, we could not assign them values in a loop but would have to assign each one independently. In this instance, we are generating nonsense data but in a real program, this loop could be reading in a series of data from a file such as would be done with a database. The advantage of the array should be very clear, especially if we were to change the array limits to several thousand elements.

The statement in line 13 assigns a value to one of the elements at random to illustrate the method. Notice that the 7th element of the array named "Automobiles" is assigned the value of 54. The address of this data is therefore the variable name "Automobiles[7]" and the data contained in that address is 54. We have therefore assigned values to the 12 variables by a nonsensical but known scheme, and now we can use the 12 variables in any way that is legal within Modula-2.

The next loop causes the 12 values to be displayed on the monitor in a neat orderly fashion. In line 20 we display the index of the variable in question, and in line 22, we display the actual variable. Keep in mind that the

index could have been INTEGER and still be used to display an array of type CARDINAL provided we defined "Index" as an integer and always used it as such. Spend enough time with this program so that you thoroughly understand it, then compile and run it.

WHAT ABOUT AN ILLEGAL SUBSCRIPT?

Modula-2 does very strong "type checking" and limit checking. If, in the above program, you tried to assign a value to "Automobiles[13]", which doesn't exist, a run time error would be generated and the program would cease. This is one of the advantages of Modula-2 over some of the older programming languages. Some compilers have the ability to enable or disable this feature.

MULTIPLY DIMENSIONED ARRAYS

Load the file named ARRAYS2.MOD for an example of a program with two-dimensional arrays. In this program, the VAR section contains the "Checkerboard" variable which is defined as an 8 element ARRAY in which each element is an 8 element ARRAY, therefore being an 8 by 8 square ARRAY. Each element is capable of storing one CARDINAL type variable. The variable "Value" is defined the same way except that the method of definition is slightly different. The two methods result in the same type and number of variables.

```
(* Chapter 6 - Program 2 *)
MODULE Arrays2;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR  Index, Count      : CARDINAL;
     Checkerboard      : ARRAY[1..8] OF ARRAY[1..8] OF CARDINAL;
     Value             : ARRAY[1..8],[1..8] OF CARDINAL;

BEGIN
   FOR Index := 1 TO 8 DO
      FOR Count := 1 TO 8 DO
         Checkerboard[Index,Count] := Index + 3*Count;
         Value[Index,Count] := Index + 2*Checkerboard[Index,Count];
      END;  (* of Count loop *)
   END;     (* of Index loop *)

   WriteString("Output of checkerboard");
   WriteLn;
   FOR Index := 1 TO 8 DO
      FOR Count := 1 TO 8 DO
         WriteInt(Checkerboard[Index,Count],6);
      END;  (* of Count loop *)
      WriteLn;
   END;     (* of Index loop *)

   Value[3,5] := 77;   (* change a few of the values *)
   Value[3,6] := 3;
   Value[Value[3,6],7] := 2;  (* same as Value[3,7] := 2; *)

   WriteLn;
   WriteString("Output of Value matrix");
   WriteLn;
   FOR Count := 1 TO 8 DO
      FOR Index := 1 TO 8 DO
         WriteInt(Value[Count,Index],6);
      END;   (* of Index loop *)
      WriteLn;
   END;      (* of Count loop *)
END Arrays2.
```

In lines 11 through 16 we have two nested FOR loops. The outer loop causes "Index" to count from 1 to 8 and for each value of "Index", the variable "Count" counts through the values 1 to 8 also. The net result is that we evaluate the assignments in lines 13 and 14 once for each possible combination of "Index" and "Count". For each combination, we assign some nonsense data to "Checkerboard" then use the result of that calculation to assign some nonsense data to the variable "Value". The purpose here is to illustrate the method of using the double subscripted variables. Next we display the entire matrix of "Checkerboard". The loops cause 8 values to be displayed on one line so that the entire matrix is displayed on only 8 lines. You should study this logic because you will find output sequences like this to be very valuable.

CHANGING A FEW OF THE VALUES

In line 27 and following we change a few of the values at random for illustrative purposes. Since "Value[3,6]" is assigned the value of 3, it can be used as one of the subscripts of the next line and in fact it is. This would be a rather sloppy programming style but it is a good illustration of what can be done. Finally using the same technique as that for "Checkerboard", the "Value" matrix is displayed.

HOW MANY SUBSCRIPTS CAN BE USED?

There is no limit as to how many subscripts can be used in Modula-2 by definition, but there is a practical limit of somewhere in the range of 3 or 4. If you use too many, you will very quickly get confused and lose control of what the program is supposed to be doing. I have never seen more than 3 subscripts used in any programming language, and very few instances of more than two. Let the problem definition be your guide.

This program was pretty straightforward, and it is time for you to compile and run it.

THE TYPE DECLARATION

Load the program named TYPES.MOD for a new topic that you will use often, especially in large programs. At the top of the listing we have a group of TYPE declarations. The first line defines "ArrayDef" as a new TYPE definition that can be used in the same way you would use INTEGER or any of the other simple type definitions. In line 12, the variable named "Stuff" is defined as a variable of type "ArrayDef", and since "ArrayDef" is a 14 element ARRAY, then "Stuff" is a 14 element array of INTEGER. It seems like we didn't save anything and in fact we added a few keystrokes to the program in order to do this. If you look at line 13 you will see that we have also defined "Stuff2" as the same type of array. We have, in fact, defined them to be "type compatible" which will be very important when we get to the program itself.

```
(* Chapter 6 - Program 2 *)
MODULE Arrays2;

FROM InOut IMPORT WriteString, WriteInt, WriteLn;

VAR  Index, Count     : CARDINAL;
     Checkerboard     : ARRAY[1..8] OF ARRAY[1..8] OF CARDINAL;
     Value            : ARRAY[1..8],[1..8] OF CARDINAL;

BEGIN
   FOR Index := 1 TO 8 DO
      FOR Count := 1 TO 8 DO
         Checkerboard[Index,Count] := Index + 3*Count;
         Value[Index,Count] := Index + 2*Checkerboard[Index,Count];
      END;  (* of Count loop *)
   END;     (* of Index loop *)

   WriteString("Output of checkerboard");
   WriteLn;
   FOR Index := 1 TO 8 DO
      FOR Count := 1 TO 8 DO
         WriteInt(Checkerboard[Index,Count],6);
      END;  (* of Count loop *)
      WriteLn;
   END;     (* of Index loop *)

   Value[3,5] := 77;   (* change a few of the values *)
   Value[3,6] := 3;
   Value[Value[3,6],7] := 2;  (* same as Value[3,7] := 2; *)

   WriteLn;
   WriteString("Output of Value matrix");
   WriteLn;
   FOR Count := 1 TO 8 DO
      FOR Index := 1 TO 8 DO
         WriteInt(Value[Count,Index],6);
      END;   (* of Index loop *)
      WriteLn;
   END;      (* of Count loop *)
END Arrays2.
```

Continuing down the list of TYPE declarations, we define a TYPE with 28 characters, then a TYPE with 60 real variables, and another with 6 BOOLEAN variables. The next TYPE consists of 12 variables of TYPE "DogFood" which is itself a TYPE of 6 BOOLEANS, resulting in a TYPE consisting of 6 times 12 = 72 BOOLEANS. It is possible to continue building up TYPE definitions like this indefinitely, and as you build up applications, you will find yourself building up rather complex TYPE declarations and having a clear picture of how they go together because it is your solution to a problem. The last TYPE to be defined is that named "Boat" which has exactly the same size and characteristics as "Airplane". We will see shortly that there is a difference in these two definitions.

HOW DO WE USE ALL OF THIS?

In the VAR part of the definition part of the program, we declare some variables, two simple types and some of the types we defined above. In the program part, we assign some values to the 72 variables making up the "Puppies" matrix and the 72 variables making up the "Kitties" matrix. All of the elements of "Stuff" are then assigned nonsense values. The really interesting statement comes in line 30 where we say "Stuff2 := Stuff;". In this simple statement, all 14 values stored in "Stuff" are copied into the 14 corresponding elements of "Stuff2" without using a loop. This is possible because the two variables are TYPE compatible, they have the same TYPE definition. If you study the definitions above, you will see that "Stuff3" is of the same number and range of elements and is composed of the same type of elements, namely INTEGER, as "Stuff" but they are not TYPE compatible because were not defined with the same TYPE definition statement. In like manner, even though "Puppies" and "Kitties" are identical in type, they are not TYPE compatible.

You have the ability, through careful assignment of variables, to avoid certain kinds of programming errors. If certain variables should never be assigned to each other, a careful selection of types can prevent it. Suppose for example that you have a program working with peaches and books. You would never want to copy a matrix of peaches to one defining books, it just wouldn't make sense. Those two matrices should be defined with different type declarations even though they may be identical in size.

Compile and run this program, even though it will result in no output, then move the comment delimiter in line 31 to a position following the assignment statement and see if it does give you a TYPE incompatibility error.

DEFINING A CONSTANT

Load the program named CONSTANT.MOD for a definition of the constant as used in Modula-2. We will finally keep the promise made when we studied LOOPDEMO in chapter 4. The new reserved word CONST is used to define a constant for use in the program. The constant "MaxSize" can be used anywhere in the program that it is desired to use the number 12, because they are in fact identical. Two additional CONST values are defined for illustrative purposes only. In the TYPE declaration section we use the constant "MaxSize" to define two types, then use them to define several variables.

```
(* Chapter 6 - Program 4 *)
MODULE Constant;

CONST MaxSize    = 12;
      IndexStart = 49;
      CheckItOut = TRUE;

TYPE  BigArray  = ARRAY[1..MaxSize] OF CARDINAL;
      CharArray = ARRAY[1..MaxSize] OF CHAR;

VAR   AirPlane   : BigArray;
      SeaPlane   : BigArray;
      Helicopter : BigArray;
      Cows       : CharArray;
      Horses     : CharArray;
      Index      : CARDINAL;

BEGIN     (* main program *)
   FOR Index := 1 TO MaxSize DO
      AirPlane[Index] := Index*2;
      SeaPlane[Index] := Index*3 + 12;
      Helicopter[MaxSize - Index + 1] := Index + AirPlane[Index];
      Horses[Index] := 'X';
      Cows[Index] := "R";
   END;   (* of Index loop *)
END Constant.
```

In the program there is one FOR loop using the same constant "MaxSize" as the upper limit. It doesn't seem to be too useful yet, but suppose your boss came to you and said to change the program so that it handled 142 cases instead of 12. The way the program is written, you would only have to change the value of the constant, recompile, and you would be done. If you had used the number 12 everywhere, you would have to replace every 12 with the new number, 142, being careful not to change the one in line 21 which is a different kind of 12. Of course even that would not be too difficult in such a simple program, but in a program with 5000 lines of code, one simple change could take a week.

Compile and run this program.

THE OPEN ARRAY IN A PROCEDURE

Load and display the program named ARAYPASS.MOD for an example of a program with arrays being passed to a procedure. Notice how the procedures are formatted. The rows of asterisks make them really stand out and easy to find. You will develop your own personal style of formatting in a way that is clear and easy to follow for you.

```
(* Chapter 6 - Program 5 *)
MODULE ArayPass;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

TYPE OneArray = ARRAY[10..15] OF CARDINAL;
     TwoArray = ARRAY[-8..210] OF CARDINAL;

VAR  SizeOne : OneArray;
     SizeTwo : TwoArray;
     Index   : INTEGER;

(* *************************************************** AddNumbers *)
PROCEDURE AddNumbers(Donkey : OneArray);

VAR CountUp, Sum : CARDINAL;

BEGIN
   Sum := 0;
   FOR CountUp := 10 TO 15 DO
      Sum := Sum + Donkey[CountUp];
   END;
   WriteCard(Sum,5);
   WriteLn;
END AddNumbers;

(* ********************************************* GenAddNumbers *)
PROCEDURE GenAddNumbers(Donkey : ARRAY OF CARDINAL);

VAR CountUp, Sum : CARDINAL;

BEGIN
   Sum := 0;
   FOR CountUp := 0 TO HIGH(Donkey) DO
      Sum := Sum + Donkey[CountUp];
   END;
   WriteCard(Sum,5);
   WriteLn;
END GenAddNumbers;

BEGIN      (* ************************************** main program *)
   FOR Index := 10 TO 15 DO
      SizeOne[Index] := 10;
   END;

   FOR Index := 210 TO -8 BY -1 DO
      SizeTwo[Index] := 1;
   END;

   WriteString("The sum of the SizeOne numbers is");
   AddNumbers(SizeOne);

   WriteString("Gen sum of the SizeOne numbers is");
   GenAddNumbers(SizeOne);
   WriteString("Gen sum of the SizeTwo numbers is");
   GenAddNumbers(SizeTwo);

END ArayPass.
```

The two procedures in this program are identical except for the way the arrays are passed to them. In the first procedure named "AddNumbers", the variable named "Donkey" is passed the array by using the same type which was used to define one of the arrays. The procedure merely adds the values of the elements of the array passed to it and writes the result out to the monitor. The way it is written, it is only capable of adding arrays that are indexed from 10 to 15. Any other array will cause a "type incompatible" error. This is simply called passing an array to the procedure.

The second procedure named "GenAddNumbers" has its input array defined as an "ARRAY OF CARDINAL" with no limits stated. This procedure can add all of the variables in any CARDINAL array regardless of the range of its subscripts. The lower subscript will always be defined as zero within this type of procedure, and the upper limit of the array can be found with the predefined procedure "HIGH". It is used as shown in the example. The first time this procedure is called in the main program, it is called with the variable "SizeOne". In the procedure, the array subscripts for "Donkey" will be 0 through 5. When the variable named "SizeTwo" is the array sent to the procedure, then "Donkey" will have the limits of 0 and 218. The second procedure definition method is therefore more general. This is called passing an "open array" to the procedure.

WHICH ONE SHOULD I USE?

There will be times when you wish to use the general case for passing a parameter, the "open array". A good example is the procedure named "WriteString" that we have been using in this tutorial. It would be a bit cumbersome if we were only allowed to pass a 10 character string to it each time. Since it can accept a string of any length, it is evidently defined with an "ARRAY OF CHAR" in its header. (We will see in a later chapter that this particular procedure is exactly that, a procedure that someone has thoughtfully programmed for you. You only need to tell the system where it can be found using the IMPORT statement.)

There will likewise be times when you will desire to use the more specific method of definition. If you are using a lot of arrays and have a specific operation that needs to be done to only a few arrays that have a common definition, you would be wise to use this method. The computer could then tell you if you tried to use the procedure on an array that it was not intended for. This is making wise use of the type checking available in the computer.

HANDLING STRINGS IN MODULA-2

Load the last file for this chapter, STRINGEX.MOD for an example of using strings in Modula-2. This program is the first program to deviate from the standard library as defined by Niklaus Wirth. When he defined the language, he suggested several library procedures that should be available in every Modula-2 compiler and most compiler writers have followed his suggestions quite closely. He failed to define a standard library for the string handling procedures. There is therefore some freedom for each compiler writer to define the string handling routines in any way he pleases. Most however, have followed at least a resemblance to a standard, so the procedure calls are very similar from compiler to compiler. It may be necessary for you to modify this file to suit your particular compiler. The COMPILER.DOC file on your distribution disk has comments for modifications needed for several compilers, but if yours is not listed, it will be up to you to make the required modifications to the source file.

```
(* Chapter 6 - Program 6 *)
MODULE StringEx;

(* Note - The "Strings" procedures used here are not standard because
          there is no standard.  You may need to modify some or all
          of the string procedure calls to get them to work.  Consult
          the documentation for your compiler and library.         *)

FROM InOut   IMPORT WriteString, WriteInt, WriteLn;
FROM Strings IMPORT Assign, Concat;

TYPE SevenChar = ARRAY[0..6] OF CHAR;

VAR Horse : ARRAY[0..12] OF CHAR;
    Cow   : ARRAY[0..5] OF CHAR;
    S1    : SevenChar;
    S2    : SevenChar;
    Index : CARDINAL;

(* ***************************************************** Display *)
PROCEDURE Display(Stuff : ARRAY OF CHAR);
BEGIN
   WriteString("Array(");
   WriteString(Stuff);
   WriteString(") - ");
   FOR Index := 0 TO HIGH(Stuff) DO
      WriteInt(ORD(Stuff[Index]),4);
   END;
   WriteLn;
END Display;

(* ************************************************ main program *)
BEGIN
   Horse := "ABCDEFGHIJKL";           (* Copy constant to variable *)
   Display(Horse);

   Cow := "12345";
   Assign(Cow,Horse);               (* Assign variable to variable *)
   Display(Horse);

   S1 := "Neat";
   S2 := "Things";
   Concat(S1,S2,Horse);       (* Concatenate variables to variable *)
   Display(Horse);
   S1 := S2;                        (* Assign variable to variable *)

   Concat(Horse,Cow,Horse); (* Concatenate one variable to another *)
   Display(Horse);

   Concat(Cow,Horse,Horse);        (* Concatenate to the beginning *)
   Display(Horse);
END StringEx.
```

A complete description of the libraries and what they are will be given in chapter 8.

BACK TO THE PROGRAM ON YOUR DISPLAY

The first thing that is different here is the addition of another IMPORT statement in line 10, this one importing procedures from the module named "Strings". This is the module containing the procedures which we will need in this program. A string is an array of type CHAR, each element of the array being capable of storing one character. Thus an array of CHAR type elements is capable of storing a word, a sentence, a paragraph, or even a whole chapter, depending on how big the array is. Using the example on your screen, we will learn how to manipulate text data.

One additional feature of the example program will be found on line 24. In this line the "WriteString" procedure is used in a way we have not used as yet. Instead of having an expression in quotes, it has the name of a variable within its parentheses. It will display whatever characters are stored in the string named "stuff" defined by the "ARRAY OF CHAR". So if we learn how to get a string of characters stored in a variable of type "string", we can display anything on the monitor that we can generate internal to the computer.

According to the definition of Modula-2, a string is an ARRAY OF CHAR with a 0 as a terminator. We will get more familiar with strings as we continue our study.

SOME NEW STRING PROCEDURES

The first line of the program itself, line 34, contains a string assignment. In this case, we are telling the system to copy the constant "ABCDEFGHIJKL" into the variable named "Horse". The array into which you are copying must begin at index 0 in order for this to work because all character constants are, by definition, started at zero. The variable "Horse", which only contains room for 12 characters will only receive the first 12 characters of the constant. The procedure "Display" is called with Horse as the variable and the variable is displayed between parentheses for clarity of understanding, and the 12 characters of the variable are displayed in their ASCII equivalent. When you finally run this program, compare some of the values to the ASCII table that is included with the DOS documentation that came with your computer.

In line 37 of the program, the constant "12345" is assigned to the variable "Cow". In the next line, the variable "Cow" is assigned to the variable "Horse", and the display procedure is called again. This time, the variable "Cow" is shorter than the destination, so the system has to compensate for the difference. After it transfers the 5 characters to

"Horse", it will place a 0 (zero) in the next position to indicate the end of the string. The definition of the string still has 12 places, but there are only 5 places of interest, so the system will consider all places past the 5th as undefined. This time the system only prints out 5 characters in the procedure. The list of ASCII equivalents shows that the other values are still there, the output routine simply stopped when it came to the 0 in the sixth position.

Note that the Assign statement may be different for different compilers because it is not a part of the Modula-2 definition by Niclaus Wirth.

CONCATENATION

Concatenation is simply putting two strings together to make up one bigger string. Beginning in line 41, two new string variables are defined, "S1" and "S2", then the two new variables are concatenated together and assigned to our old favorite variable named "Horse". The variable "Horse" should now contain the silly expression "NeatThings", and when you run the program, you will find that it does. It also has a 0 in character position 11 now to indicate the end of the string. Line 47 concatenates "Horse" to "Cow" and stores the result in "Horse", but since the expression is now too long, part of it will get truncated and simply thrown away. Finally, "Cow" is concatenated to "Horse", and the result stored back into "Horse". This has the effect of shifting the prior contents of "Horse" right and adding the characters stored in "Cow" to the beginning. Line 45 is an example of a string assignment. This is only possible because they are of the same TYPE. The variable "Cow" has a different TYPE so can't be assigned to either of these two variables. Note that the TYPE does not have to start at zero for this to work.

Note that even though "Horse" was the only variable used in the calls to "Display", any of the other strings could have been used also. This is the topic of the fourth programming exercise below.

Compile and run the program and see if it really does do all that it should do as described above, keeping in mind that you may have to modify the file to accommodate your particular compiler.

PROGRAMMING EXERCISES

1. Write a program to store the CARDINAL values 201 to 212 in an array then display

them on the monitor.

2. Write a program to store a 10 by 10 array containing the products of the indices, therefore a multiplication table. Display the matrix on the monitor.

3. Modify the program in 2 above to include a constant so that by simply changing the constant, the size of the matrix and the range of the table will be changed.

4. Modify the program named STRINGEX.MOD to include calls to "Display" with each of the string variables at the end of the program.

Chapter 7 - Overall Program Construction

We have pretty well covered the topic of how to put all the parts together to build up a program. In this chapter we will go over the whole process in order to clear up any loose ends and have the entire process in one place. There is nothing magic about the way the various pieces fit together but the rules must be followed in order to build a usable program.

Load and display the program named OVERPROG.MOD for the first look at the overall structure of the Modula-2 program. It would be well for you to keep in mind that there is a major category that we have not even hinted at yet, the issue of modules. They will be covered in Part III of this tutorial, and although they are very important, you can begin writing meaningful programs before you even hear what modules are or how they are used.

```
(* Chapter 7 - Program 1 *)
MODULE OverProg;     (* Overall program construction example *)

FROM InOut IMPORT WriteString, WriteLn;

     PROCEDURE Proc1;
     BEGIN
       WriteString("Procedure 1");
       WriteLn;
     END Proc1;

     PROCEDURE Proc2;
          PROCEDURE Proc3;
          BEGIN
            WriteString("Procedure 3");
            WriteLn;
          END Proc3;

          PROCEDURE Proc4;
               PROCEDURE Proc5;
               BEGIN
                 WriteString("Procedure 5");
                 WriteLn;
               END Proc5;
          BEGIN
            WriteString("Procedure 4");
            WriteLn;
            Proc5;
            Proc3;
          END Proc4;
     BEGIN
       WriteString("Procedure 2");
       WriteLn;
```

```
        Proc3;
        Proc4;
      END Proc2;

BEGIN
  WriteString("Main Program");
  WriteLn;
  Proc2;
  Proc1;
END OverProg.
```

NESTED PROCEDURES

The program on display contains several levels of nested procedures as an illustration for you. The main program has lines 1 through 37 as its declaration part, and lines 38 through 43 as its statement part. Since the procedure definitions actually define the procedures called for by the main program, they correctly belong in the declaration part of the program. Only two of the procedures are actually callable by the main program, "Proc1", and "Proc2". The procedure "Proc1" is a simple procedure, but "Proc2" has additional procedures in its declaration part.

Procedure "Proc2" contains a declaration part in lines 13 through 30, and a statement part in lines 31 through 36. Its declaration part contains two procedures, "Proc3" and "Proc4". The nesting is carried one step farther in "Proc4" which contains the procedure "Proc5" in its declaration part. Procedures can be nested to whatever level desired according to the definition of Modula-2.

WHO CAN CALL WHO?

It is important for you to clearly understand which procedure can call which other procedures. A procedure can call any procedure on the same level as itself provided that both have the same parentage, or any procedure that is included in its own declaration part at the level of its own declaration part. For example, the main program can only call "Proc1", and "Proc2". The others are nested within "Proc2" and are not available to the main program. Likewise the statement part of "Proc2" can call "Proc1", because it is on the same level, and "Proc3" and "Proc4", because they are within its declaration part. The procedure "Proc5" can only be called by "Proc4", because no other procedure is at its level. Note that if another triple nesting were included in "Proc1", its third level procedure could not be called by "Proc5" because they would not have the same parentage.

Nested procedures can be very useful when you wish to use a procedure that you don't want any other part of the program to be able to access or even see. A private procedure can therefore be written with no concern that the name may clash with some other part of the program and cause undesirable effects.

The important thing to gain from this program is that nesting is possible and can be very useful, and the definition of a procedure is the same as that of the main program. This means that procedures can be nested within procedures in any way that aids in designing the program.

Compile and run this program and see if you understand the output from it.

WHERE DO WE PLACE CONSTANTS, TYPES, AND VARIABLES?

Load MOREPROG.MOD, for examples of where you can put the other definitions in the declaration part of the program and the procedures. This is a repeat of the last program with CONST, TYPE, and VAR declarations added in every place where it is legal to put them. This is done as an example to you of where they can be put, so no explanation of details will be given. Some time spent studying this program should aid you in understanding even better the overall program construction problem.

```
(* Chapter 7 - Program 2 *)
MODULE MoreProg;   (* More program construction examples *)

FROM InOut IMPORT WriteString, WriteLn;

CONST MainC = 27;
TYPE  MainT = ARRAY[3..7] OF CARDINAL;
VAR   MainV : MainT;

     PROCEDURE Proc1;
     CONST Proc1C = 33;
     TYPE  Proc1T = ARRAY[-23..-15] OF CHAR;
     VAR   Proc1V : MainT;
           Proc11 : Proc1T;
     BEGIN
       WriteString("Procedure 1");
       WriteLn;
     END Proc1;

     PROCEDURE Proc2;
     CONST Proc2C = 22;
     TYPE  Proc2T = ARRAY[3..5],[-4..0] OF BOOLEAN;
     VAR   Proc2V : MainT;
           Proc21 : Proc2T;
```

```
          PROCEDURE Proc3;
          CONST Proc3C = -234;
          TYPE  Proc3T = ARRAY[12..13] OF MainT;
          VAR   Proc3V : MainT;
                Proc31 : Proc2T;
                Proc32 : Proc3T;
          BEGIN
            WriteString("Procedure 3");
            WriteLn;
          END Proc3;

          PROCEDURE Proc4;
          CONST Proc4C = 111;
          TYPE  Proc4T = CARDINAL;
          VAR   Proc4V : MainT;
                Proc41 : Proc2T;
                Proc42 : Proc4T;
               PROCEDURE Proc5;
               CONST Proc5C = "A";
               TYPE  Proc5T = ARRAY[22..222] OF CHAR;
               VAR   Proc5V : MainT;
                     Proc51 : Proc2T;
                     Proc52 : Proc4T;
                     Proc53 : Proc5T;
               BEGIN
                 WriteString("Procedure 5");
                 WriteLn;
               END Proc5;
          BEGIN
            WriteString("Procedure 4");
            WriteLn;
            Proc5;
            Proc3;
          END Proc4;
     BEGIN
       WriteString("Procedure 2");
       WriteLn;
       Proc3;
       Proc4;
     END Proc2;

BEGIN
  WriteString("Main Program");
  WriteLn;
  Proc2;
  Proc1;
END MoreProg.
```

WHAT ABOUT ORDER OF DECLARATIONS?

Load the program LASTPROG.MOD for an example of how the various fields can be ordered in the declaration part of the program. Notice that there are 2 procedures, two CONST's, two TYPE's, and two VAR's defined, but they are defined in a seemingly random order. The order is random and was done only to illustrate to you that the order doesn't matter as long as everything is defined before it is used.

```
(* Chapter 7 - Program 3 *)
MODULE LastProg;

FROM InOut IMPORT WriteString, WriteLn;

   PROCEDURE Dummy;
   BEGIN
   END Dummy;

VAR Index : CARDINAL;
TYPE Stuff = ARRAY[34..55] OF CHAR;

   PROCEDURE Smart;
   BEGIN
   END Smart;

CONST Number = 120;
TYPE  TypeOf = ARRAY[1..Number] OF Stuff;
CONST Neater = -345;
VAR   Count23 : TypeOf;
      Counter : INTEGER;

BEGIN
   WriteString("This is really a stupid program.");
   WriteLn;
END LastProg.
```

In only one case does the order matter. The compiler is very picky about where the IMPORT list goes because the Modula-2 language definition requires it to be first. In addition, the EXPORT list must immediately follow the IMPORT list. We will cover both of these in detail later, for now simply remember that the order of all declarations can come in random order as long as they follow the IMPORT/EXPORT lists and come before the statement part of the program.

PROGRAMMING EXERCISES

1. Using the program OVERPROG, add some calls to illegal places to see what messages the compiler displays.

2. Using the program MOREPROG, add some illegal variable references to see what messages the compiler displays.

Chapter 8 - Input/Output

A SIMPLE OUTPUT PROGRAM

Load and display the file named SIMPLOUT.MOD for an example of the simple output functions. This program is limited to writing only to the monitor but we will get to files and printer output shortly. We must first establish some basic principles for use with library procedures.

```
(* Chapter 8 - Program 1 *)
MODULE SimplOut;

FROM InOut IMPORT WriteString, WriteLn;         (* unqualified *)
IMPORT InOut;          (* This imports every procedure in InOut *)
IMPORT Terminal;    (* This imports every procedure in Terminal *)

VAR Index : CARDINAL;

BEGIN
   WriteString("This is from InOut, ");
   InOut.WriteString("and so is this.");
   Terminal.WriteLn;

   Terminal.WriteString("This is from Terminal, ");
   Terminal.WriteString('and so is this.');
   WriteLn;

   FOR Index := 1 TO 10 DO
      InOut.WriteCard(Index,5);
   END;
   InOut.WriteLn;
END SimplOut.
```

The first line of the declaration part of the program imports our two familiar procedures "WriteString" and "WriteLn" in the same manner we are used to. The next line imports every procedure in "InOut" and makes them available for use in the program without specifically naming each one in the IMPORT list. The third line imports every procedure from "Terminal" so that they too are available for our use. The procedures that are imported explicitly can be used in exactly the same manner that we have been using them all along, simply name the procedure with any arguments they use. The others can only be used with a "qualifier" that tells which library module they come from.

An example is the easiest way to describe their use so refer to the program before you. Line 11 uses the explicitly defined procedure from "InOut", line 12 uses the same procedure from "InOut", and line 15 uses the procedure of the same name from "Terminal". Line 11 uses the unqualified procedure call from "InOut", and lines 12 and 15 use the qualified method of calling the procedures from both library modules.

In this case, the two procedures do the same thing, but it is not required that procedures with the same name do the same thing. By adding the library module name to the front of the procedure name with a period between them, we tell the system which of the two procedures we wish to use. If we tried to explicitly import both "WriteString" procedures we would get a compile error, so this is the way to use the same name twice.

WHAT IS A LIBRARY MODULE?

What I have been calling a library module is more properly termed a "module" and is the biggest benefit that Modula-2 enjoys over other programming languages. This is the quality that gives Modula-2 the ability to have separately compiled modules, because a module is a compilation unit. When you get to Part III of this tutorial, you will learn how to write your own modules containing your own favorite procedures, and call them in any program in the same manner that you have been calling the procedures provided by your compiler writer.

None of the procedures you have been importing are part of the Modula-2 language, they are extensions to the language provided for you by your compiler writer. Since they are not standard parts of the language, they may vary from compiler to compiler. For that reason, I have tried to use those defined by Niklaus Wirth in his definition of the language, and no others.

STUDY YOUR REFERENCE MANUAL

This would be a good place for you to stop and spend some time reading your reference manual. Look up the section in your manual that is probably called the "Library" and read through some of the details given there. You will find that there are many things listed there that you will not understand at this point, but you will also find many things there that you do understand. Each module will have a number of procedures that are "exported" so that you can "import" them and use them. Each procedure will have a definition of what arguments are required in order to use it. Most of these definitions should be understandable to you. One thing you will find is that only the "PROCEDURE procname;" is given along with the arguments, with the actual code of the procedure omitted. We will study about this in "Part III" also. The part that is shown is the "DEFINITION MODULE" which only gives the calling requirements. The "IMPLEMENTATION MODULE" which gives the actual program code of the procedure is usually not given by compiler writers.

As you study the library modules, you will find procedures to handle strings, variables and conversions between the two. You will find "mouse" drivers, "BIOS" calls to the inner workings of your operating system, and many other kinds of procedures. All of these procedures are available for you to use in your programs. They have been written, debugged, and documented for your use once you learn to use them. In addition, you will have the ability to add to this list by creating your own modules containing your own procedures.

BACK TO THE PROGRAM "SIMPLOUT"

Notice that in lines 13, 17, and 22, three different ways are used to call "WriteLn", even though there are actually only two procedures (that happen to do the same thing). A little time spent here will be time well spent in preparing for the next few programs. When you think you understand this program, compile and run it.

NOW FOR SOME SIMPLE INPUT PROCEDURES

Load the program named SIMPLIN.MOD for our first example of a program with some data input procedures. In every program we have run so far in this tutorial, all data has been stored right in the program statements. It would be a very sad computer that did not have the ability to read variable data in from the keyboard and files. This program is our first that can read from an external device, and it will be limited to only the keyboard.

```
(* Chapter 8 - Program 2 *)
MODULE SimplIn;

FROM InOut IMPORT WriteString, WriteCard, WriteLn, Write,
                  ReadString, ReadCard, Read, EOL;

VAR Count, Number : CARDINAL;
    List          : ARRAY[1..6] OF CARDINAL;
    StringOfData  : ARRAY[1..80] OF CHAR;
    Alpha         : CHAR;

BEGIN
                          (* Example of reading in a word at a time *)
```

```
   WriteString("Input three words of information.");
   WriteLn;
   FOR Count := 1 TO 3 DO                        (* Read 3 words in *)
      ReadString(StringOfData);
      WriteString("---->");
      WriteString(StringOfData);
      WriteLn;
   END;
                                         (* Example of character reading *)
   WriteLn;
   WriteString("Input 50 characters.");
   WriteLn;
   FOR Count := 1 TO 50 DO
      Read(Alpha);
      Write(Alpha);
   END;
   WriteLn;
                              (* Example of reading in a line at a time. *)
   WriteLn;
   WriteString("Input three lines of information.");
   WriteLn;
   FOR Count := 1 TO 3 DO                      (* count three lines *)
      Number := 1;
      REPEAT               (* repeat until an end-of-line is found *)
         Read(Alpha);
         Write(Alpha);
         IF Alpha <> EOL THEN
            StringOfData[Number] := Alpha;
            Number := Number + 1;
         END;
      UNTIL Alpha = EOL;
      StringOfData[Number] := 0C;        (* End of string indicator *)
      WriteString("---->");
      WriteString(StringOfData);
      WriteLn;
   END;
                               (* Example of reading CARDINAL numbers in *)
   WriteLn;
   WriteString("Enter 6 CARDINAL numbers.");
   WriteLn;
   FOR Count := 1 TO 6 DO
      ReadCard(List[Count]);
      WriteLn;                (* New line for separation of numbers *)
   END;
   WriteLn;
   FOR Count := 1 TO 6 DO             (* Now, write the numbers out *)
      WriteCard(List[Count],8);
   END;
   WriteLn;

END SimplIn.
```

This program is broken up into four groups of statements, each illustrating some aspect of reading data from the keyboard. This could have been four separate files but it will be easier to compile and run one file.

Beginning with line 14 we have an example of the "ReadString" procedure which reads characters until it receives a space, a tab, a return, or some other nonprintable character. This loop will read three words on one line, one word on each of three lines, or any combination to get three words of groups of printable ASCII characters. After each word or group is read, it is simply "printed" to the monitor for your inspection.

ONE CHARACTER AT A TIME

The next group of statements is a loop in which 50 ASCII characters are read in and immediately echoed out to the monitor. It should be evident to you that the characters are read one at a time, and since the same variable is used for each character, they are not stored or saved in any way. In actual practice, the characters would be stored for whatever purpose you intend to use them for. When you run this part of the program, it will seem like the computer is simply acting like a word processor, echoing your input back to the monitor.

ONE LINE AT A TIME

The next section, beginning in line 32, reads in a full line before writing it out to the monitor. In this program we are introduced to the "EOL" which is a constant defined by the system for our use. It must be imported from "InOut" just like the procedures are, and it is a constant that is equal to that ASCII value that is returned when we hit the "return" key. It is therefore equal to the End-Of-Line character, and that is how it got its name. If we compare the input character to it, we can determine when we get to the End-Of-Line. That is exactly what this loop does. It continues to read characters until we find an EOL, then it terminates the input loop and displays the line of data. Notice that this time we do not simply read the data and ignore it but instead add it character by character to the ARRAY named "StringOfData". Of course, the next time through the loop we overwrite it. The careful student will also notice that, in line 45 we wrote a zero character in the character of the line just past the end of the line. The zero is to indicate the end-of-string for the string handling procedures. This portion of the program is easy, but will require a little time on your part to completely dissect it.

READING IN SOME NUMBERS, CARDINAL

Beginning in line 51, we have an example of reading 6 CARDINAL numbers in a loop. The procedure "ReadCard" will, when invoked by your program, read as many digits as you give it. When it reads any character other than a 0 through 9, it will terminate and return the number to your calling program. Notice that this time all 6 numbers are read in, stored, and when all are in, they are all displayed on one line. This should be easy for you to decipher.

COMPILING AND RUNNING THIS PROGRAM

There is no program that you have studied here that is as important for you to compile and run as this one is. You should spend considerable time running this program and comparing the results with the listing. Enter some invalid data when you are running the "ReadCard" portion of it to see what it does. When you are running the "line at a time" portion, try to enter more than 80 characters to see what it will do with it. This is a good point for you to learn what happens when errors occur. After you understand what this program does, we will proceed to file input and output.

FILE INPUT/OUTPUT

Load and display the file named FILEIO.MOD for your first file reading and writing. The library module named "InOut" has the ability to either read and write from/to the keyboard and monitor, or to read and write from/to files. The program before you redirects the input and output to files for an illustration of how to do it.

```
(* Chapter 8 - Program 3 *)
MODULE FileIO;

FROM InOut IMPORT WriteString, WriteInt, WriteLn, Write, Read,
                  OpenInput, OpenOutput, CloseInput, CloseOutput,
                  Done;

IMPORT Terminal;                 (* This is used for monitor output *)
                                 (* and InOut is used for the file  *)
                                 (* input and output.               *)

VAR Character : CHAR;

BEGIN
   REPEAT                                    (* open the input file *)
      Terminal.WriteString("Enter Input filename  - ");
      OpenInput("MOD");
   UNTIL Done;                      (* Quit when open is successful *)

   REPEAT                                   (* open the output file *)
```

```
      Terminal.WriteString("Enter Output filename - ");
      OpenOutput("DOG");
   UNTIL Done;                 (* quit when the open is successful *)

   REPEAT               (* character read/write loop - quit at EOF *)
      Read(Character);
      IF Done THEN                            (* Done = FALSE at EOF *)
         Write(Character);
      END;
   UNTIL NOT Done;
   CloseInput;
   CloseOutput;

END FileIO.
```

Line 16 requests the operator, namely you, to enter a filename to be used for input. There is nothing different about this statement than the others you have been using. The next line requests the system to open a file for inputting, and part of the procedure "OpenInput" is to go to the keyboard waiting for the filename to be typed in. So the message in line 16 is in preparation for what we know will happen in line 17. Whatever filename is typed in is opened for reading if it is found on the disk. The "MOD" in the parentheses is a default extension supplied, (this can be any extension you desire). If no extension is supplied by the operator, and if the filename does not have a "." following it, ".MOD" will be added to the filename. If the system can find the requested filename.extension, the "Done" flag is made TRUE and we can test it. In this example, if the flag is returned FALSE, we ask the operator to try again until he finally inputs a filename that exists on the default disk/directory.

NOW TO OPEN AN OUTPUT FILE

Once again, in line 21, we request a filename for output anticipating the operation of the "OpenOutput" in line 22. Line 22 waits for a keyboard input of a filename and if the filename entered has no extension, it adds the extension ".DOG" and attempts to open the file for writing. When you input the filename, adding a "." to the end of the filename will prevent the extension being added. If the Filename.extension does not exist, it will be created for you. If it does exist, it's contents will be erased.

It is nearly assured that the file will be created and the "Done" flag will be supplied as TRUE, but it would be good practice to check the flag anyway. It will be apparent when we get to the program on printer output, that it is impossible to open a file with certain names, one being "PRN", because the name is reserved for printer identification and the "Done" flag will be returned FALSE.

HOW DO I USE THE OPENED FILES?

Anytime you use this technique to open a file for writing, any procedure from InOut will now be redirected to that file. Anytime you use this technique to open a file for reading, any procedure from InOut will access the file named instead of the keyboard. In addition, the library module named "RealInOut" will also be redirected with "InOut". Any time you read or write, instead of using the keyboard and monitor, the input and output files will be used. The input and output will be the same except for where it goes to and comes from, and it is possible to only open one and leave the other intact. Thus input can be from a file, and output can still go to the monitor.

When I/O is redirected, the module "Terminal" is still available for use with the monitor and keyboard as I/O using this module can not be redirected. The module named "Terminal" does not have the flexibility of input and output that is found in "InOut" so it is a little more difficult to use.

There is a major drawback when using "InOut" with the I/O redirected. You are limited to one file for input and one file for output at one time. Finally, this method cannot be used to open a "fixed" or prenamed file, since it always surveys the keyboard for the filename. It will probably come as no surprise to you that all of these limitations will be overcome with another method given in the next two programs.

The program itself should be easy to follow, once you realize that the flag named "Done" returns TRUE when a valid character is found following a "Read", and FALSE when an End-Of-File (EOF) is detected. The "Done" flag is set up following each operation so its use is dictated by which procedure was called last. The program simply copies all characters from one file to another. When completed, the two procedures named "CloseInput" and "CloseOutput" are called to do just that, to close the files and once again make the I/O available to the keyboard and monitor. In this case, however, we immediately terminate the program without taking advantage of the return to normal.

Compile and run this program, being careful not to give it the name of an existing file for output, or it will overwrite the old data in the file and copy new data into it. That is the reason for the extension "DOG". Few people will have a file with that extension. For input, use the present filename (FILEIO.MOD), for output, use "STUFF", "STUFF.", and "STUFF.STU", observing the resulting new filename each time.

THE COMPLETE FILESYSTEM

Load and display the file named VARYFILE.MOD for an example using the complete "FileSystem" module. As stated earlier, Modula-2 does not have any input/output methods defined as part of the language. This is because the I/O available on computers is so diverse, there would be no way of defining a method that could be used on all computers. To eliminate the problem, Niklaus Wirth simply defined no I/O as part of the language, but he did suggest a few standard modules to perform the basic I/O tasks. Since they are only suggestions, compiler writers are not constrained to follow them, but in the interest of portability, most will. A very limited subset of all of the procedures are the only ones that will be used in the tutorial portion of this course. (A few other procedures will be used in the example programs given in those chapters.) It will be up to you to see that the procedures are in order with your compiler, and where they differ, to modify them. A few notes are available for your assistance in the COMPILER.DOC file on your distribution disk for those compilers available at the time of release of this tutorial.

```
(* Chapter 8 - Program 4 *)
MODULE VaryFile;

FROM FileSystem IMPORT Lookup, Close, File, Response, ReadChar;
FROM InOut      IMPORT Write, WriteString, ReadString, WriteLn;

VAR  NameOfFile : ARRAY[1..15] OF CHAR;
     InFile     : File;
     Character  : CHAR;

BEGIN
   REPEAT                  (* repeat until a good filename is found *)
      WriteLn;
      WriteString("Enter name of file to display ---> ");
      ReadString(NameOfFile);
      Lookup(InFile,NameOfFile,FALSE);
   UNTIL InFile.res = done;                  (* good filename found *)

   REPEAT       (* character read/display loop - quit at InFile.eof *)
      ReadChar(InFile,Character);
      IF NOT InFile.eof THEN
         Write(Character);
      END;
   UNTIL InFile.eof;                      (* quit when eof is found *)
   Close(InFile);

END VaryFile.
```

BACK TO THE PROGRAM NAMED VARYFILE

This time we IMPORT several procedures from the library module named "FileSystem" for I/O use. This time, we ask for the input filename and store it internally in a string variable. This implies that we can also define the filename as a constant that is carried in the program, making it possible to use a certain preprogrammed filename for input. We use the procedure "Lookup" to open the file. This procedure uses three arguments within the parentheses, the first being the symbolic filename which is a record of information about the file. (We will come to records later, don't worry too much about it at this point.) The second

argument is the name of the file on disk we wish to access, and the third argument is a boolean variable or constant. If it is TRUE, and the file name is not found, a new file of that name will be created. If it is FALSE, and the file is not found, a new file will not be created, and the record variable "InFile.res" will return the value "notdone". (That refers to one variable named "res" which is a part of the record "InFile".)

Note that the variable "InFile", is a record composed of many parts, but for the immediate future we only need to be concerned with its definition. It is defined as a variable of type "File" which is imported from the module named "FileSystem". Until you study the lesson in this tutorial on records, simply copy the method used here for file Input/Output.

Once the file is opened, you can use any of the procedures included in the "FileSystem" module, being careful to follow the rules given in your library documentation. The remainder of the program should be self- explanatory and will be left to your inspection. With this example in hand, spend some time studying your "FileSystem" module to become familiar with it, then compile the program and run it to observe its operation.

NOW FOR MULTIPLE FILE OPERATIONS

Load and display the file named PRINTFLE.MOD for an example program that uses 4 files at once, and still writes to the monitor. This program is very similar to the last in that it opens one file for reading, but it opens three files for writing. Each of the four files has its own identifier, a record of type "File", and each has its own filename. The three output files are firmly fixed to certain filenames, rather than ask the operator for names, and the third filename is a very special name, "PRN". This is not a file but is the access to the printer. Anything written to this file will go to your line printer, so you should turn your printer on in anticipation of running it. Your compiler may also allow a few other names such as "LPT0", "LPT1", etc, and there may be other names reserved for serial I/O such as to talk to a modem, a joystick, etc. You will need to consult your compiler documentation for a complete list of special names.

```
(* Chapter 8 - Program 5 *)
```

```
MODULE PrintFle;

FROM FileSystem IMPORT Lookup, Close, File, Response, ReadChar,
                       WriteChar;
FROM InOut      IMPORT Write, WriteString, ReadString, WriteLn;

VAR  NameOfFile : ARRAY[1..15] OF CHAR;
     InFile     : File;
     OutFile    : File;
     CapFile    : File;
     PrtFile    : File;
     Character  : CHAR;

BEGIN
   REPEAT                  (* repeat until a good filename is found *)
      WriteLn;
      WriteString("Enter name of file to display,");
      WriteString(" store, and print ---> ");
      ReadString(NameOfFile);
      Lookup(InFile,NameOfFile,FALSE);
   UNTIL InFile.res = done;                   (* good filename found *)

   Lookup(OutFile,"ANYNAME.TXT",TRUE);
   Lookup(CapFile,"CAPSONLY.TXT",TRUE);
   Lookup(PrtFile,"PRN",TRUE);

   WriteLn;
   REPEAT       (* character read/display loop - quit at InFile.eof *)
      ReadChar(InFile,Character);
      IF NOT InFile.eof THEN
         Write(Character);
         WriteChar(OutFile,Character);
         WriteChar(CapFile,CAP(Character));
         WriteChar(PrtFile,Character);
      END;
   UNTIL InFile.eof;                        (* quit when eof is found *)
   Close(InFile);
   Close(OutFile);
   Close(CapFile);
   Close(PrtFile);

END PrintFle.
```

The program itself is very simple and similar to the last one. A character is read from the input file, and output to the three output files and to the monitor. In the case of "CapFile", the character is capitalized before it is output simply to indicate to you that the files are indeed different. Study it until you understand it, then compile and run it. Look at the contents of the new files to see if they are correct.

MORE NEAT THINGS WE CAN DO WITH FILES

There are many more things that you can do with the "FileSystem" module. It is possible to open a file, begin reading until you come to a selected position, and change to a write file to overwrite some of the data with new data. You can write to a file, change it to a read file, reset it to the beginning, and read the data back out. You can rename a file, or delete it. It will be up to you to study the documentation for your "FileSystem" module, and learn how to use it effectively.

YOU ARE AT A SPECIAL POINT IN MODULA-2

With the completion of this chapter, you have arrived at a very special point in your study of Modula-2. Many people arrive at this point in a language and quit studying, preferring to use the language in a somewhat limited sense rather than to go on and learn the advanced topics. If your needs are few, you can quit here also and be well assured that you can write many programs with Modula-2. In fact there will be very few times when you cannot do all that you wish to do. However, if you choose to go on to the advanced topics, you will find that some of the programming chores will be greatly simplified.

Whether you decide to go on to the advanced topics or not, it would be wise for you to stop at this point and begin using what you have learned to actually write some programs for your own personal use. Everybody has need occasionally for a program to do some sort of translation of data in a text file for example. Write programs to do some data shuffling from file to file changing the format in some way. You should be able to think up several programs that you would find useful.

Spend some time studying and running the programs in the next chapter, then modify them to suit your needs, building up a few utilities for your software collection. The best way to learn to program is to program. You have all of the tools you need to get started, so you would do well to get started. Adding some programming experience will be a big help if you decide to continue your study into the advanced features of Modula-2.

Chapter 9 - Example programs

The programs included in this chapter are intended to be illustrations to you in how to write a complete program. The programs are meant to be useful to you either as an example of how to do some operation or as utility programs for your general use.

TIMEDATE - Get Time and Date

```
MODULE TimeDate;

(* Note - DOSCALL calls interrupt 21(hex) = 33(dec) which is the   *)
(*        general purpose BIOS interrupt.  It is used for many     *)
(*        operations on the disks, and other peripherals.  See     *)
(*        your DOS manual for details.  Some DOS manuals don't     *)
(*        have any information on this so your next best bet is    *)
(*        Peter Norton's book "Programmers Guide to the IBM-PC.    *)

FROM SYSTEM IMPORT DOSCALL;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR MonthDay, Year, Month, Day        : CARDINAL;
    HourMinute, SecondMillisec        : CARDINAL;
    Hour, Minute, Second, Millisecond : CARDINAL;
    Index, Count, Stuff               : CARDINAL;

BEGIN
   FOR Index := 1 TO 15 DO

      DOSCALL(2AH,Year,MonthDay);     (* Year is alone in a word    *)
      Month := MonthDay DIV 256;      (* Month is in top half       *)
      Day := MonthDay MOD 256;        (* Day is in bottom half      *)

      DOSCALL(2CH,HourMinute,SecondMillisec);
      Hour := HourMinute DIV 256;
      Minute := HourMinute MOD 256;
      Second := SecondMillisec DIV 256;
      Millisecond := 10*(SecondMillisec MOD 256); (* actually this *)
                                      (* is in hundredths of seconds *)

      WriteString("The time is ");
      WriteCard(Hour,2);
      WriteString(":");
      WriteCard(Minute,2);
      WriteString(":");
      WriteCard(Second,2);
      WriteString(":");
      WriteCard(Millisecond,3);
      WriteString("   ");
      WriteCard(Month,2);
      WriteString("/");
      WriteCard(Day,2);
      WriteString("/");
      WriteCard(Year,4);
      WriteLn;

               (* The following loop takes about one second to  *)
               (* finish its full count on an IBM-PC at normal  *)
               (* speed.  It may vary in time for your system.  *)

      FOR Count := 1 TO 13300 DO
         Stuff := 32000 DIV Count;
      END;
   END;
END TimeDate.
```

This program calls one of the DOS functions to get the current time and date. They are input as variables and can be printed or displayed in any format you desire. Your particular compiler may use a different format because there is no standard in Modula-2. This is one of those areas that will probably deviate from compiler to compiler. If you study your documentation that came with your compiler, you will find many other calls of this type described. This program is meant to be an illustration of how to use this particular call.

AREAS - Calculate Areas

```
(* This program is actually a very silly program with little or    *)
(* no utilitarian value.  It is valuable as an illustration of a   *)
(* method of implementing a menu for selection purposes.  Notice   *)
(* when you run it, that the response to your input is immediate   *)
(* No "return" is required.                                        *)

MODULE Areas;

FROM InOut IMPORT Read, WriteString, Write, WriteLn;
FROM RealInOut IMPORT WriteReal, ReadReal;

VAR InChar, CapInChar : CHAR;

(* ************************************************ AreaOfSquare *)
PROCEDURE AreaOfSquare;
VAR Length, Area : REAL;
BEGIN
   WriteString("Square    Enter length of a side ");
   ReadReal(Length);
   Area := Length * Length;
   WriteLn;
   WriteString("The area is ");
   WriteReal(Area,15);
   WriteLn;
```

```
END AreaOfSquare;

(* ********************************************* AreaOfRectangle *)
PROCEDURE AreaOfRectangle;
VAR Width, Height, Area : REAL;
BEGIN
   WriteString("Rectangle    Enter Width ");
   ReadReal(Width);
   WriteLn;
   WriteString("Enter Height ");
   ReadReal(Height);
   Area := Width * Height;
   WriteString("      The area is ");
   WriteReal(Area,15);
   WriteLn;
END AreaOfRectangle;

(* ********************************************** AreaOfTriangle *)
PROCEDURE AreaOfTriangle;
VAR Base, Height, Area : REAL;
BEGIN
   WriteString("Triangle    Enter base ");
   ReadReal(Base);
   WriteLn;
   WriteString("Enter height ");
   ReadReal(Height);
   Area := 0.5 * Base * Height;
   WriteString("      The area is ");
   WriteReal(Area,15);
   WriteLn;
END AreaOfTriangle;

(* ************************************************ AreaOfCIrcle *)
PROCEDURE AreaOfCircle;
VAR Radius, Area : REAL;
BEGIN
   WriteString("Circle      Enter Radius ");
   ReadReal(Radius);
   WriteLn;
   Area := 3.141592 * Radius * Radius;
   WriteString("The area is ");
   WriteReal(Area,15);
   WriteLn;
END AreaOfCircle;

(* ************************************************ Main Program *)
BEGIN
   REPEAT
      WriteLn;
      WriteString("You only need to input the first letter ");
      WriteString("of the selection.");
      WriteLn;
      WriteString("Select shape; Square Rectangle Triangle ");
      WriteString("Circle Quit");
      WriteLn;
      WriteString("Requested shape is ");
      Read(InChar);
      CapInChar := CAP(InChar);           (* Get capital of letter *)
      CASE CapInChar OF
        'S' : AreaOfSquare; |
        'R' : AreaOfRectangle; |
        'T' : AreaOfTriangle; |
        'C' : AreaOfCircle; |
        'Q' : WriteString("Quit    Return to DOS");
              WriteLn;
      ELSE
         Write(InChar);
         WriteString(" Invalid Character ");
         WriteLn;
      END;
   UNTIL CapInChar = 'Q';
END Areas.
```

This program is intended to be an illustration for you of how to build up a larger program than any other that we have examined up to this point. Notice that the main program is simply one CASE statement that calls all of the other procedures. It would be very simple to include the code from each procedure right in the CASE statement and have no procedure calls, but it would make the program very difficult to understand. The way this example is coded, the code is very easy to understand. After you understand the main program, it is a very simple matter to visit each procedure to see exactly what they do.

Notice how the menu works in this program. It reads one keystroke and responds immediately making it a very simple program to use.

PC - Printer Control

```
MODULE PC;

(* The codes in this program are for an EPSON RX-80.  You may have *)
(* to adjust the codes for your printer.                            *)

FROM InOut IMPORT WriteString, WriteLn, Read, Write;
FROM FileSystem IMPORT Lookup, WriteChar, Close, File;

VAR InputChar, Char0, Char1 : CHAR;
    PrintFile               : File;

BEGIN
   WriteString("F Formfeed");                   WriteLn;
   WriteString("C/N Compressed/Normal");        WriteLn;
   WriteString("D/S DoubleStrike/SingleStrike"); WriteLn;
   WriteString("E/R Emphasized/Regular");       WriteLn;
   WriteLn;
   WriteString("Enter Selection --> ");
   Read(InputChar);
   Write(InputChar);
   InputChar := CAP(InputChar);

                         (* Character input - now output the code *)

   CASE InputChar OF
    'F' : Char0 := 14C;      (* Formfeed *)
          Char1 := 0C;   |
    'C' : Char0 := 17C;      (* Compressed mode *)
          Char1 := 0C;   |
    'N' : Char0 := 22C;      (* Normal width *)
          Char1 := 0C;   |
    'D' : Char0 := 33C;      (* Double Strike - esc-'G' *)
          Char1 := 107C; |
    'S' : Char0 := 33C;      (* Single Strike - esc-'H' *)
          Char1 := 110C; |
    'E' : Char0 := 33C;      (* Emphasized Print - esc-'E' *)
          Char1 := 105C; |
    'R' : Char0 := 33C;      (* Emphasized Off - esc-'@' *)
          Char1 := 100C;
   ELSE
      WriteString("Invalid Character.");
   END;

   Lookup(PrintFile,"PRN",TRUE);    (* Open print file *)
   WriteChar(PrintFile,Char0);
   IF Char1 > 0C THEN
      WriteChar(PrintFile,Char1);
   END;
   Close(PrintFile);

END PC.
```

This is a very useful program that you can use to control your printer. It is specifically set up for an Epson RX-80, but you can modify the control characters to set up your printer to whatever mode you desire. To use the program, you call the program and supply a single letter according to the displayed menu, and the program will send the character or characters to the printer to select the enhanced, compressed, or whatever mode you desire. If your printer is located physically remote from you, you can use this program to send a formfeed to the printer by selecting the F option. If you have some longer control sequences to send, you may want to store the values in a string and use a loop to output the data until you come to an 0C character.

LIST - List Program File

```
MODULE List;    (* Program to list Modula-2 source files with page  *)
                (* numbers and line numbers.                        *)

FROM FileSystem IMPORT Lookup, Close, File, ReadChar, Response,
                       WriteChar;
FROM Conversions IMPORT ConvertCardinal;
FROM TimeDate IMPORT GetTime, Time;
IMPORT ASCII;
IMPORT InOut;

TYPE BigString   = ARRAY[1..80] OF CHAR;
     SmallString = ARRAY[1..25] OF CHAR;

VAR InFile     : File;          (* The Input File record           *)
    Printer    : File;          (* The Printer File record         *)
    NameOfFile : SmallString;   (* Storage for the filename        *)
    InputLine  : BigString;     (* The Input line of characters    *)
    LineNumber : CARDINAL;      (* The current line number         *)
    LinesOnPage : CARDINAL;     (* Number of Lines on this page    *)
    PageNumber  : CARDINAL;     (* Page Number                     *)
    Index       : CARDINAL;     (* Used locally in several proc's  *)
    Year,Day,Month     : CARDINAL;
    Hour,Minute,Second : CARDINAL;
    GoodFile    : BOOLEAN;

(* ********************************************** WriteCharString *)
(* Since there is no WriteString procedure in the FileSystem       *)
(* module, this procedure does what it would do.  It outputs a     *)
(* string until it comes to the end of it or until it comes to a   *)
(* character 0.                                                    *)
PROCEDURE WriteCharString(CharString : ARRAY OF CHAR);
BEGIN
   Index := 0;
   LOOP
      IF Index > HIGH(CharString) THEN EXIT END; (* Max = 80 chars  *)
      IF CharString[Index] = 0C THEN EXIT END;   (* If a 0C is found*)
      WriteChar(Printer,CharString[Index]);
      INC(Index);
   END;
END WriteCharString;


(* *********************************************** WriteLnPrinter *)
(* Since there is no WriteLn procedure in the FileSystem module,   *)
(* procedure does its job.                                         *)
PROCEDURE WriteLnPrinter;
CONST CRLF = 12C;
BEGIN
   WriteChar(Printer,CRLF);
END WriteLnPrinter;


(* *********************************************** GetFileAndOpen *)
(* This procedure requests the filename, receives it, and opens the *)
(* source file for reading and printing.  It loops until a valid   *)
(* filename is found.                                              *)
```

```
PROCEDURE GetFileAndOpen(VAR GoodFile : BOOLEAN);
BEGIN
   InOut.WriteLn;
   InOut.WriteString("Name of file to print ---> ");
   InOut.ReadString(NameOfFile);
   Lookup(InFile,NameOfFile,FALSE);
   IF InFile.res = done THEN
      GoodFile := TRUE;
      Lookup(Printer,"PRN",TRUE);            (* open printer as a file
*)
   ELSE
      GoodFile := FALSE;
      InOut.WriteString("   File doesn't exist");
      InOut.WriteLn;
   END;
END GetFileAndOpen;


(* ************************************************** Initialize *)
(* This procedure initializes some of the counters.              *)
PROCEDURE Initialize;
VAR PackedTime : Time;
BEGIN
   LineNumber := 1;
   LinesOnPage := 0;
   PageNumber := 1;
   GetTime(PackedTime);
   Day := PackedTime.day MOD 32;
   Month := PackedTime.day DIV 32;
   Month := Month MOD 16;
   Year := 1900 + PackedTime.day DIV 512;
   Hour := PackedTime.minute DIV 60;
   Minute := PackedTime.minute MOD 60;
   Second := PackedTime.millisec DIV 1000;
END Initialize;


(* ******************************************** PrintTimeAndDate *)
(* This procedure prints the time and date at the top of every page *)
PROCEDURE PrintTimeAndDate;
VAR OutChars : ARRAY[0..4] OF CHAR;
BEGIN
   WriteCharString("      ");
   ConvertCardinal(Hour,2,OutChars);
   WriteCharString(OutChars);
   WriteCharString(":");
   ConvertCardinal(Minute,2,OutChars);
   WriteCharString(OutChars);
   WriteCharString(":");
   ConvertCardinal(Second,2,OutChars);
   WriteCharString(OutChars);
   WriteCharString("  ");
   ConvertCardinal(Month,2,OutChars);
   WriteCharString(OutChars);
   WriteCharString("/");
   ConvertCardinal(Day,2,OutChars);
   WriteCharString(OutChars);
   WriteCharString("/");
   ConvertCardinal(Year,4,OutChars);
   WriteCharString(OutChars);
END PrintTimeAndDate;


(* ************************************************** OutputHeader *)
(* This procedure prints the filename at the top of each page along *)
(* with the page number.                                            *)
PROCEDURE OutputHeader;
VAR PageOut : ARRAY[1..4] OF CHAR;
BEGIN
   WriteCharString("   Filename --> ");
   WriteCharString(NameOfFile);
   WriteCharString("           ");
   PrintTimeAndDate;
   WriteCharString("   Page");
   ConvertCardinal(PageNumber,4,PageOut);
   WriteCharString(PageOut);
   WriteLnPrinter;
   WriteLnPrinter;
   INC(PageNumber);
END OutputHeader;


(* ************************************************** OutputFooter *)
(* This procedure outputs 8 blank lines at the bottom of each page. *)
PROCEDURE OutputFooter;
BEGIN
   FOR Index := 1 TO 8 DO
      WriteLnPrinter;
   END;
END OutputFooter;


(* ****************************************************** GetALine *)
(* This procedure inputs a line from the source file.  It quits when*)
(* it finds an end-of-line, an end-of-file, or after it gets 80     *)
(* characters.                                                      *)
PROCEDURE GetALine;
VAR LocalChar : CHAR;
BEGIN
   FOR Index := 1 TO 80 DO      (* clear the input area so that the *)
      InputLine[Index] := 0C;   (* search for 0C will work.         *)
   END;

   Index := 1;
   LOOP
      ReadChar(InFile,LocalChar);
      IF InFile.eof THEN EXIT END;
      InputLine[Index] := LocalChar;
      IF LocalChar = ASCII.EOL THEN EXIT END;
      INC(Index);
      IF Index = 81 THEN EXIT END;
   END;
```

```
END GetALine;


(* ************************************************** OutputLine *)
(* Output a line of test with the line number in front of it, after *)
(* checking to see if the page is full.                              *)
PROCEDURE OutputLine;
VAR Count       : CARDINAL;
    CardOutArea : ARRAY[1..8] OF CHAR;
BEGIN
   INC(LinesOnPage);
   IF LinesOnPage > 56 THEN
      OutputFooter;
      OutputHeader;
      LinesOnPage := 1;
   END;
   ConvertCardinal(LineNumber,6,CardOutArea);
   INC(LineNumber);
   WriteCharString(CardOutArea);
   WriteCharString("  ");
   WriteCharString(InputLine);
END OutputLine;


(* ************************************************* SpacePaperUp *)
(* At the end of the listing, space the paper up so that a new page *)
(* is ready for the next listing.                                    *)
PROCEDURE SpacePaperUp;
VAR Count : CARDINAL;
BEGIN
   Count := 64 - LinesOnPage;
   FOR Index := 1 TO Count DO
      WriteLnPrinter;
   END;
   Close(InFile);
   Close(Printer);
END SpacePaperUp;


(* ************************************************* Main Program *)
(* This is nothing more than a big loop.  It needs no comment.      *)
BEGIN
   GetFileAndOpen(GoodFile);
   IF GoodFile THEN
      Initialize;
      OutputHeader;
      REPEAT
         GetALine;
         IF NOT InFile.eof THEN
            OutputLine;
         END;
      UNTIL InFile.eof;
      SpacePaperUp;
   END;
END List.
```

If you ran the batch file named LISTALL as suggested at the beginning of this tutorial to print out all of the source files, you have already used this program. It is the program that will list any ASCII file, adding line numbers, page numbers, and the date and time, on the printer. It is specifically designed to be a program listing utility. The operation is very simple, and you should have no trouble in understanding this program or what it does.

Additional programs will be given at the end of Part III for your information. You will no doubt find additional example programs in various books and periodicals and it would be to your advantage to to spend some time studying them as illustrations of both good and bad practices.

Chapter 10 - Scalars, subranges, and Sets

PREREQUISITES FOR THIS MATERIAL

In order to understand the material in this chapter, you should have a fairly good understanding of the material in Part I of this tutorial.

A scalar, also called an enumerated type, is a list of values which a variable of that type may assume. Look at the file named ENTYPES.MOD for an example of some scalars. The first TYPE declaration defines "Days" as being a type which can take on any one of seven values. Since, within the VAR declaration, "Day" is assigned the type of "Days", then "Day" is a variable which can assume any one of seven different values. Moreover, "Day" can be assigned the value "mon", or "tue", etc., which makes the program easier to follow and understand. Internally, the Modula-2 system does not actually assign the value "mon" to the variable "Day", but it uses an integer representation for each of the names. This is important to understand because you must realize that you cannot print out "mon", "tue", etc., but can only use them for indexing control statements.

```
(* Chapter 10 - Program 1 *)
MODULE Entypes;

FROM InOut IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteReal;

TYPE Days = (mon,tue,wed,thu,fri,sat,sun);
     TimeOfDay = (morning,afternoon,evening,night);

VAR Day             : Days;
    Time            : TimeOfDay;
    RegularRate     : REAL;
    EveningPremium  : REAL;
    NightPremium    : REAL;
    WeekendPremium  : REAL;
    TotalPay        : REAL;

BEGIN    (* Main program *)
   WriteString('                         Pay rate table');
   WriteLn;
   WriteLn;
   WriteString('  DAY       Morning    Afternoon');
   WriteString('    Evening      Night');
   WriteLn;

   RegularRate := 12.00;       (* This is the normal pay rate *)
   EveningPremium := 1.10;     (* 10 percent extra for working late *)
   NightPremium := 1.33;      (* 33 percent extra for graveyard *)
   WeekendPremium := 1.25;    (* 25 percent extra for weekends *)

   FOR Day := mon TO sun DO

      CASE Day OF
        mon : WriteString('Monday   '); |
        tue : WriteString('Tuesday  '); |
        wed : WriteString('Wednesday'); |
        thu : WriteString('Thursday '); |
        fri : WriteString('Friday   '); |
        sat : WriteString('Saturday '); |
        sun : WriteString('Sunday   ');
      END;     (* of CASE statment *)

      FOR Time := morning TO night DO
         CASE Time OF
           morning   : TotalPay := RegularRate; |
           afternoon : TotalPay := RegularRate; |
           evening   : TotalPay := RegularRate * EveningPremium; |
           night     : TotalPay := RegularRate * NightPremium;
         END;  (* of CASE statement *)

         CASE Day OF
           sat : TotalPay := TotalPay * WeekendPremium; |
           sun : TotalPay := TotalPay * WeekendPremium;
         ELSE (* Do nothing *)
         END;  (* of CASE statement *)

         WriteReal(TotalPay,12);
      END;  (* of Time loop *)
      WriteLn;
   END;  (* of FOR loop *)

END Entypes.
```

Note that there is an upper limit of 16 enumerated types placed on you by most implementations of Modula-2. This is actually a very low limit and is most unfortunate that this limit exists.

The second line of the type definition defines "TimeOfDay" as another "type". The variable "Time" can only be assigned one of four values since it is defined as the type "TimeOfDay". It should be clear that even though it can be "morning", it cannot be assigned "morningtime" or any other variant spelling of morning, since it is simply another identifier which must have an exact spelling to be understood by the compiler.

Several REAL variables are defined to allow us to demonstrate the use of the scalar variables. After writing a header for our output, the REAL variables are initialized to some values that are probably not real life values, but will serve to illustrate use of the scalar variable.

A BIG SCALAR VARIABLE LOOP

The remainder of the program is one large loop being controlled by the variable "Day" as it goes through all of its values, one at a time. Note that the loop could have gone from "tue" to "sat" or whatever portion of the range desired. It does not have to go through all of the values of "Day". Using "Day" as the CASE variable, the name of one of the days of the week is written out each time we go through the loop. Another loop controlled by "Time" is executed four times, once for each value of "Time". The two CASE statements within the inner loop are used to calculate the total pay rate for each time period and each day. The data is formatted carefully to make a nice looking table of pay rates as a function of "Time" and "Day".

Take careful notice of the fact that the scalar variables never entered into the calculations, and they were not printed out. They were only used to control the flow of the logic. It was much neater than trying to remember that "mon" is represented by a 0, "tue" is represented by a 1, etc. In fact, those numbers are used for the internal representation of the scalars, but we can relax and let the Modula-2 system worry about the internal representation of our scalars.

Compile and run this program and observe the form of the output data. The only format available with some compilers are the "E" notation which does not make for a very well formatted or easily read output. Don't let this worry you, when we get to Part III of this tutorial we will see how we can write our own output routines to display, or print, floating point numbers in any format we can think up.

One other thing should be pointed out in this module. If you observe the CASE statements you will notice that the one that starts in line 33 does not have an ELSE clause. It is really not needed because every possible value that the variable "Day" can have is covered in the various branches. In the CASE statement starting in line 51, there is an ELSE clause because only two of the possible 7 values are acted on in the CASE body itself. Without the ELSE, the program would not know what to do with a value of "mon" through "fri", so the ELSE is required here, but not in the earlier one.

LETS LOOK AT SOME SUBRANGES

Examine the program SUBRANGE.MOD for an example of subranges. It may be expedient to define some variables that only cover a part of the full range as defined in a scalar type. Notice that "Days" is declared a scalar type as in the last program, and "Work" is declared a type with an even more restricted range. In the VAR declaration, "Day" is once again defined as the days of the week and can be assigned any of the days by the program. The variable "Workday", however, is assigned the type "Work", and can only be assigned the days "mon" through "fri". If an attempt is made to assign "Workday" the value "sat", a runtime error will be generated. A carefully written program will never attempt that, and it would be an indication that something is wrong with either the program or the data. This is one of the advantages of Modula-2 over older languages.

```
(* Chapter 10 - Program 2 *)
MODULE Subrange;

TYPE Days = (mon,tue,wed,thu,fri,sat,sun);
     Work = [mon..fri];
     Rest = [sat..sun];

VAR  Day      : Days;  (* This is any day of the week *)
     Workday  : Work;  (* These are the working days  *)
     Weekend  : Rest;  (* The two weekend days only   *)
     Index    : [1..12];
     Alphabet : ['a'..'z'];
     Start    : ['a'..'e'];

BEGIN    (* Main program *)
(*  The following statements are commented out because they
    contain various errors that will halt compilation.

   Workday := sat;   sat is not part of Workday's subrange.
   Rest := fri;      fri is not part of Rest's subrange.
   Index := 13;      Index is only allowed to go up to 12,
   Index := -1;        and down to 1.
   Alphabet := 'A';  Alphabet, as defined, includes only the
                       lowercase alphabet.
   Start := 'h';     h is not in the first five letters.

   End of the commented out section of program.
*)

   Workday := tue;
   Weekend := sat;
   Day := Workday;
   Day := Weekend;
   Index := 3 + 2 * 2;
   Start := 'd';
   Alphabet := Start;
                            (* Since Alphabet is 'd'     *)
   INC(Alphabet);           (*   and now 'e'             *)
   Start := Alphabet;       (* Start will be 'e'         *)
   DEC(Start,2);            (* Start will be 'c'         *)
   Day := wed;
   INC(Day);                (* Day will now be 'thu'     *)
   INC(Day);                  (* Day will now be 'fri'     *)
   Index := ORD(Day);         (* Index will be 4 (fri = 4) *)

END Subrange.
```

Further examination will reveal that "Index" is declared as being capable of storing only the range of INTEGERS from 1 to 12. During execution of the program, if an attempt is made to assign "Index" any value outside of that range, a runtime error will be generated. Suppose the variable "Index" was intended to refer to your employees, and you have only 12. If an attempt was made to refer to employee number 27, or employee number -8, there is clearly an error somewhere in the data and you would want to stop running the payroll to fix the problem. Modula-2 would have saved you a lot of grief.

SOME STATEMENTS WITH ERRORS IN THEM

In order to have a program that would compile without errors, and yet show some errors, the first part of the program is not really a part of the program since it is within a comment area. This is a trick to remember when you are debugging a program, a troublesome part can be commented out until you are ready to include it with the rest. The errors are self explanatory.

Going beyond the area commented out, there are seven assignment statements as examples of subrange variable use. Notice that the variable "Day" can always be assigned the value of either "Workday" or "Weekend", but the reverse is not true because "Day" can assume values that would be illegal for the other variables.

THREE VERY USEFUL FUNCTIONS

The last section of the example program demonstrates the use of three very important functions when using scalars. The first is the "INC" function which returns the value of the scalar following that scalar used as the argument. If the argument is the last value in the list, a runtime error is generated. The next function is the "DEC" that returns the value of the prior scalar to that used in the argument. All scalars have an internal representation starting at 0 and increasing by one until the end is reached. The third function is the "ORD" which simply returns the numerical value of the scalar variable.

In our example program, ORD(Day) is 5 if "Day" has been assigned "sat", but ORD(Weekend) is 0 if "Weekend" has been assigned "sat". As you gain experience in programming with scalars and subranges, you will realize the value of these three functions.

A few more thoughts about subranges are in order before we go on to another topic. A subrange is always defined by two predefined constants, and is always defined in an ascending order. A variable defined as a subrange type is actually a variable defined with a restricted range, and should be used as often as possible in order to prevent garbage data. There are actually very few variables ever used that cannot be restricted by some amount. The limits may give a hint at what the program is doing and can help in understanding the program operation. Subrange types can only be constructed using the simple data types.

SETS

Now for a new topic, sets. Examining the example program SETS.MOD will reveal use of some sets. A scalar type is defined first, in this case the scalar type named "Goodies". A set is then defined with the reserved words SET OF followed by a predefined scalar type. Most microcomputers have an upper limit of 16 elements that can be used in a set.

```
(* Chapter 10 - Program 3 *)
MODULE Sets;

FROM InOut IMPORT WriteString, WriteLn;

TYPE Goodies = (IceCream,WhippedCream,Banana,Nuts,Cherry,
                ChocSyrup,StrawBerries,Caramel,SodaWater,
                Salt,Pepper,Cone,Straw,Spoon,Stick);

     Treat = SET OF Goodies;

VAR  Sundae       : Treat;
     BananaSplit  : Treat;
     Soda         : Treat;
     IceCreamCone : Treat;
     NuttyBuddy   : Treat;
     Mixed        : Treat;
     Index        : Treat;

BEGIN
        (* Define all ingredients used in each treat *)
IceCreamCone := Treat{IceCream,Cone};
Soda := Treat{Straw,SodaWater,IceCream,Cherry};
BananaSplit := Treat{IceCream..Caramel};
BananaSplit := BananaSplit + Treat{Spoon};
NuttyBuddy := Treat{Cone,IceCream,ChocSyrup,Nuts};
Sundae := Treat{IceCream,WhippedCream,Nuts,Cherry,ChocSyrup,
           Spoon};
```

```
                (* Combine for a list of all ingredients used *)
Mixed := IceCreamCone + Soda + BananaSplit + NuttyBuddy +
         Sundae;

                (* Now find what ingredients are not used *)

Mixed := Treat{IceCream..Stick} - Mixed;

   IF IceCream     IN Mixed THEN
                  WriteString('Ice cream not used');
                  WriteLn; END;
   IF WhippedCream IN Mixed THEN
                  WriteString('Whipped Cream not used');
                  WriteLn; END;
   IF Banana       IN Mixed THEN
                  WriteString('Bananas not used');
                  WriteLn; END;
   IF Nuts         IN Mixed THEN
                  WriteString('Nuts not used');
                  WriteLn; END;
   IF Cherry       IN Mixed THEN
                  WriteString('Cherries not used');
                  WriteLn; END;
   IF ChocSyrup    IN Mixed THEN
                  WriteString('Chocolate Syrup not used');
                  WriteLn; END;
   IF StrawBerries IN Mixed THEN
                  WriteString('Strawberries not used');
                  WriteLn; END;
   IF Caramel      IN Mixed THEN
                  WriteString('Caramel not used');
                  WriteLn; END;
   IF SodaWater    IN Mixed THEN
                  WriteString('SodaWater not used');
                  WriteLn; END;
   IF Salt         IN Mixed THEN
                  WriteString('Salt not used');
                  WriteLn; END;
   IF Pepper       IN Mixed THEN
                  WriteString('Pepper not used');
                  WriteLn; END;
   IF Cone         IN Mixed THEN
                  WriteString('Cone not used');
                  WriteLn; END;
   IF Straw        IN Mixed THEN
                  WriteString('Straw not used');
                  WriteLn; END;
   IF Spoon        IN Mixed THEN
                  WriteString('Spoon not used');
                  WriteLn; END;
   IF Stick        IN Mixed THEN
                  WriteString('Stick not used');
                  WriteLn; END;
END Sets.
```

Several variables are defined as sets of "Treat", after which they can individually be assigned portions of the entire set. Consider the variable "IceCreamCone" which has been defined as a set of type "Treat". This variable is composed of as many elements of "Goodies" as we care to assign to it. In the program, we define it as being composed of "IceCream", and "Cone". The set "IceCreamCone" is therefore composed of two elements, and it has no numerical or alphabetic value as most other variables have. Continuing in the program, you will see 4 more delicious deserts defined as sets of their components. Notice that the banana split is first defined as a range of terms, then another term is added to the group. All five are combined in the set named "Mixed", then "Mixed" is subtracted from the entire set of values to form the set of ingredients that are not used in any of the deserts. Each ingredient is then checked to see if it is IN the set of unused ingredients, and printed out if it is. Running the program will reveal a list of the unused elements.

In this example, better programming practice would have dictated defining a new variable, possibly called "Remaining" for the ingredients that were unused. It was desirable to illustrate that "Mixed" could be assigned a value based on subtracting itself from the entire set, so the poor variable name was used.

This example resulted in some nonsense results but hopefully it led your thinking toward the fact that sets can be used for inventory control, possibly a parts allocation scheme, or some other useful system.

Chapter 11 - Records

PREREQUISITES FOR THIS MATERIAL

In order to do a profitable study of this material, you will need a good understanding of all of the material in Part I. The material concerning the scalar type from chapter 11 is also needed.

We come to the grandaddy of all data structures in Modula-2, the RECORD. A record is composed of a number of variables any of which can be of any predefined data type, including other records. Rather than spend time trying to define a record in detail, lets go right to the first example program, SMALLREC.MOD. This is a program using nonsense data that will illustrate the use of a record.

```
(* Chapter 11 - Program 1 *)
MODULE SmallRec;

FROM InOut   IMPORT WriteString, WriteCard, WriteLn;

TYPE Description = RECORD
       Year   : CARDINAL;
       Model  : ARRAY[0..20] OF CHAR;
       Engine : ARRAY[0..8] OF CHAR
       END;

VAR  Cars  : ARRAY[1..10] OF Description;
     Index : CARDINAL;

BEGIN   (* Main Program *)
   FOR Index := 1 TO 10 DO
      Cars[Index].Year := 1930 + Index;
      Cars[Index].Model := " Duesenberg";
      Cars[Index].Engine := "V8";
   END;

   Cars[2].Model := " Stanley Steamer";
   Cars[2].Engine := "Coal";
   Cars[7].Engine := "V12";
   Cars[9].Model := " Ford";
   Cars[9].Engine := "rusted";
   Cars[9].Year := 1981;

   FOR Index := 1 TO 10 DO
      WriteString('My');
      WriteCard(Cars[Index].Year,5);
      WriteString(Cars[Index].Model);
      WriteString(" has a ");
      WriteString(Cars[Index].Engine);
      WriteString(' engine.');
      WriteLn;
   END;
END SmallRec.
```

A VERY SIMPLE RECORD

There is only one entry in the TYPE declaration part of the program, namely the record identified by "Description". The record is composed of three fields, the "Year", "Model", and "Engine" variables. Notice that the three fields are each of a different type, indicating that the record can be of mixed types. You have a complete example of the way a record is defined before you. It is composed of the identifier "Description", the reserved word RECORD, the list of elements, and followed by END;. Notice that this only defines a TYPE, it does not define any variables. That is done in the VAR declaration where the variable "Cars" is defined to have 10 complete records of the type "Description". The variable "Cars[1]" has three components, "Year", "Model", and "Engine", and any or all of these components can be used to store data pertaining to "Cars[1]".

In order to assign values to the various fields, the variable name is followed by the sub-field with a separating period. Keep in mind that "Cars[1]" is a complete record containing three variables, and to assign or use one of the variables, you must designate which sub-field you are interested in. See the program where the three fields are assigned meaningless data for illustration. The "Year" field is assigned an integer number varying with the subscript, all "Model" fields are assigned the name "Duesenburg", and all "Engine" variables are assigned the value "V8". In order to further illustrate that there are actually 30 variables in use here, a few are changed at random in the next few statements, being very careful to maintain the required types as defined in the TYPE declaration part of the program. Finally, all ten composite variables, consisting of 30 actual variables in a logical grouping are printed out using the same "var.subfield" notation described above.

If the preceding description of a record is not clear in your mind, review it very carefully. It's a very important concept in Modula-2, and you won't have a hope of a chance of understanding the next example until this one is clear.

A SUPER RECORD

Examine the example file BIGREC.MOD for a very interesting record. First we have a constant defined. Ignore it for the moment, we will come back to it later. Within the TYPE declaration we have three records defined, and upon close examination, you will notice that the first two records are included as part of the definition of the third record. The record identified as "Person", actually contains 8 variable definitions, three within the "FullName" record, two of its own, and three within the "Date" record. This is a TYPE declaration and does not actually define any variables, that is done in the VAR part of the program.

```
(* Chapter 11 - Program 2 *)
MODULE BigRec;

FROM InOut   IMPORT WriteString, Write, WriteLn;

CONST  NumberOfFriends = 50;

TYPE   FullName = RECORD
         FirstName : ARRAY[0..12] OF CHAR;
         Initial   : CHAR;
         LastName  : ARRAY[0..15] OF CHAR;
       END;

       Date = RECORD
         Day, Month, Year : CARDINAL;
       END;

       Person = RECORD
         Name     : FullName;
         City     : ARRAY[0..15] OF CHAR;
         State    : ARRAY[0..3] OF CHAR;
         BirthDay : Date;
       END;

VAR   Friend              : ARRAY[1..NumberOfFriends] OF Person;
      Self,Mother,Father : Person;
      Index              : CARDINAL;

BEGIN  (* Main Program *)
   Self.Name.FirstName := "Charley";
   Self.Name.Initial := 'Z';
   Self.Name.LastName := "Brown";

   WITH Self DO
      City := "Anywhere";
      State := "CA";
      BirthDay.Day := 17;
      WITH BirthDay DO
         Month := 7;
         Year := 1938;
      END;
   END;   (* All data for Self now defined *)

   Mother := Self;
   Father := Mother;

   FOR Index := 1 TO NumberOfFriends DO
      Friend[Index] := Mother;
   END;

   WriteString(Friend[27].Name.FirstName);
   WriteString(' ');
   Write(Friend[33].Name.Initial);
   WriteString(' ');
   WriteString(Father.Name.LastName);
   WriteLn;
END BigRec.
```

The VAR part of the program defines some variables beginning with the array of "Friend" containing 50 (because of the constant definition in the CONST part) records of "Person". Since "Person" defines 8 fields, we have now defined 8 times 50 = 400 separate and distinct variables. Each of the 400 separate variables has its own type associated with it, and the compiler will generate an error if you try to assign any of those variables the wrong type of data. Since "Person" is a TYPE definition, it can be used to define more than one variable, and in fact it is used again to define three more records, "Self", "Mother", and "Father". These three records are each composed of 8 variables, so we have 24 more variables which we can manipulate within the program. Finally we have the variable "Index" defined as a simple CARDINAL type variable. Notice that if we desired, we could also define a variable of type "FullName" composed of 3 simple variables.

HOW TO MANIPULATE ALL OF THAT DATA

In the program we begin by assigning data to all of the fields of "Self". Examining the first three statements of the main program, we see the construction we learned in the last example program being used, namely the period between descriptor fields. The main record is named "Self", and we are interested in the first part of it namely the "Name" part of the person record. Since the "Name" part of the person record is itself composed of three parts, we must designate which part of it we are interested in. The complete description "Self.Name.FirstName" is the complete description of the first name of "Self" and is the first assignment statement which is assigned the name of "Charley". The next two fields are handled in the same way and are self explanatory.

WHAT IS THE WITH STATEMENT?

Continuing on to the fourth field, the "City", there are only two levels required because

"City" is not another record definition. The fourth field is therefore completely defined by "Self.City". Notice the "WITH Self DO" statement. This is a shorthand notation used with record definitions to simplify coding. From the BEGIN at the next statement to the matching END; about 10 statements later, any variables within the "Self" record are used as though they had a "Self." in front of them. It greatly simplifies coding to be able to omit the leading identifier within the WITH section of code. You will see that "City", and "State", are easily assigned values without further reference to the "Self" variable. When we get to the "Day" part of the birthday, we are back to three levels and the complete definition is "Self.Birthday.Day" but once again, the "Self." part is taken care of automatically because we are still within the "WITH Self DO" area.

To illustrate the WITH statement further, another is introduced, "WITH Birthday DO", and an area is defined by the BEGIN END pair. Within this area both leading identifiers are handled automatically to simplify coding, and "Month" is equivalent to writing "Self.Birthday.Month" if both WITH statements were removed. You may be wondering how many levels of nesting are allowed in record definitions. There doesn't appear to be a limit according to the Modula-2 definition, but we do get a hint at how far it is possible to go. In most implementations of Modula-2, you are allowed to have WITH statements nested to nine levels, and it would be worthless to nest WITH statements deeper than the level of records. Any program requiring more levels than nine is probably far beyond the scope of your programming ability, and mine, for a long time.

After assigning a value to the year, the entire record of "Self" is defined, all eight variables.

SUPER-ASSIGNMENT STATEMENTS

The next statement, "Mother := Self;" is very interesting. Since both of these are records, both are the same type of record, and both therefore contain 8 variables, Modula-2 is smart enough to recognize that, and assign all eight values contained in "Self" to the corresponding variables of "Mother". So after one statement, "Mother" is completely defined. The next statement assigns the same values to the eight respective fields of "Father", and the next two lines assign all 50 "Friend" variables the same data. We have therefore generated 400 + 24 = 424 separate pieces of data so far in this program. We could print it all out, but since it is nonsense data, it would only waste time and paper. The next three lines write out three sample pieces of the data for your inspection.

WHAT GOOD IS ALL OF THIS

It should be obvious to you that what this program does, even though the data is nonsense, appears to be the beginning of a database management program, which indeed it is. It is a crude beginning, and has a long way to go to be useful, but you should see a seed for a useful program.

Now to go back to the CONST as promised. The number of friends was defined as 50 and used for the size of the array and in the assignment loop near the end of the program. You can now edit this number and see how big this database can become on your computer. Your compiler should be capable of storing about 1000 records even within the smallest model available on any compiler. If your compiler uses a larger memory model, you will be able to store significantly more records. See how big you can make the number of friends before you get the memory overflow message. Keep the number in mind because when we get to the chapter on Pointers and Dynamic Allocation, you should see a marked increase in allowable size, especially if you have a large amount of RAM installed in your computer. If your compiler uses a large memory model, you won't see an increase in size but it will be an interesting exercise anyway.

A VARIANT RECORD

If any part of this chapter is still unclear, it would be good for you to go back and review it at this time. The next example will really tax your mind to completely understand it, especially if the prior material is not clear.

Examine the program VARREC.MOD for an example of a program with a variant record definition. In this example, we first define a scalar type, namely "KindOfVehicle" for use within the record. Then we have a record defining "Vehicle", intended to define several different types of vehicles, each with different kinds of data. It would be possible to define all variables for all types of vehicles, but it would be a waste of storage space to define the number of tires for a boat, or the number of propeller blades used on a car or truck. The variant record lets us define the data precisely for each vehicle without wasting data storage space.

```
(* Chapter 11 - Program 3 *)
MODULE VarRec;

FROM InOut   IMPORT WriteString, WriteInt, WriteLn;

TYPE   KindOfVehicle = (Car,Truck,Bicycle,Boat);

       Vehicle = RECORD
         OwnerName   : ARRAY[0..25] OF CHAR;
         GrossWeight : CARDINAL;
         Value       : REAL;
         CASE WhatKind : KindOfVehicle OF
           Car     : Wheels  : CARDINAL;
                     Engine  : ARRAY[0..12] OF CHAR |
           Truck   : Motor   : ARRAY[0..8] OF CHAR;
                     Tires   : CARDINAL;
                     PayLoad : CARDINAL            |
           Bicycle : Tyres   : INTEGER             |
           Boat    : PropBlades : INTEGER;
                     Sail    : BOOLEAN;
                     Power   : ARRAY[0..8] OF CHAR;
         END; (* of CASE *)
       END;   (* of record *)

VAR   Sunfish, Ford, Schwinn, Mac : Vehicle;

BEGIN
   Ford.OwnerName := "Walter";         (* Fields defined in order *)
   Ford.GrossWeight := 5750;
   Ford.Value := 2595.00;
   Ford.WhatKind := Truck;
   Ford.Motor := "V8";
   Ford.Tires := 18;
   Ford.PayLoad := 12000;

   WITH Sunfish DO
      WhatKind := Boat;       (* Fields defined in random order *)
      Sail := TRUE;
      PropBlades := 3;
      Power := "Wind";
      GrossWeight := 375;
      Value := 1300.00;
      OwnerName := "Herman and George";
   END;

   Ford.Engine := "Flathead";         (* Tag-field not defined yet *)
   Ford.WhatKind := Car;              (* but it must be before it  *)
                                      (* can be referred to        *)
   Ford.Wheels := 4;        (* Notice that the non-variant part is *)
                            (* not redefined here.                 *)

   Mac := Sunfish;    (* Entire record copied, including tag-field *)

   IF Ford.WhatKind = Car THEN                (* This should print *)
      WriteString(Ford.OwnerName);
      WriteString(" owns the car with the ");
      WriteString(Ford.Engine);
      WriteString(' engine.');
      WriteLn;
   END;

   IF Sunfish.WhatKind = Bicycle THEN     (* This should not print *)
      WriteString("The sunfish is a bicycle");
      WriteLn;
   END;

   IF Mac.WhatKind = Boat THEN                (* This should print *)
```

```
      WriteString("The Mac is now a boat with");
      WriteInt(Mac.PropBlades,2);
      WriteString(" propeller blades.");
      WriteLn;
   END;
END VarRec.
```

WHAT IS A TAG-FIELD?

In the record definition we have the usual RECORD header followed by three variables defined in the same manner as the records in the last two example programs. Then we come to the CASE statement. Following this statement, the record is different for each of the four types defined in the associated scalar definition. The variable "WhatKind" is called the tag-field and must be defined as a scalar type prior to the record definition. The tag-field is what selects the variant used, when the program uses one of the variables with this record type. The tag-field is followed by a colon and its type definition, then the reserved word OF. A list of the variants is then given, with each of the variants having the variables for its particular case defined. The list of variables for one variant is called the field list.

A few rules are in order at this point. The variants do not have to have the same number of variables in each field list, and in fact, one or more of the variants may have no variables at all in its variant part. If a variant has no variables, it must still be defined with a blank followed by a semi-colon. All variables in the entire variant part must have unique names. The three variables, "Wheels", "Tires", and "Tyres", all mean the same thing to the user, but they must be different for the compiler. You may use the same identifiers again in other records and for simple variables anywhere else in the program. The Modula-2 compiler can tell which variable you mean by its context. Using the same variable name should be discouraged as bad programming practice because it may confuse you or another person trying to understand your program at a later date.

USING THE VARIANT RECORD

We properly define four variables with the record type "Vehicle" and go on to examine the program itself.

We begin by defining one of our variables of type "Vehicle", namely the variable named "Ford". The seven lines assigning values to "Ford" are similar to the prior examples with the exception of the fourth line. In the fourth line the tag-field which selects the particular variant used is set equal to the value "Truck", which is a scalar definition, not a variable. This means that the variables named "Motor", "Tires", and "Payload" are available for use with the record "Ford", but the variables named "Wheels", "Engine", "Tyres", etc. are not available in the record named "Ford".

Next, lets define the record "Sunfish" as a boat, and define all of its variables. All of sunfish's variables are defined but in a rather random order to illustrate that they need not be defined in a particular order. Recall the use of WITH from the last example program.

To go even further in randomly assigning the variables to a record, we redefine "Ford" as having an "Engine" which it can only have if it is a car. This is one of the fine points of the record. If you assign any of the variant variables, the record is changed to that variant, but it is the programmers responsibility to assign the correct tag- field to the record, not the Modula-2 compiler's. Good programming practice would be to assign the tag-field before assigning any of the variant variables. The remainder of the "Ford" variables are assigned to complete that record, the non-variant part remaining from the last assignment.

The variable "Mac" is now set equal to the variable "Sunfish". All variables within the record are copied to "Mac" including the tag-field, making "Mac" a boat.

NOW TO SEE WHAT WE HAVE IN THE RECORDS

We have assigned "Ford" to be a car, and two boats exist, namely "Sunfish" and "Mac". Since "Schwinn" was never defined, it has no data in it, and is at this point useless. The "Ford" tag-field has been defined as a car, so it should be true in the IF statement, and the first message should print. The "Sunfish" is not a bicycle, so it will not print. The "Mac" has been defined as a boat in the single assignment statement, so it will print a message with an indication that all of the data in the record was transferred to its variables.

Even though we can make assignment statements with records, they cannot be used in any mathematical operations such as addition, or multiplication. They are simply used for data storage. It is true however, that the individual elements in a record can be used in any mathematical statements legal for their respective types.

One other point should be mentioned. The tag-field can be completely eliminated resulting in a "free union" variant record. This is possible because Modula-2, as you may remember from above, will automatically assign the variant required when you assign data to one of the variables within a variant. This is the reason that all variables within any of the variants must have unique names. The free union record should be avoided in your early programming efforts because you cannot test a record to see what variant it has been assigned to.

A NOTE TO PASCAL PROGRAMMERS

A record with a free union variant is commonly used in Pascal to do type transfers, but this should be discouraged in Modula-2 since it has a complete set of carefully defined type transfer functions for that purpose. In addition, the method of data storage is not specified as a part of the language and a free union would not operate the same way with different compilers if used for the purpose of type transfer.

PROGRAMMING EXERCISE

1. Write a simple program with a record to store the names of five of your friends and display the names.

Chapter 12 - Pointers and Dynamic Allocation

PREREQUISITES FOR THIS MATERIAL

In order to understand this chapter, you should have a good grasp of the entirety of Part I and a clear understanding of chapter 11.

For certain types of programs, pointers and dynamic allocation can be a tremendous advantage, but most programs do not need such a high degree of data structure. For that reason, it would probably be to your advantage to lightly skim over these topics and come back to them later when you have a substantial base of Modula-2 programming experience. It would be good to at least skim over this material rather than completely neglecting it, so you will have an idea of how pointers and dynamic allocation work and that they are available for your use when needed.

A complete understanding of this material will require deep concentration as it is very complex and not at all intuitive. Nevertheless, if you pay close attention, you will have a good grasp of pointers and dynamic allocation in a short time.

WHAT ARE POINTERS, AND WHAT GOOD ARE THEY?

If you examine POINTERS.MOD, you will see a very trivial example of pointers and how they are used. In the VAR declaration, you will see that the two variables have the two reserved words POINTER TO in front of their respective types. These are not actually variables, instead, they point to dynamically allocated variables that have not been defined yet, and they are called pointers. We will see, when we get to chapter 14, that a pointer can be used to point to any variable, even a statically defined one, but that will have to wait awhile.

```
(* Chapter 12 - Program 1 *)
MODULE Pointers;

FROM InOut   IMPORT WriteString, WriteInt, WriteLn;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;

TYPE Name = ARRAY[0..20] OF CHAR;

VAR  MyName : POINTER TO Name;     (* MyName points to a string *)
```

```
     MyAge  : POINTER TO INTEGER; (* MyAge points to an INTEGER *)

BEGIN

   ALLOCATE(MyAge,TSIZE(INTEGER));
   ALLOCATE(MyName,TSIZE(Name));

   MyAge^ := 27;
   MyName^ := "John Q. Doe";

   WriteString("My name is ");
   WriteString(MyName^);
   WriteString(" and I am ");
   WriteInt(MyAge^,2);
   WriteString(" years old.");
   WriteLn;

   DEALLOCATE(MyAge,TSIZE(INTEGER));
   DEALLOCATE(MyName,TSIZE(Name));

END Pointers.
```

The pointer "MyName" is a pointer to a 20 character string and is therefore not a variable into which a value can be stored. This is a very special TYPE, and it cannot be assigned a character string, only a pointer value or address. The pointer actually points to an address somewhere within the computer memory, and can access the data stored at that address.

Your computer has some amount of memory installed in it. If it is an IBM-PC or compatible, it can have up to 640K of RAM which is addressable by various programs. The operating system requires about 60K of the total, and your program can use up to 64K assuming that your compiler uses a small memory model. Adding those two numbers together results in about 124K. Any memory you have installed in excess of that is available for the stack and the heap. The stack is a standard DOS defined and controlled area that can grow and shrink as needed. Many books are available to define the stack if you are interested in more information on it.

The heap is a Modula-2 entity that utilizes otherwise unused memory to store data. It begins immediately following the program and grows as necessary upward toward the stack which is growing downward. As long as they never meet, there is no problem. If they meet, a run-time error is generated. The heap is therefore outside of the actual program space.

If you did not understand the last two paragraphs, don't worry. Simply remember that dynamically allocated variables are stored on the heap and do not count in the 64K limitation placed upon you by some compilers.

Back to our example program, POINTERS.MOD. When we actually begin executing the program, we still have not defined the variables we wish to use to store data in. The first executable statement in line 15 generates a variable for us with no name and stores it on the heap. Since it has no name, we cannot do anything with it, except for the fact that we do have a pointer "MyAge" that is pointing to it. By using the pointer, we can store any INTEGER in it, because that is its type, and later go back and retrieve it.

WHAT IS DYNAMIC ALLOCATION?

The variable we have just described is a dynamically allocated variable because it was not defined in a VAR declaration, but with an ALLOCATE procedure. The ALLOCATE procedure creates a variable of the type defined by the pointer, puts it on the heap, and finally assigns the address of the variable to the pointer itself. Thus "MyAge" contains the address of the variable generated. The variable itself is referenced by using the pointer to it followed by a ^, and is read, "the variable to which the pointer points".

The ALLOCATE procedure requires 2 arguments, the first of which is a pointer which will be used to point to the desired new block of dynamically allocated menory, and the second which gives the size of the block in bytes. The supplied function TSIZE will return the size of the block of data required by the TYPE supplied to it as an argument. Be sure to use the TYPE of the data and not the TYPE of the pointer to the data for the argument. Another procedure is available named SIZE which returns the size of any variable in bytes.

The next statement assigns a place on the heap to an ARRAY type variable and puts its address in "MyName". Following the ALLOCATE statements we have two assignment statements in which the two variables pointed at are assigned values compatible with their respective types, and they are both written out to the video display. Notice that both of these operations use the ^ which is the dereference operator. By adding the dereference operator to the pointer name, you can use the entire name just like any other variable name.

The last two statements are illustrations of the way the dynamically allocated variables are removed from use. When they are no longer needed, they are disposed of with the DEALLOCATE procedure, and the space on the heap is freed up for use by other dynamically allocated variables.

In such a simple program, pointers cannot be appreciated, but it is necessary for a simple illustration. In a large, very active program, it is possible to define many variables, dispose of some of them, define more, and dispose of more, etc. Each time some variables are deallocated, their space is then made available for additional variables defined with the ALLOCATE procedure.

The heap can be made up of any assortment of variables, they do not have to all be the same. One thing must be remembered. Anytime a variable is defined, it will have a pointer pointing to it. The pointer is the only means by which the variable can be accessed. If the pointer to the variable is lost or changed, the data itself is lost for all practical purposes.

WHAT ABOUT THE "NEW" AND "DISPOSE" PROCEDURES?

The NEW and DISPOSE procedures are a carryover from Pascal and are available on some Modula-2 compilers. When they are available, they are simply translated internally into calls to ALLOCATE and DEALLOCATE which must be imported in order to use NEW and DISPOSE. Since they are being removed from the language definition, their use should be discouraged in favor of the more universal ALLOCATE and DEALLOCATE procedures.

DYNAMICALLY STORING RECORDS;

The next example program, DYNREC.MOD, is a repeat of one we studied in an earlier chapter. For your own edification, review the example program BIGREC.MOD before going ahead in this chapter.

```
(* Chapter 12 - Program 2 *)
MODULE DynRec;

FROM InOut   IMPORT WriteString, Write, WriteLn;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;

CONST  NumberOfFriends = 50;

TYPE   FullName = RECORD
         FirstName : ARRAY[0..12] OF CHAR;
         Initial   : CHAR;
         LastName  : ARRAY[0..15] OF CHAR;
       END;

       Date = RECORD
         Day   : CARDINAL;
         Month : CARDINAL;
         Year  : CARDINAL;
       END;

       PersonID = POINTER TO Person;
       Person = RECORD
         Name      : FullName;
         City      : ARRAY[0..15] OF CHAR;
         State     : ARRAY[0..3] OF CHAR;
         BirthDay  : Date;
       END;

VAR   Friend  : ARRAY[1..NumberOfFriends] OF PersonID;
      Self, Mother, Father : PersonID;
      Temp    : Person;
      Index   : CARDINAL;

BEGIN  (* Main program *)
   ALLOCATE(Self,TSIZE(Person));    (* Create a dynamically
                                       allocated variable *)
   Self^.Name.FirstName := "Charley ";
   Self^.Name.Initial := 'Z';
   Self^.Name.LastName := " Brown";
   WITH Self^ DO
      City := "Anywhere";
      State := "CA";
      BirthDay.Day := 17;
      WITH BirthDay DO
         Month := 7;
         Year := 1938;
      END;
   END;     (* All data for Self is now defined *)

   ALLOCATE(Mother,TSIZE(Person));
   Mother := Self;

   ALLOCATE(Father,TSIZE(Person));
   Father^ := Mother^;

   FOR Index := 1 TO NumberOfFriends DO
      ALLOCATE(Friend[Index],TSIZE(Person));
      Friend[Index]^ := Mother^;
   END;

   Temp := Friend[27]^;
   WriteString(Temp.Name.FirstName);
   Write(Self^.Name.Initial);
   WriteString(Mother^.Name.LastName);
   WriteLn;

   DEALLOCATE(Self,TSIZE(Person));
(* DEALLOCATE(Mother,TSIZE(Person)); Since Mother is lost, it cannot
                                               be disposed of *)
   DEALLOCATE(Father,TSIZE(Person));
   FOR Index := 1 TO NumberOfFriends DO
      DEALLOCATE(Friend[Index],TSIZE(Person));
   END;

END DynRec.
```

Assuming that you are back in DYNREC.MOD, you will notice that this program looks very similar to the earlier one, and in fact they do exactly the same thing. The only difference in the TYPE declaration is the addition of a pointer "PersonID", and in the VAR declaration, the first four variables are defined as pointers here, and were defined as record variables in the last program.

WE JUST BROKE THE GREAT RULE OF MODULA-2

Notice in the TYPE declaration that we used the identifier "Person" before we defined it, which is illegal in Modula-2. Foreseeing the need to define a pointer prior to the record, the designer of Modula-2 allows us to break the rule in this one place. The pointer could have been defined after the record in this case, but it was more convenient to put it before, and in the next example program, it will be required to put it before the record. We will get there soon.

Examining the VAR declaration, we see that "Friend" is really 50 pointers, so we have now defined 53 different pointers to records, but no variables other than "Temp" and "Index". We dynamically allocate a record with "Self" pointing to it, and use the pointer to fill the new record. Compare the statements that fill the record with the corresponding statements in "BIGREC" and you will see that they are identical except for the addition of the ^ to each use of the pointer to designate the data pointed to.

THIS IS A TRICK, BE CAREFUL

Now go down to the place where "Mother" is assigned a record and is then pointing to the record. It seems an easy thing to do then to simply assign all of the values of "Self" to all the values of "Mother" as shown in the next statement, but it doesn't work. All the statement does, is make the pointer "Mother" point to the same place where "Self" is pointing. The data space that was allocated to the pointer "Mother" is now somewhere on the heap, but since we have lost the original pointer to it, we cannot find it, use it, or deallocate it. This is an example of losing data on the heap. The proper way is given in the next two statements where all fields of "Father" are defined by all fields of "Mother" which is pointing at the original "Self" record. Note that since "Mother" and "Self" are both pointing at the same record, changing the data with either pointer results in the data appearing to be changed in both because there is, in fact, only one data field.

A NOTE FOR PASCAL PROGRAMMERS

In order to WRITE from or READ into a dynamically assigned record it is necessary to use a temporary record since dynamically assigned records are not allowed to be used in I/O statements in Pascal. This is not true in Modula-2, and you can write directly out of a dynamically allocated record in Modula-2. This is illustrated in the section of the program that writes some data to the monitor.

Finally, the dynamically allocated variables are deallocated prior to ending the program. For a simple program such as this, it is not necessary to deallocate them because all dynamic variables are deallocated automatically when the program is terminated, but the DEALLOCATE steps are included for illustration. Notice that if the "DEALLOCATE(Mother)" statement was included in the program, the data could not be found due to the lost pointer, and the program would be unpredictable, probably leading to a system crash.

SO WHAT GOOD IS THIS ANYWAY?

Remember when you were initially studying BIGREC? I suggested that you see how big you could make the constant "NumberOfFriends" before you ran out of memory. At that time you probably found that it could be made slightly greater than 1000 before you got the memory overflow message at compilation. If your compiler uses a large memory model, you may have been able to go much larger. Try the same thing with DYNREC to see how many records it can handle, remembering that the records are created dynamically, so you will have to run the program to actually run out of memory. The final result will depend on how much memory you have installed, and how many memory resident programs you are using such as "Sidekick". If you have a full memory of 640K, I would suggest you start somewhere around 8000 records of "Friend".

Now you should have a good idea of why Dynamic Allocation can be used to greatly increase the usefulness of your programs. There is, however, one more important topic we must cover on dynamic allocation. That is the linked list.

WHAT IS A LINKED LIST?

Understanding and using a linked list is by far the most baffling topic you will confront in Modula-2. Many people simply throw up their hands and never try to use a linked list. I will try to help you understand it by use of an example and lots of explanation. Examine the program LINKLIST.MOD for an example of a linked list. I tried to keep it short so you could see the entire operation and yet do something meaningful.

```
(* Chapter 12 - Program 3 *)
MODULE LinkList;

FROM InOut   IMPORT WriteString, Write, WriteLn;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;

TYPE  NextPointer = POINTER TO FullName;
      FullName = RECORD
        FirstName : ARRAY[0..12] OF CHAR;
        Initial   : CHAR;
        LastName  : ARRAY[0..15] OF CHAR;
        Next      : NextPointer;
      END;

VAR   StartOfList : NextPointer;
      PlaceInList : NextPointer;
      TempPlace   : NextPointer;
      Index       : CARDINAL;

BEGIN   (* Main Program *)

                  (* Generate the first name in the list *)

   ALLOCATE(PlaceInList,TSIZE(FullName));
   StartOfList := PlaceInList;
   PlaceInList^.FirstName := "John ";
   PlaceInList^.Initial := 'Q';
   PlaceInList^.LastName := " Doe";
   PlaceInList^.Next := NIL;

                  (* Generate another name in the list *)

   TempPlace := PlaceInList;
   ALLOCATE(PlaceInList,TSIZE(FullName));
   TempPlace^.Next := PlaceInList;
   PlaceInList^.FirstName := "Mary ";
   PlaceInList^.Initial := 'R';
   PlaceInList^.LastName := " Johnson";
   PlaceInList^.Next := NIL;

                 (* Add 10 more names to complete the list *)

   FOR Index := 1 TO 10 DO
      TempPlace := PlaceInList;
      ALLOCATE(PlaceInList,TSIZE(FullName));
      TempPlace^.Next := PlaceInList;
      PlaceInList^.FirstName := "Billy ";
      PlaceInList^.Initial := 'R';
      PlaceInList^.LastName := " Franklin";
      PlaceInList^.Next := NIL;
   END;

                   (* Display the list on the video monitor *)

   PlaceInList := StartOfList;
   REPEAT
      WriteString(PlaceInList^.FirstName);
      Write(PlaceInList^.Initial);
      WriteString(PlaceInList^.LastName);
      WriteLn;
      TempPlace := PlaceInList;
      PlaceInList := PlaceInList^.Next;
   UNTIL TempPlace^.Next = NIL;
END LinkList.
```

To begin with, notice that there are two TYPEs defined, a pointer to the record and the record itself. The record, however, has one thing about it that is new to us, the last entry, "Next" is a pointer to this very record. This record then, has the ability to point to itself, which would be trivial and meaningless, or to another record of the same type which would be extremely useful in some cases. In fact, this is the way a linked list is used. I must point out, that the pointer to another record, in this case called "Next", does not have to be last in the list, it can be anywhere it is convenient for you.

A couple of pages ago, we discussed the fact that we had to break the great rule of Modula-2 and use an identifier before it was defined. This is the reason the exception to the rule was allowed. Since the pointer points to the record, and the record contains a reference to the pointer, one has to be defined after being used, and by rules of Modula-2, the pointer can be defined first. That is a mouthful but if you just use the syntax shown in the example, you will not get into trouble with it.

STILL NO VARIABLES?

It may seem strange, but we still will have no variables defined, except for our old friend "Index". In fact for this example, we will only define 3 pointers. In the last example we defined 54 pointers, and had lots of storage room. Before we are finished, we will have at least a dozen pointers but they will be stored in our records, so they too will be dynamically allocated.

Lets look at the program itself now. First, we create a dynamically allocated record and define it by the pointer "PlaceInList". It is composed of the three data fields, and another pointer. We define "StartOfList" to point to the first record created, and we will leave it unchanged throughout the program. The pointer "StartOfList" will always point to the first record in the linked list which we are building up.

We define the three variables in the record to be any name we desire for illustrative purposes, and set the pointer in the record to NIL. NIL is a reserved word that doesn't put an address in the pointer but defines it as empty. A pointer that is currently NIL cannot be used to write a value to the display as it has no value, but it can be tested in a logical statement to see if it is NIL. It is therefore a dummy assignment. With all of that, the first record is completely defined.

DEFINING THE SECOND RECORD

When you were young you may have played a searching game in which you were given a clue telling you where the next clue was at. The next clue had a clue to the location of the third clue. You kept going from clue to clue until you found the prize. You simply exercised a linked list. We will now build up the same kind of a list in which each record will tell us where the next record is at.

We will now define the second record. Our goal will be to store a pointer to the second record in the pointer field of the first record. In order to keep track of the last record, the one in which we need to update the pointer, we will keep a pointer to it in "TempPlace". Now we can create another new record and use "PlaceInList" to point to it. Since "TempPlace" is still pointing at the first record, we can use it to store the value of the pointer to the new record in the old record. The 3 data fields of the new record are assigned nonsense data for our illustration, and the pointer field of the new record is assigned NIL.

Lets review our progress to this point. We now have the first record with a name, composed of 3 parts, and a pointer to the second record. We also have a second record storing a different name and a pointer assigned NIL. We also have three pointers, one pointing to the first record, one pointing to the last record, and one we used just to get here since it is only a temporary pointer. If you understand what is happening so far, lets go on to add some additional records to the list. If you are confused, go back over this material again.

TEN MORE RECORDS

The next section of code is contained within a FOR loop so the statements are simply repeated ten times. If you observe carefully, you will notice that the statements are identical to the second group of statements in the program (except of course for the name assigned). They operate in exactly the same manner, and we end up with ten more names added to the list. You will now see why the temporary pointer was necessary, but pointers are cheap, so feel free to use them at will. A pointer only uses 4 bytes of memory.

We now have generated a linked list of twelve entries. We have a pointer pointing at the first entry, and another pointer pointing at the last. The only data stored within the program itself are three pointers, and one integer, all of the dynamically allocated data is on the heap. This is one advantage to a linked list, it uses very little internal memory, but it is costly in terms of programming. You should never use a linked list simply to save memory, but only because a certain program lends itself well to it. Some sorting routines are extremely fast because of using a linked list, and it could be advantageous to use in a database.

A graphic picture of the data should aid in your understanding of what we have done so far.

```
StartOfList-->FirstName    (first Record)              Initial
LastName              Next---->FirstName   (Second Record)
Initial                       LastName
Next---->FirstName    (Third Record) Note; The pointer
Initial   actually points to          LastName   all 4 elements of
Next----> etc.   the record.               .
.                              .                              .
etc.--->FirstName    (Record 11)                   Initial
LastName                   Next---->FirstName    (Record 12)
Initial        PlaceInList------->LastName
Next---->NIL
```

HOW DO WE GET TO THE DATA NOW?

Since the data is in a list, how can we get a copy of the fourth entry for example? The only way is to start at the beginning of the list and successively examine pointers until you reach the desired one. Suppose you are at the fourth and then wish to examine the third. You cannot back up, because you didn't define the list that way, you can only start at the beginning and count to the third. You could have defined the record with two pointers, one pointing forward, and one pointing backward. This would be a doubly-linked list and you could then go directly from entry four to entry three.

Now that the list is defined, we will read the data from the list and display it on the video monitor. We begin by defining the pointer, "PlaceInList", as the start of the list.

Now you see why it was important to keep a copy of where the list started. In the same manner as filling the list, we go from record to record until we find the record with NIL as a pointer.

Finally, it is necessary to DEALLOCATE the list, being careful to check for the ending NIL before you deallocate it. It will be left for you to DEALLOCATE the records if you have such a desire.

There are entire books on how to use linked lists, and many Modula-2 programmers will seldom, if ever, use them. For this reason, additional detail is considered unnecessary, but to be a fully informed Modula-2 programmer, some insight into linked lists is necessary.

PROGRAMMING EXERCISE

1. Write a program to store a few names dynamically, then display the stored names on the monitor. As your first exercise in dynamic allocation, keep it very simple.

2. For a much more involved project, read in a list of simple names and sort them alphabetically by searching through the list to find where they should go. Insert each new name into the list by changing pointer values. For example, to add a new element between number 3 and 4, make the pointer in 3 point to the new element, and make the pointer in the new element point to number 4. It is important to note that adding data to the beginning or end of the list must be handled as special cases. This is definitely an advanced programming exercise but you will be greatly rewarded for your effort if you complete it.

Chapter 13 - Modules, Local and Global

PREREQUISITES FOR THIS MATERIAL

Before attempting to understand this material, you should have a good grasp of the principles taught in Part I of this tutorial. None of the material from Part II is required to do a meaningful study of modules in Modula-2.

WHAT GOOD ARE MODULES?

Modules are the most important feature of Modula-2 over its predecessor Pascal making it very important for you to understand what they are and how they work. Fortunately for you, there are not too many things to learn about them and after you master them you will find many uses for them as you develop programs, and especially large programs.

Load and display the program named LOCMOD1.MOD for your first example of a program with an embedded module. Modules are nothing new to you because every program you have examined has been a module. At this time, however, we will introduce a local module.

```
(* Chapter 13 - Program 1 *)
MODULE LocMod1;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR Index : CARDINAL;

       MODULE LocalStuff;
       EXPORT GetNumber;    (* Nothing else is visible outside *)
                            (* Nothing outside is visible here *)
       VAR Counter : CARDINAL;

              PROCEDURE GetNumber() : CARDINAL;
              BEGIN
                 Counter := Counter + 3;
                 RETURN Counter;
              END GetNumber;

       BEGIN
       Counter := 4;    (* This is only run at load time *)
       END LocalStuff;
```

```
BEGIN      (* Main program *)
   FOR Index := 1 TO 8 DO
      WriteString("The count is now ");
      WriteCard(GetNumber(),8);
      WriteLn;
   END;    (* Do loop *)
END LocMod1.
```

WHAT IS A LOCAL MODULE?

A local module is simply a module nested within another module, just like the example on your monitor at this time. The module named "LocalStuff" is nested within the main module and is heavily indented for clarity. Since nothing is imported into the local module, nothing that belongs to the main module can be seen from within the nested module. In addition, since the procedure "GetNumber" is the only thing exported from the local module, nothing else is available to the main module. In effect, the local module is an impenetrable wall through which nothing can pass without the benefit of the IMPORT and EXPORT list. In this case, the variable "Counter" cannot be modified in any way by the main module, either intentionally or accidentally and the procedure "GetNumber" will very stubbornly refuse to allow any flexibility in its output, adding three to its internally stored variable each time it is called. It may seem to you that this result can be accomplished easily by using another procedure without the module but we will see shortly that it will not be the same.

THE BODY OF THE LOCAL MODULE

The body of the local module has one statement contained within it, "Counter := 4;", that is executed only when the module is loaded, and at no other time. This is therefore an initialization section for the module. Any valid statements can be put here and they will be executed when the program is loaded, or you can omit the body altogether by omitting the BEGIN and any statements. Actually, this body is no different than the body of the main program since it too is executed one time when the program is loaded, except for the fact that the main program is required to have a body or you will have no program.

THE MODULE VERSUS THE PROCEDURE

We must digress a bit to see the difference in these two important topics in Modula-2. A procedure is an executable section of code whereas a module is a grouping of variables, constants, types, and procedures. A module is never executed since it is simply a grouping identifier.

The variables in a procedure do not exist when it is not being executed, but instead are generated dynamically when the procedure is called. A variable therefore, has a lifetime associated with it in addition to a "type". This may seem strange to you but if you think about it for awhile, it will help explain how recursive procedure calls work. The module, on the other hand, exists anytime its surrounding code exists, in this case, the main program. Since the module always exists, the variable "Counter" also always exists because it is defined as a part of the module. If this variable were defined within a procedure, it would be automatically regenerated every time the procedure were called and would therefore not remember the value it contained the prior time the procedure was called. We could choose to define the variable as global and it would therefore always be available and never regenerated, but we would be left with the possibility of anything in the program modifying it either accidentally or on purpose. In a program as small as this one, it would not be a problem, but it is intended to illustrate the solution to a problem embedded in a much larger program.

Suppose, for example, that you wished to generate random numbers for some use within a program. You could include all of the code within a module using the module body for the seed initialization, and a procedure to generate one random number each time it was called. The structure would be essentially the same as that given here, but the actual code would be different. Nothing in the main program or any of its procedures could in any way corrupt the job given to the random number generator.

BACK TO THE PROGRAM ON YOUR MONITOR

In this case we have one local module defined within the main module but as many as desired could be used, and we have one procedure in the local module whereas we could have as many as desired. In fact, we could have local modules embedded in a procedure, or in other local modules. There is no real limit as to how you can structure your program to achieve the desired results. One thing must be remembered. If you embed a local module within a procedure, all of its variables are defined dynamically each time the procedure in which it is embedded is called, and its body is also executed each time. This can be used to advantage in some situations, but it would be best to leave this construct to the future when you have more experience with Modula-2.

In the body of the main module you will find nothing new except for the call to the function procedure "GetNumber()" which is actually nothing new except that it is embedded in the local module "LocalStuff". Compile and run the program to see if it does what you expect it to do.

TWO LOCAL MODULES

It would be well to point out at this time that if you define two local modules at the same level, one could EXPORT a variable, procedure, constant, or type and the other could IMPORT it and use it in any legal fashion. You therefore have the ability to very carefully define the mechanism by which the two modules interact.

ANOTHER LOCAL MODULE

The program we have been inspecting had the procedure exported without qualification, so it could only be referred to by its simple name. This could have led to a naming conflict which can be solved by using a qualified export as is done in the next program. Load and display the program named LOCMOD2.MOD. This program is very similar to the last except for moving the output statements to the procedure.

```
(* Chapter 13 - Program 2 *)
MODULE LocMod2;

FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR Index : CARDINAL;

     MODULE MyStuff;
     IMPORT WriteString, WriteCard, WriteLn;
     EXPORT QUALIFIED WriteStuff;
     VAR Counter : CARDINAL;
          PROCEDURE WriteStuff;
          BEGIN
             Counter := Counter + 3;
             WriteString("The value of the counter is ");
             WriteCard(Counter,8);
             WriteLn;
          END WriteStuff;
     BEGIN
        Counter := 4;
     END MyStuff;

BEGIN      (* Main program *)
   FOR Index := 1 TO 8 DO
      MyStuff.WriteStuff;
   END;
END LocMod2.
```

First, you should notice that the procedure name is exported using "EXPORT QUALIFIED" which allows the use of the qualified call to the procedure in line number 25. There can never be a conflict of names in calling a procedure this way because it is illegal to use the same name for a module more than once at any level. In a local module, you have a choice of using either qualified or unqualified export of items, but the exported items must all be of the same export type because only one export list is allowed per module.

IMPORTING INTO A LOCAL MODULE

The three output procedures are used in the local module "MyStuff", but because it is only permissible to import items from a module's immediate surroundings, the procedures must first be imported into the main module.

The procedure named "WriteStuff" is even more tightly controlled than that in the last program because this one doesn't even return a value to the calling program. It updates its own internally stored value, displays it, and returns control to the calling program.

Compile and run this program, then we will go on to global modules.

GLOBAL MODULES

As useful as local modules are, they must take a back seat to the global module with which you are already fairly familiar because you have been using them throughout this tutorial. The modules "InOut", "Terminal", and "FileSystem" are examples of global modules that you already know how to use. Now you will learn how to write your own global modules that can be called in exactly the same way as these standard modules from any program.

YOUR FIRST DEFINITION MODULE

In order to get started, load and display the program named CIRCLES.DEF on your monitor. The first thing you will notice is that we used a different extension for this program because there is another part to the program with the same name but the usual extension "MOD". What you have displayed on your screen is the definition part of the global module and it serves two very important purposes. First, it defines the interface you need to use the module in one of your programs, and it defines the details of the interface for the compiler so it can do type checking for you when you call this module. The Modula-2 compiler uses the information contained here to check all types and numbers of variables just like it would do in a singly compiled program.

```
(* Chapter 13 - Program 3 *)
DEFINITION MODULE Circles;

EXPORT QUALIFIED AreaOfCircle, PerimeterOfCircle;

PROCEDURE AreaOfCircle(Radius : REAL; VAR Area : REAL);

PROCEDURE PerimeterOfCircle(Radius : REAL; VAR Perim : REAL);

END Circles.
```

The program on your monitor does very little. In fact its purpose is to do nothing because there are no executable statements in it. It is only to define the interface to the actual program statements contained elsewhere. Notice that the procedures are exported using the qualified option. All identifiers that are exported from a definition module must be qualified so that the user has the option of importing them either way. It is legal to export procedures, variables, constants, or types for use elsewhere as needed for the programming problem at hand, but the majority of exported items are procedures. It should be obvious that nothing within the module is available to any other part of the program unless it is exported.

THE IMPLEMENTATION MODULE

We are not finished with the definition part of the module yet but we will look at the implementation part of it for a few moments. Load the program named CIRCLES.MOD and display it on your monitor. This is the part of the module that actually does the work. Notice that there are three procedures here, two of which were defined in the definition part of the module making them available to other programs. The procedure named "GetPi" is a hidden or private procedure that is only available for use within this module. The other two procedures are available to any program that wishes to use them simply by importing them.

```
(* Chapter 13 - Program 3 *)
IMPLEMENTATION MODULE Circles;

PROCEDURE GetPi(VAR Pi : REAL);
BEGIN
```

```
   Pi := 3.14159;
END GetPi;

PROCEDURE AreaOfCircle(Radius : REAL; VAR Area : REAL);
VAR Pie : REAL;
BEGIN
   GetPi(Pie);
   Area := Pie * Radius * Radius;
END AreaOfCircle;

PROCEDURE PerimeterOfCircle(Radius : REAL; VAR Perim : REAL);
VAR Cake : REAL;
BEGIN
   GetPi(Cake);
   Perim := 2.0 * Cake * Radius;
END PerimeterOfCircle;

BEGIN            (* IMPLEMENTATION MODULE body, empty in this case *)
END Circles.
```

Anything defined in the definition part of the module is also available here for use without redefining it, except for the procedure headers which must be completely defined in both places. Anything imported into the definition part of the module must also be imported here if it will be used in this module, imported identifiers are not automatically transferred into this part of the module.

MORE ABOUT THE USE OF TWO PARTS

The definition part of the module defines the public information about the module and the implementation part of the module defines the private or hidden information about the module. It may seem sort of silly to go to the trouble of separating a module into two parts but there are at least three good reasons to do so.

1. You may not care how the module is implemented.

In all of the programs we have run up to this point, you probably didn't care how the "WriteString" procedure did its job. You only wanted it to do the job it was supposed to do to aid you in learning to use Modula-2 efficiently. It would have been senseless to have cluttered your monitor with the details of how it worked every time you wanted to know how to use it.

2. It hides details of implementation.

If you were working on a large programming project and you were assigned to job of writing a procedure for others to use that did some well defined task, you would define the interface carefully and be finished. If, however, one of the users studied your detailed code and found a way to trick it into doing something special, he may use the trick in his part of the program. If you then wanted to improve your routine and remove the code that allowed the trick, the interface would no longer work. To prevent this, you give others only the interface to work with and they cannot look for tricks. This is called "information hiding" and is a very important technique which is used on large projects.

3. It allows for orderly development.

It is possible to define all of the definition parts of the modules and have all members of the development team agree to the interface. Long before the details of the individual procedures are worked out, the entire team knows what each procedure will do and they can all begin work on their respective parts of the overall system. This is very effective when used on a large team effort.

COMPILATION ORDER IS IMPORTANT

In order for the above principles to work effectively, a very definite order of compilation must be adhered to. If the identifiers declared in the definition part are automatically available in the implementation part of the module, then it is obvious that the definition part must be compiled before the implementation part of the module can be compiled. Also, if the definition part is modified and recompiled, then the implementation part may also require modifications to comply with the changes and it must also be recompiled.

The next rule is not nearly so obvious but you will understand it when we explain it. When a calling module is compiled, it checks each of the imported identifiers to see that the types and number of variables agree with the calling sequences used in the program. This is part of the strong "type checking" done for you by Modula-2. If you modify and recompile one of the called definition modules and attempt to relink the program together, you may have introduced a type incompatibility. In order to prevent this, Modula-2 requires you to recompile every module that calls a modified definition module. It does this by generating a "key" when you compile a definition module and storing the "key" when you compile the calling module. If you attempt to link a program with differing "keys", this indicates that the definition module was changed, resulting in a new "key" and hence a mismatch, and the linker will generate an error.

WHY ALL OF THIS TROUBLE?

It may not seem to be worth all of the extra trouble that the Modula-2 compiler and linker go through to do this checking but it is important for a large program. The information used in the definition part of the module is the type of information that should be well defined in the design stages of a programming project, and if well done, very few or no changes should be required during the coding phase of the project. Therefore it is expected that recompiling several definition modules should not happen very often. On the other hand, during the coding and debugging phase of the project, it is expected that many changes will be required in the implementation parts of the modules. Modula-2 allows this and still maintains very strong type checking across module boundaries to aid in detecting sometimes very subtle coding errors.

The above paragraph should be interpreted as a warning to you. If you find that you are constantly recompiling modules due to changes in the definition modules, you should have spent more time in the software design.

NOW TO ACTUALLY USE IT ALL

With all of that in mind, it will be necessary for you to reload the program named CIRCLES.DEF which is the definition part of the module, and compile it. Your compiler will generate several different files for use in cross checking. After you get a good compile, reload the program named CIRCLES.MOD which is the implementation part of the module and compile it. During this compile, some of the files generated by CIRCLES.DEF will be referred to. It would be an interesting exercise to modify a procedure call in one of the programs to see what kind of an error is displayed. After a good compile on both of these modules, you have a new module in your library that can be used just like any of the other global libraries that came with your compiler.

Load and display the program GARDEN.MOD for an example of a program that calls your new library or global module. This program is very simple and should pose no problem in understanding for you. The two new procedures are imported and used just like any other procedure. Compile and run this program.

```
(* Chapter 13 - Program 4 *)
MODULE Garden;

FROM InOut IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteReal;
FROM Circles IMPORT AreaOfCircle, PerimeterOfCircle;
```

```
VAR Radius, Area, Circumference : REAL;

BEGIN    (* Main program *)
   Radius := 12.0;
   AreaOfCircle(Radius,Area);
   PerimeterOfCircle(Radius,Circumference);

         (* Calculations complete - output results *)

   WriteString("Radius = ");
   WriteReal(Radius,12);
   WriteString("     Area =");
   WriteReal(Area,12);
   WriteLn;
   WriteString("Circumference = ");
   WriteReal(Circumference,12);
   WriteLn;
END Garden.
```

A FINAL WORD ABOUT GLOBAL MODULES

From the above description of global modules, it may not be very obvious to you that it is perfectly legal for one global module to call another which in turn calls another, etc. Program structure is entirely up to you. For example, we could have called "WriteString" and some of our other familiar procedures from within the "AreaOfCircle" procedure. The order of compilation must be kept in mind or you will not get a good compilation and linking of your completed program.

Remember that there is nothing magic about the global or library (the names are synonymous) modules supplied with your compiler. They are simply global modules that have already been programmed and debugged for you by the compiler writer. This is probably a good time to mention to you that you may have only received the source code for the definition part of the library modules with your compiler. Many compiler writers will supply the source code for the implementation part of the library modules only if you supply them with a little more money. After all, they are in business for the money and most people never wish to modify the supplied routines but are happy to use them as is. All compiler writers will supply you with the definition part of the library modules because they are your only means of interfacing with them.

THE PROCEDURE TYPE, SOMETHING NEW

Load and display the program named PROCTYPE.MOD on your monitor for an example of a procedure TYPE. In line 6 we define a variable "OutputStuff" to be a PROCEDURE type of variable that requires an "ARRAY OF CHAR" as an argument.

This variable name can now be used to refer to any procedure that uses a single "ARRAY OF CHAR" as an argument.

```
(* Chapter 13 - Program 5 *)
MODULE ProcType;

FROM InOut IMPORT WriteString, WriteLn;

VAR OutputStuff : PROCEDURE(ARRAY OF CHAR);
    VarLine     : ARRAY[0..50] OF CHAR;

     PROCEDURE WriteWithNote(InString : ARRAY OF CHAR);
     BEGIN
        WriteString("Written with a note ---> ");
        WriteString(InString);
        WriteLn;
     END WriteWithNote;

     PROCEDURE WriteWithComment(InputLine : ARRAY OF CHAR);
     BEGIN
        WriteString(InputLine);
        WriteString(" <--- Written with a comment");
        WriteLn;
     END WriteWithComment;

BEGIN
   VarLine := "This is a line of data.";
                                        (* This uses WriteWithNote *)
   OutputStuff := WriteWithNote;
   OutputStuff(VarLine);
   OutputStuff("Extra output ");
                                      (* This uses WiteWithComment *)
   OutputStuff := WriteWithComment;
   OutputStuff(VarLine);
                                          (* This uses WriteString *)
   OutputStuff := WriteString;
   OutputStuff(VarLine);
   OutputStuff(" End of the line");
   WriteLn;
             (* The Procedures can be used in normal fashion too *)
   WriteLn;
   WriteWithNote("This is straight output.");
   WriteWithComment("This too is straight output.");
   WriteString(VarLine);
END ProcType.
```

In the definition part of the program two procedures are defined, each of which uses a single "ARRAY OF CHAR" as an argument. Then in the main program the variable "OutputStuff" is assigned each of the new procedures and used to call them. In addition, it is used to call the supplied procedure "WriteString" to illustrate the possibility of doing so. Finally, the procedures are all called in their normal manner to illustrate that there is nothing magic about them. Any procedure type can be used to call any procedures that use the same type of parameter calls as those defined when it is created as a variable.

Chapter 14 - Machine Dependent Facilities

PREREQUISITES FOR THIS MATERIAL

Before attempting to understand this material, you should understand the material presented in Part I of this tutorial and a clear understanding of the material on pointers in Part II.

THIS IS WHERE YOU CAN GET INTO TROUBLE

Modula-2 does a good job of insulating you from the underlying peculiarities of your computer due to the strong TYPE checking which it does. It can prevent you from making many kinds of rather stupid blunders simply by forcing you to follow its predefined conventions. There are times, however, when you wish to ignore some of its help and do something that is out of the ordinary. If you had a need to directly interface with some external device, you would need to get down to the nitty gritty of the operating system and do some things that are outside of the realm of normal programming practice. Modula-2 will allow you to do such things but you will pay a price because you take the chance of hopelessly confusing the system.

The principles taught in this chapter can lead you directly into the operating system where you will have more freedom than you would have thought possible with Modula-2, but it will place more responsibility on you. This material is only for the advanced programmer because it will require a knowledge of the inner workings of the computer and the operating system. Nevertheless, it would be good for you, as a student of Modula-2, to at least read this material, examine the example programs, and compile and run them. You will then have a store of knowledge of these things so that you can use them when you need them.

TYPE RELAXATION EXAMPLE

Load and display the program named TYPEREL.MOD for an example of a program with some very unusual type transfer functions. Note first that three TYPES are defined, each being the same size considering storage requirements. The first TYPE is 10 INTEGERS, the second is 10 CARDINALS, and the third is 20 CHAR variables which requires the same amount of storage as 10 INTEGERS or CARDINALS. The fact that all three TYPES are the same size is very important for what we will do later in the program.

```
(* Chapter 14 - Program 1 *)
MODULE TypeRel;

FROM InOut IMPORT WriteCard,WriteLn;

TYPE IntType  = ARRAY[1..10] OF INTEGER;
     CardType = ARRAY[81..90] OF CARDINAL;
     CharType = ARRAY[1..20] OF CHAR;

VAR  IntVars  : IntType;
     CardVars : CardType;
     CharVars : CharType;
     Index    : INTEGER;
     Count    : CARDINAL;

BEGIN

   Count := 10;
   Index := INTEGER(Count);
   FOR Count := 1 TO 10 DO
      IntVars[Count] := INTEGER(Count) + 64;
   END;
   CardVars[81] := CARDINAL(IntVars[1]);
   CardVars := CardType(IntVars);
   CharVars := CharType(CardVars);

   FOR Index := 81 TO 85 DO
      WriteCard(CardVars[Index],8);
   END;

   WriteLn;

   FOR Index := 1 TO 10 DO
      Count := ORD(CharVars[Index]);
      WriteCard(Count,4);
   END;

   WriteLn;

END TypeRel.
```

The first thing we do in the program part of the MODULE is to assign a number to "Count", a CARDINAL type variable. In the next line we assign the value of "Count" to "Index" even though they are of different TYPES because we transform the TYPE in the same manner that we did back in Chapter 3 when we studied TRANSFER.MOD but this time we will go a bit farther with the transformations. Actually, we don't need the type transformation here because INTEGER and CARDINAL are assignment compatible.

We load up the INTEGER array "IntVars" with some nonsense data to work with, the data being the series of numbers from 65 to 74, which should be easy for you to ascertain. Then in line 23, we copy one of the array data points to one of the other to illustrate that the type transformation works even on array elements.

NOW FOR THE BIG TYPE TRANSFORMATION

In line 24 of the program we copy the entire field of 10 INTEGER type variables into the array of 10 CARDINAL type variables. The only restriction is that both of the fields must be exactly the same size which these two are. In order to do the transformation, the TYPE of the resulting data area is used in front of the parentheses of the source variables. Line 25 goes a step farther and copies the new CARDINAL data into 20 CHAR type variables, which is permissible because 20 CHAR variables uses the same amount of storage as 10 CARDINAL variables. You could even transform a record made up of several different types into all CHAR variables, or all INTEGERS, or even another completely different record. The only requirement is that both of the groups be exactly the same size.

This may appear to be a really neat thing to be able to do but there are problems that you will find with this new transformation. There are no data conversions done, only type conversions, which means that you may wind up with a real mess trying to decipher just what the transformed data means. In addition, since each compiler may define the various types of data slightly different, your program will not be easily transportable to another computer, or maybe not even to another compiler on the same computer.

Five of the CARDINAL numbers are displayed on the monitor, then 10 of the CHAR numbers are displayed to show you that they really are the same numbers. The order of the numbers are reversed when output as individual bytes because of the way the data is stored in the microprocessor in your computer. This in itself is an indication that there is no data conversion, but only a data copying, byte by byte.

One other rule must be pointed out, you cannot do a data transformation within a function call, but it is simple enough to do the transformation to a dummy variable and use the dummy variable in the function call if that is necessary. This will be illustrated shortly. Compile and run this program after you study it.

WORD AND ADDRESS VARIABLES

Load and display WORDADDR.MOD for an example using some new data types. In order to get down to the lowest level of the machine, we need these new types, ADR, WORD, and ADDRESS, which must be IMPORTED from the pseudo module SYSTEM. The pseudo module SYSTEM does not exist as an external module as the others do because the kinds of things it does are closely associated with the compiler itself. The designers of Modula-2 have therefore defined this module to make these things available to us.

```
(* Chapter 14 - Program 2 *)
MODULE WordAddr;

FROM SYSTEM      IMPORT ADR,WORD,ADDRESS;
FROM InOut       IMPORT WriteString,WriteCard,WriteLn;

VAR Index  : INTEGER;
    CardNo : CARDINAL;
    Peach  : ADDRESS;
    MonoVideo[0B000H:0H]  : ARRAY[1..4000] OF CHAR;
    ColorVideo[0B800H:0H] : ARRAY[1..4000] OF CHAR;

PROCEDURE PrintNumber(DatOut : WORD);
VAR Temp : CARDINAL;
BEGIN
   WriteString("The value is ");
   Temp := CARDINAL(DatOut);
   WriteCard(Temp,4);
   WriteLn;
END PrintNumber;

BEGIN
   Index := 17;
   CardNo := 38;
   Peach := ADR(Index);      (* Pointer to an INTEGER      *)
   Peach := ADR(CardNo);     (* Pointer to a CARDINAL      *)
   PrintNumber(Index);       (* Called with an INTEGER     *)
   PrintNumber(CardNo);      (* Called with a CARDINAL     *)

   Peach := 0B000H:1A2H;     (* Pointer to Segment:Offset *)

END WordAddr.
```

The new data type "WORD" is compatible with all data types that use a single word for storage, but it is somewhat limited in what you can do with it. It is most useful as the formal argument to a function which can be called with any data type that is contained in one word. In lines 27 and 28, the same procedure is called, once with an INTEGER, and once with a CARDINAL. Since the procedure is designed to handle either, it will print out both numbers by converting them first to CARDINAL using the type transformation in line 17, then calling the output procedures. Once again, the type transformation cannot be done in the procedure call so a temporary variable is used.

A NEW KIND OF POINTER

The variable "Peach" is assigned the type ADDRESS which is also imported from the pseudo module SYSTEM, and is therefore a pointer to any WORD type of variable. Peach can therefore point to an INTEGER or a CARDINAL as is done in lines 25 and 26. The procedure "ADR" returns the address of any WORD type of variable and it too must be IMPORTED from the pseudo module SYSTEM.

ABSOLUTE ADDRESSES

Notice the two strange looking variables in lines 10 and 11. The variable "MonoVideo" is an array of 4000 CHAR type variables but we have forced it to be located at a very specific location in memory, namely at segment=B000(hex) and offset=0000(hex). This is the method provided for you by Modula-2 by which you can force a variable to be at a specific memory location. In this case we have defined the variable to be stored in the locations in memory where the monochrome monitor display is stored so we can store data directly into the monochrome monitor display area.

The variable "ColorVideo" is the same except that the location referenced is that area where the output for a color monitor is stored. You can see that you can gain control over the actual hardware with this capability but it does require a lot of knowledge of the hardware and the operating system.

In the last line of the program the variable "Peach" is assigned the address of a specific location as an illustration only. This is only possible because "Peach" is a variable of type ADDRESS.

It should be clear to you that with these functions, it is possible to do a lot of data shuffling that could not otherwise be done. The next example program will illustrate their use further.

MORE ADDRESSING EXAMPLES

Load and display the program ADRSTUFF.MOD. This program uses the ADDRESS type and adds two new, rather simple functions, SIZE and TSIZE. Actually, these are not completly new since we used the TSIZE function in the chapter on pointers and dynamic allocation. These two functions will return the size in bytes of any variable or of any type. The program on your monitor has several types defined, then several variables, and finally initializes all of the elements of the array "Stuff" to some nonsense data. The really interesting things begin happening at line 25.

```
(* Chapter 14 - Program 3 *)
MODULE AdrStuff;

FROM InOut   IMPORT WriteInt, WriteLn;
FROM SYSTEM  IMPORT ADR, SIZE, TSIZE, ADDRESS;

TYPE IntArray = ARRAY[1..8] OF INTEGER;
     BigArray  = ARRAY[1..5] OF IntArray;

VAR  Stuff     : BigArray;
     NeatPoint : ADDRESS;
     IncreAmt  : CARDINAL;
     Index     : INTEGER;
     Count     : INTEGER;
     Amount    : INTEGER;

BEGIN
                              (* Load the array with nonsense data *)
   FOR Index := 1 TO 8 DO
      FOR Count := 1 TO 5 DO
         Stuff[Count][Index] := Index + 10 * Count;
      END;
   END;
                         (* Perform some simple pointer operations *)
   NeatPoint := ADR(Stuff[1][1]);
   Index := INTEGER(NeatPoint^);
   WriteInt(Index,6);
   IncreAmt := TSIZE(IntArray);
   NeatPoint := NeatPoint + IncreAmt;
   Index := INTEGER(NeatPoint^);
   WriteInt(Index,6);
   WriteLn;
                      (* Perform some pointer operations in a loop *)
   Count := INTEGER(TSIZE(BigArray)) DIV INTEGER(TSIZE(IntArray));
   FOR Index := 1 TO Count DO
      NeatPoint := ADR(Stuff[1][1]) +
                      CARDINAL((Index-1)*(INTEGER(TSIZE(IntArray))));
      Amount := INTEGER(NeatPoint^);
      WriteInt(Amount,6);
   END;

   IncreAmt := SIZE(Stuff);
   Count := INTEGER(SIZE(NeatPoint));

END AdrStuff.
```

The pointer "NeatPoint" is pointed at the first element of the array "Stuff", and its value is dereferenced into "Index". The type transformation is required because the result of the dereferencing is a CARDINAL. The data is written out. Next the size of the type "IntArray" is assigned to the variable "IncreAmt", which should be 8 words or 16 bytes. In line 29 we do some pointer arithmetic by adding the size of the type "IntArray" to the original value of the pointer which should cause it to point to the next row of the array. After dereferencing the pointer and getting its new value, we print it out to find that it did indeed move to the next row of the array.

Based on the above discussion, it should be apparent to you that you can move the pointer all around the array named "Stuff" and get whatever data you wish. The next section uses a loop to continue the process through all five rows. The only thing that may be confusing is line number 34 where we get the size of the "BigArray" type and divide it by the size of the "IntArray" type. The result should be 5, and you will see that it does five iterations through the loop. This is really a dumb way to get through this particular loop but it is only for purposes of illustration that it is done. Notice all of the type transformations to INTEGER in these statements, this is because the functions all return a CARDINAL type of data. Doing all of this in CARDINAL numbers would have made it much cleaner, but this was more illustrative for you.

TWO MORE LINES OF ILLUSTRATION

Lines 42 and 43 are given as an illustration for you of how to use the SIZE function. It simply returns the size in bytes, of any variable used as an argument.

PROGRAMMING EXERCISES

1. Modify the "AdrStuff" module to print out some of the type and variable sizes such as those calculated in lines 41 and 42.

2. Write a program with an array of 100 CARDINAL elements, fill the elements with nonsense data, and use a pointer to print out every 12th value starting at the highest element (number 100) and working downward.

Chapter 15 - Concurrency

PREREQUISITES FOR THIS MATERIAL

In order to understand this material you should have a good grasp of the material in Part I of this tutorial and at least a cursory knowledge of the material in chapter 14 concerning the pseudo module SYSTEM and especially the ADR and SIZE functions.

WHAT IS CONCURRENCY

True concurrency is when two processes are taking place at exactly the same time with neither affecting the other in a degrading way. This is possible in many areas of your life, such as when the Grandfather clock is running at the same time as the furnace and the television set. These are different processes all running at the same time. In the field of computers, true concurrency requires at least two separate processors that are running completely separate programs or different parts of the same program.

Usually, when computers are said to be running concurrent processes, reference is being made to a time sharing situation in which two or more programs are swapped in and out of the computer at a rapid rate, each getting a small slice of time, a millisecond being typical. Each process appears to be running constantly but is in fact only running a small part of the time, sort of stuttering. The swapping in and out is taking place so fast that, to the casual user, it appears that the entire computer is working on only his program.

WHAT IS MODULA-2 CONCURRENCY?

The process in Modula-2, which is called concurrency, is neither of the above, and it should probably not be called concurrency. It is a new way to call and return from procedures, and although it is possible to use this new method of procedure linkage to simulate something that looks like concurrent processes, it is not true concurrency. It is a part of the language, and a useful part, so we will cover it in this chapter.

OUR FIRST COROUTINE

Load and display the program named COROUT.MOD for our first look at this new technique. Without lots of explanation, there are lots of new items IMPORTED and a few variables defined, the most interesting being the three that are defined as PROCESS type. These will be the three processes we will use in this example. Next, there are two PROCEDURES which define what the coroutines will do. It must be noted that these PROCEDURES are not allowed to have any formal parameters in their headers. Finally we come to the main program which is where we will start to define how to use the coroutines.

```
(* Chapter 15 - Program 1 *)
MODULE Corout;

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

FROM SYSTEM IMPORT WORD, PROCESS, ADR, SIZE,
                   NEWPROCESS, TRANSFER;

VAR  main, Process1, Process2 : PROCESS;
     WorkSpace1, WorkSpace2   : ARRAY[1..300] OF WORD;
     Index : CARDINAL;

PROCEDURE MainProcess;
BEGIN
   FOR Index := 1 TO 5 DO
      WriteString('This is loop');
      WriteCard(Index,2);
      IF Index > 2 THEN
         TRANSFER(Process1,Process2);
         WriteString(' and back to main loop');
      END;
      WriteLn;
   END;      (* of FOR loop *)
   WriteString('End of the MainProcess loop');
   WriteLn;
   TRANSFER(Process1,main);
END MainProcess;

PROCEDURE SubProcess;
BEGIN
   LOOP
      WriteString(' in SubProcess');
      TRANSFER(Process2,Process1);
      WriteString(' back');
   END;
END SubProcess;

BEGIN   (* Main Module Body *)
   NEWPROCESS(MainProcess,ADR(WorkSpace1),SIZE(WorkSpace1),
              Process1);
   NEWPROCESS(SubProcess,ADR(WorkSpace2),SIZE(WorkSpace2),
              Process2);
   TRANSFER(main,Process1);
   WriteString('End of the program');
   WriteLn;
END Corout.
```

First, in the main program, we call "NEWPROCESS" which loads a new process without running it. We give it the name of the procedure where the process can be found, and give the system a workspace by assigning a previously defined array to it. This is done by giving the system the address and the size of the array. Finally, we give it a name by which we can call the process. It must be noted that the workspace must be sufficient for the new process to store all of its variables, so give the process a generous workspace. If it is too small, a runtime error will be generated. We then load a second process in preparation for running the program by calling NEWPROCESS again.

HOW DO WE GET IT STARTED?

When we began the main program, we actually loaded and started a process, the one we are presently executing. We have done this for every program in this tutorial so far but paid no attention to it as far as referring to it as a process. We have not given our running process a name yet, but we will when we leave it because we have defined another PROCESS type variable named "main". To start the next process we use the TRANSFER function with the name of the process we wish to terminate and the one we wish to start. This is illustrated in line 43 from which we jump to line 15 in the process named "Process1". In Process1 we begin a FOR loop where we write a string and a cardinal number then when "Index" exceeds 2, we do another TRANSFER, this time from Process1 to Process2. Thus we jump from line 19 in one procedure to line 31 in another where we begin an infinite loop. We print another string in line 32 and once again do a TRANSFER from line 33 to somewhere. The place where we go at this point is what makes the coroutines different from the standard procedure.

WHERE DO WE GO WHEN WE RETURN TO A COROUTINE?

Instead of jumping to the beginning of the procedure again, which would be line 15 in this example, we return to the statement following the one we left from. In this case we will return from line 33 to line 20 and complete the loop we started earlier. When Index is increased to 4, we will once again TRANSFER control from line 19 to "Process2", but instead of jumping to line 31 we will return where we left off there at line 34. That loop will complete, and we will once again return to line 20. When the FOR loop in "Process1" finishes, we execute lines 24 and 25 then the TRANSFER in line 26 will return us to the main module body where we will arrive at line 44 and complete the last two write statements and do a normal exit to the operating system.

Rather than a technical discussion of how to use coroutines, this example was defined step by step. If it was not clear, reread the entire description until you understand it. There is really nothing else there is to know about coroutines other than that knowledge gained by experience in using them, which is true of any new principle in computer programming or any other facet of life.

WHAT WAS ALL THAT ABOUT?

The thing that is really different about coroutines is the fact that they are both (or all three or more) loaded into the computers memory and they stay loaded for the life of the program. This is not true of normal procedures. It is transparent to you, but procedures are not all simply loaded into the computer and left there, they are dynamically allocated and started as they are called. That is why variables are not saved from one invocation of a procedure to the next. The variables within a process are loaded once, and stay resident for the life of the program.

In the example program on your monitor, all three processss including the main program are loaded and none is really the main program since any of the programs have the ability to call the others.

Load and display COROUT2.MOD for the second example with coroutines. This program is identical with the last except that two lines are removed from the first process and placed in a normal procedure which is called from line 22 to illustrate that some of the code can be placed outside of the coroutine process to make the resident part smaller. The implication here is that you could have a four way coprocess going on, each one of which had a very small resident portion, and each one of which called some huge regular procedures. In fact, there is no reason why two or more of the coprocesses could not call the same normal procedure.

```
(* Chapter 15 - Program 2 *)
MODULE Corout2;

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

FROM SYSTEM IMPORT WORD, PROCESS, ADR, SIZE,
                   NEWPROCESS, TRANSFER;

VAR  main, Process1, Process2 : PROCESS;
     WorkSpace1, WorkSpace2   : ARRAY[1..300] OF WORD;
     Index : CARDINAL;

PROCEDURE WriteStuff;
```

```
BEGIN
   WriteString('This is loop');
   WriteCard(Index,2);
END WriteStuff;

PROCEDURE MainProcess;
BEGIN
   FOR Index := 1 TO 5 DO
      WriteStuff;
      IF Index > 2 THEN
         TRANSFER(Process1,Process2);
         WriteString(' and back to main loop');
      END;
      WriteLn;
   END;
   WriteString('End of the MainProcess loop');
   WriteLn;
   TRANSFER(Process1,main);
END MainProcess;

PROCEDURE SubProcess;
BEGIN
   LOOP
      WriteString(' in SubProcess');
      TRANSFER(Process2,Process1);
      WriteString(' back');
   END;
END SubProcess;

BEGIN   (* Main Module Body *)
   NEWPROCESS(MainProcess,ADR(WorkSpace1),SIZE(WorkSpace1),
              Process1);
   NEWPROCESS(SubProcess,ADR(WorkSpace2),SIZE(WorkSpace2),
              Process2);
   TRANSFER(main,Process1);
   WriteString('End of the program');
   WriteLn;
END Corout2.
```

Study this program until you understand the implications, then compile and run it to see that it does exactly the same thing as the last program.

HOW ABOUT THREE PROCESSES?

Load and display COROUT3.MOD for an example with three concurrent processes. This program is identical to the first program with the addition of another process. In this program, process 1 calls process 2, which calls process 3, which once again calls process 1. Thus the same loop is traversed but with one additional stop along the way. It should be evident to you that there is no reason why this could not be extended to any number of processes each calling the next or in any looping scheme you could think up provided of course that it had some utilitarian purpose.

```
(* Chapter 15 - Program 3 *)
MODULE Corout3;

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

FROM SYSTEM IMPORT WORD, PROCESS, ADR, SIZE,
                   NEWPROCESS, TRANSFER;

VAR  main, Process1, Process2 ,Process3 : PROCESS;
     WorkSpace1, WorkSpace2, WorkSpace3 : ARRAY[1..300] OF WORD;
     Index : CARDINAL;

PROCEDURE MainProcess;
BEGIN
   FOR Index := 1 TO 5 DO
      WriteString('This is loop');
      WriteCard(Index,2);
      IF Index > 2 THEN
         TRANSFER(Process1,Process2);
         WriteString(' and back to main loop');
      END;
      WriteLn;
   END;
   WriteString('End of the MainProcess loop');
   WriteLn;
   TRANSFER(Process1,main);
END MainProcess;

PROCEDURE SubProcess;
BEGIN
   LOOP
      WriteString(' in SubProcess');
      TRANSFER(Process2,Process3);
      WriteString(' back');
   END;
END SubProcess;

PROCEDURE ThirdProcess;
BEGIN
   LOOP
      WriteString(' in ThirdProcess');
      TRANSFER(Process3,Process1);
      WriteString(' back');
   END;
END ThirdProcess;

BEGIN   (* Main Module Body *)
   NEWPROCESS(MainProcess,ADR(WorkSpace1),SIZE(WorkSpace1),
              Process1);
   NEWPROCESS(SubProcess,ADR(WorkSpace2),SIZE(WorkSpace2),
              Process2);
   NEWPROCESS(ThirdProcess, ADR(WorkSpace3),SIZE(WorkSpace3),
              Process3);
   TRANSFER(main,Process1);
   WriteString('End of the program');
   WriteLn;
END Corout3.
```

WHAT IS THE BIG DIFFERENCE?

Actually, the big difference between real concurrent processing and what this is doing is in the division of time and in who, or what, is doing the division. In real concurrent processing, the computer hardware is controlling the operation of the processing and allocating time slots to the various processes. In the pseudo concurrent processing we are doing, it is the processes themselves that are doing the time allocation leading to the possibility that if one of the processes went bad, it could tie up all of the resources of the system and no other process could do anything. You could consider it a challenge to write a successful system that really did share time and resources well.

The important thing to consider about this technique is that it is not a major breakthrough in a programming language, but one additional tool available in Modula-2 that is not available in the other popular languages such as Pascal, C, Fortran, or BASIC.

ONE MORE INFINITE EXAMPLE OF CONCURRENCY

Load and display the program named INFINITE.MOD for another example of a program with concurrency. In this program, three processes are created and control is transferred to the first one after which they call each other in a loop with no provision for ever returning to the main program. The computer will continually loop around the three processes checking the printer, the keyboard, and the system clock to see if they need servicing. It must be pointed out that it would be a simple matter to include all three checks in a single loop in the main program and do away with all of this extra baggage. This method had one advantage over the simple loop and that is the fact that each coprocess can have its own variables which cannot be affected by the operation of the other processes and yet are all memory resident at all times.

```
(* Chapter 15 - Program 4 *)
MODULE Infinite;                    (* Infinite Coroutine loop *)

FROM InOut IMPORT WriteCard, WriteString, WriteLn;

FROM SYSTEM IMPORT WORD, PROCESS, ADR, SIZE,
                   NEWPROCESS, TRANSFER;

VAR  main, Process1, Process2 ,Process3 : PROCESS;
     WorkSpace1, WorkSpace2, WorkSpace3 : ARRAY[1..300] OF WORD;
     Index : CARDINAL;

PROCEDURE MainProcess;
BEGIN
   LOOP
      WriteString('Main Process');
      (* This may check the printer and send another character  *)
      (* for printing if it is ready.                           *)
      TRANSFER(Process1,Process2);
      WriteLn;
   END;
END MainProcess;

PROCEDURE SubProcess;
BEGIN
   LOOP
      WriteString(' SubProcess');
      (* This may check to see if there are any additional      *)
      (* characters to be read from the keyboard.               *)
      TRANSFER(Process2,Process3);
   END;
END SubProcess;

PROCEDURE ThirdProcess;
BEGIN
   LOOP
      WriteString(' ThirdProcess');
      (* This may check to see if there was another update in    *)
      (* the clock requiring service by the system.              *)
      TRANSFER(Process3,Process1);
   END;
END ThirdProcess;

BEGIN   (* Main Module Body *)
   NEWPROCESS(MainProcess,ADR(WorkSpace1),SIZE(WorkSpace1),
              Process1);
   NEWPROCESS(SubProcess,ADR(WorkSpace2),SIZE(WorkSpace2),
              Process2);
   NEWPROCESS(ThirdProcess, ADR(WorkSpace3),SIZE(WorkSpace3),
              Process3);
   TRANSFER(main,Process1);
   (* Note that we never return here, we stay in the status loop *)
END Infinite.
```

This program can be compiled and run but it will execute forever since it has no way to stop it. You can stop it because it is writing to the monitor and therefore will be checking the control-break combination. Simply hit control-break and the program will be terminated by the operating system. It should be mentioned that if this technique were used in a real situation, it would probably not be writing to the monitor continually.

IS THIS REALLY USEFUL?

In a situation where you needed to service interrupts in some prescribed and rapid fashion, a technique involving Modula-2 concurrency could prove to be very useful. There are other procedures available in Modula-2 to aid in writing a pseudo-concurrent program but they are extrememly advanced and would require an initimate knowledge of your particular systems hardware, especially the interrupt system.

Since using these techniques are extremely advanced programming techniques, they will not be covered here. They are beyond the scope of this tutorial. It would be to your advantage to study them and know that they exist, then someday you may find that they fill the bill and greatly simplify some particular programming problem.

Chapter 16 - Complete example programs

The intent of this chapter is to give several example programs that use nearly every capability of Modula-2 as illustrations of large usable programs. The programs are usable utilities, but primarily they are intended to illustrate the method of building up a medium sized program from the various constructs studied in the earlier chapters.

BAKLIST.MOD

```
MODULE BakList;

(* This program is used to generate a list of all files in all     *)
(* subdirectories except for the DOS files and the list file which *)
(* this program generates.  The file FULLDISK.LST is created and   *)
(* filled in the root directory of the default drive containing    *)
(* the entire tree from the default subdirectory to all end points.*)
(* If this program is called from the root directory, the tree for *)
(* the entire default directory will be listed.  The resulting file*)
(* can then be edited with any text editor to allow copying of all *)
(* files or only selected files.                                   *)
(*                                                                 *)
(*             Copywrite (c) 1987 - Coronado Enterprises           *)


FROM InOut         IMPORT WriteString,WriteCard,Write,WriteLn;
FROM RealInOut     IMPORT WriteReal;
FROM Real2Fil      IMPORT WriteStringFile,WriteLnFile;
FROM DiskDirectory IMPORT CurrentDrive,CurrentDirectory;
FROM Strings       IMPORT Concat,Copy,Length,Insert,CompareStr,
                          Delete;
FROM FileSystem    IMPORT Lookup,Close,File,Response;
FROM SYSTEM        IMPORT AX,BX,CX,DX,SI,DI,ES,DS,CS,SS,SIZE,TSIZE,
                          ADR,ADDRESS,GETREG,SETREG,SWI;
FROM Storage       IMPORT ALLOCATE,DEALLOCATE;
FROM DirHelps      IMPORT ReadFileStats,FileDataPointer,FileData;

VAR Drive        : CHAR;
    StartingPath : ARRAY[0..64] OF CHAR;
    FileList     : File;
    DiskTransAdr : ARRAY[1..43] OF CHAR;




(* This procedure selects the current drive and directory and opens*)
(* the file named FULLDISK.LST to be used for recording all sub-   *)
(* directories and filenames.                                      *)
```

```
PROCEDURE Initialize() : BOOLEAN;
VAR StorageFile : ARRAY[0..20] OF CHAR;
BEGIN
   CurrentDrive(Drive);                (* This generates the path *)
   CurrentDirectory(Drive,StartingPath);(* used for the start of   *)
   Insert(Drive,StartingPath,0);       (* the search.  It uses    *)
   Insert(':',StartingPath,1);         (* the current drive and   *)
   IF StartingPath[2] <> 000C THEN     (* path.                   *)
      Insert('\',StartingPath,2);
   END;

   StorageFile := ":\FULLDISK.LST";    (* This opens the file used*)
   Insert(Drive,StorageFile,0);        (* to store the file-list. *)
   Lookup(FileList,StorageFile,TRUE);  (* It is forced into the   *)
   IF FileList.res = done THEN         (* root of the current     *)
      RETURN(TRUE);                    (* directory.              *)
   ELSE
      RETURN(FALSE);
   END;
END Initialize;




(* This procedure is used to store all of the data found into a    *)
(* B-tree structure to be used in sorting the files and subdirec-  *)
(* tories alphabetically.                                          *)

PROCEDURE StoreData(NewData         : FileDataPointer;
                    VAR Files       : FileDataPointer;
                    VAR Directories : FileDataPointer);
     PROCEDURE AddToTree(VAR RootOfTree : FileDataPointer;
                         VAR NewNode    : FileDataPointer);
     VAR Result : INTEGER;
     BEGIN
        Result := CompareStr(RootOfTree^.Name,NewNode^.Name);
        IF Result = 1 THEN
           IF RootOfTree^.Left = NIL THEN
              RootOfTree^.Left := NewNode;
           ELSE
              AddToTree(RootOfTree^.Left,NewNode);
           END;
        ELSE
           IF RootOfTree^.Right = NIL THEN
              RootOfTree^.Right := NewNode;
           ELSE
              AddToTree(RootOfTree^.Right,NewNode);
           END;
        END;
     END AddToTree;

     PROCEDURE GoodFile(FileName : ARRAY OF CHAR) : BOOLEAN;
     BEGIN
        IF    CompareStr(FileName,"COMMAND.COM ") = 0 THEN
           RETURN(FALSE);
        ELSIF CompareStr(FileName,"IBMBIO.COM  ") = 0 THEN
```

```
            RETURN(FALSE);
         ELSIF CompareStr(FileName,"IBMDOS.COM  ") = 0 THEN
            RETURN(FALSE);
         ELSIF CompareStr(FileName,"FULLDISK.LST") = 0 THEN
            RETURN(FALSE);
         ELSE
            RETURN(TRUE);
         END;
      END GoodFile;

VAR  Index   : CARDINAL;
BEGIN
   IF NewData^.Attr = 010H THEN         (* Attr = 10 for directory *)
      IF NewData^.Name[0] <> "." THEN
         IF Directories = NIL THEN
            Directories := NewData;
         ELSE
            AddToTree(Directories,NewData);
         END;
      END;
   ELSE                                    (* Otherwise a filename *)
      IF GoodFile(NewData^.Name) THEN
         IF Files = NIL THEN
            Files := NewData;
         ELSE
            AddToTree(Files,NewData);
         END;
      ELSE
         WriteString("File ignored here --->");
         WriteString(NewData^.Name);
         WriteLn;
      END;
   END;
END StoreData;




(* This procedure reads the file statistics from DOS and stores the*)
(* data in a record for further use.                               *)

PROCEDURE ReadFileStatistics(VAR Files : FileDataPointer;
                             VAR Directories : FileDataPointer;
                             PathToFiles : ARRAY OF CHAR);
TYPE MaskStore = ARRAY[0..70] OF CHAR;
VAR SmallMask   : MaskStore;  (* Used for Directory output to file *)
    MaskAndFile : MaskStore;          (* Used for file search name *)
    MaskAddr    : ADDRESS;
    Error       : BOOLEAN;
    Index       : CARDINAL;
    NewData     : FileDataPointer;
    FirstFile   : BOOLEAN;
BEGIN
   WriteString(PathToFiles);
   WriteLn;
   Copy(PathToFiles,0,SIZE(PathToFiles),SmallMask);
   Delete(SmallMask,0,2);
   WriteStringFile(FileList,SmallMask);
   WriteLnFile(FileList);
   ALLOCATE(NewData,TSIZE(FileData));
   FirstFile := TRUE;
   Concat(PathToFiles,"/*.*",MaskAndFile);
   ReadFileStats(MaskAndFile,FirstFile,NewData,Error);
   IF NOT Error THEN
      StoreData(NewData,Files,Directories);
   END;

   REPEAT
      ALLOCATE(NewData,TSIZE(FileData));
      FirstFile := FALSE;
      ReadFileStats(MaskAndFile,FirstFile,NewData,Error);
      IF NOT Error THEN
         StoreData(NewData,Files,Directories);
      END;
   UNTIL Error;

END ReadFileStatistics;




(* This procedure lists all of the filenames alphabetically by     *)
(* recursively tracing the B-tree described above.                 *)

PROCEDURE ListAllFiles(Files : FileDataPointer);
VAR TempString : ARRAY[0..5] OF CHAR;
BEGIN
   IF Files <> NIL THEN
      IF Files^.Left <> NIL THEN
         ListAllFiles(Files^.Left);
      END;
         TempString := " ";
         WriteStringFile(FileList,TempString);
         WriteStringFile(FileList,Files^.Name);
         WriteLnFile(FileList);
      IF Files^.Right <> NIL THEN
         ListAllFiles(Files^.Right);
      END;
   END;
END ListAllFiles;




(* This procedure searches all Subdirectory names found in a       *)
(* search of a subdirectory for additional files and subdirector-  *)
(* ies.  The search is recursive.                                  *)

PROCEDURE DoAllSubdirectories(StartPath : ARRAY OF CHAR;
                              Directories : FileDataPointer);
VAR NewPath : ARRAY[0..64] OF CHAR;
    Index   : CARDINAL;
```

```
BEGIN
   IF Directories <> NIL THEN
      IF Directories^.Left <> NIL THEN
         DoAllSubdirectories(StartPath,Directories^.Left);
      END;
      IF Directories^.Name[0] <> '.' THEN
         Copy(StartPath,0,64,NewPath);
         Insert('\',NewPath,Length(NewPath));
         Concat(NewPath,Directories^.Name,NewPath);
         FOR Index := (SIZE(NewPath)-1) TO 1 BY -1 DO
            IF NewPath[Index] = ' ' THEN
               NewPath[Index] := 000C;
            END;
         END;
         GetAllFilesAndDirectories(NewPath);
      END;
      IF Directories^.Right <> NIL THEN
         DoAllSubdirectories(StartPath,Directories^.Right);
      END;
   END;
END DoAllSubdirectories;



(* This procedure deletes a tree after it has completed its task   *)
(* and is no longer of any use.                                    *)

PROCEDURE DeleteTree(Point : FileDataPointer);
BEGIN
   IF Point <> NIL THEN
      DeleteTree(Point^.Left);
      DeleteTree(Point^.Right);
      DEALLOCATE(Point,TSIZE(FileData));
   END;
END DeleteTree;



(* This procedure searches a subdirectory for all files names and  *)
(* additional subdirectories.                                      *)

PROCEDURE GetAllFilesAndDirectories(ThisPath : ARRAY OF CHAR);
VAR DirExists   : BOOLEAN;           (* Temporary - use logic later*)
    Files       : FileDataPointer;   (* Point to root of File tree *)
    Directories : FileDataPointer;   (* Point to root of Dir tree  *)
BEGIN
   Files := NIL;
   Directories := NIL;
   ReadFileStatistics(Files,Directories,ThisPath);
   ListAllFiles(Files);           (* List to a file for later use. *)
   DoAllSubdirectories(ThisPath,Directories);
   DeleteTree(Files);
   DeleteTree(Directories);
END GetAllFilesAndDirectories;
```

```
BEGIN   (* Main program - BakList, Backup list *)
   IF Initialize() THEN
      GetAllFilesAndDirectories(StartingPath);
      Close(FileList);
   ELSE
      WriteString("File named FULLDISK.LST cannot be opened");
      WriteLn;
   END;
END BakList.
```

This program generates a list of all files along with their subdirectories. Some files are excluded from the list, including all three files that comprise the DOS system and the file generated here, FULLDISK.LST. This is an ASCII file that can be edited with any text editor to eliminate any files that you do not wish to back up. It should be noted that the file, FULLDISK.LST, is created and filled in the root directory of the default drive.

Select the desired subdirectory that you wish to back up, and the files, subdirectories, and all of their respective contents will be listed in FULLDISK.LST. The resulting list is then used by BAKCOPY to actually copy the files to a floppy disk.

BAKCOPY.MOD

```
MODULE BakCopy;

(* This program is used to actually copy the files from the fixed  *)
(* disk to the floppy disks.  It uses the file FULLDISK.LST as the *)
(* basis for its copying.  That file is generated using the sister *)
(* program BAKLIST, and after generation, it can be modified with  *)
(* any text editor to allow elimination of any files or directories*)
(* that you do not wish to back up.                                *)
(*                                                                 *)
(*           Copywrite (c) 1987 - Coronado Enterprises             *)

(* Note that this is a preliminary version of this example program *)
(* and as such, it is not completely refined as it would need to   *)
(* be for a full production system.  Since it was never intended   *)
(* to compete with the full production backup systems available,   *)
(* but was meant only to illustrate the method of building up a    *)
(* significant sized program, it is considered to have attained    *)
(* the original goal.  It can be used as a backup system if you    *)
(* don't mind the following problems.                              *)
(*                                                                 *)
(* 1. The date and time of the files on the copy are the date and  *)
(*    time that the copies are made, not the date and time of the  *)
(*    original files.  The date and time of the original can be    *)
(*    read and copied to the copy using interrupt 21 - function    *)
(*    call 57H if you can figure out how to get the file handle.   *)
(*                                                                 *)
(* 2. This system does not copy hidden files.                      *)
(*                                                                 *)
(* 3. This system does not copy files that are too big to fit on   *)
(*    one floppy disk.                                             *)
(*                                                                 *)
(* 4. The filesize and the room remaining on the disk are handled  *)
(*    using floating point numbers instead of CARDINAL which would *)
(*    be a much needed improvement.  The floating point numbers on *)
(*    this system use enough significant digits to allow this,     *)
(*    but changing to CARDINAL would be an improvement.  Keep in   *)
(*    mind if you attempt this, that the upper limit on a CARDINAL *)
(*    is 65535 so it would require the use of two CARDINALS for    *)
(*    filesize and two for room on disk.                           *)

FROM InOut         IMPORT WriteString, WriteCard, WriteLn,
                          Write, Read;
FROM RealInOut     IMPORT WriteReal;
FROM FileSystem    IMPORT Lookup, Close, File, Response, ReadByte;
FROM Strings       IMPORT Copy, Insert, Delete;
FROM DiskDirectory IMPORT CurrentDrive;
FROM SYSTEM        IMPORT ADR;
FROM DirHelps      IMPORT GetDiskStatistics, ChangeToDirectory,
                          CopyFile, FileData, FileDataPointer,
                          ReadFileStats;

TYPE CharArray = ARRAY[0..100] OF CHAR;

VAR  InputFile         : File;
     SourceDrive       : CHAR;
     SourceFile        : CharArray;
     TargetDrive       : CHAR;
     TargetFile        : CharArray;
     InputLine         : CharArray;
     WorkingDirectory  : CharArray;
     Char              : CHAR;
     Index             : CARDINAL;
     SectorsPerCluster : CARDINAL;
     FreeClusters      : CARDINAL;
     BytesPerSector    : CARDINAL;
     TotalClusters     : CARDINAL;
     ErrorRet          : BOOLEAN;
     ErrorCode         : CARDINAL;
     FileSize          : REAL;
     RoomOnDisk        : REAL;
     RoomOnNewDisk     : REAL;
     DataForFile       : FileData;
     PointToData       : FileDataPointer;
     DiskNumber        : CARDINAL;
     EndOfCopy         : BOOLEAN;


(* This procedure is used to read in one full line from the input  *)
(* file.                                                            *)
```

```
PROCEDURE ReadALine;
BEGIN
   Index := 0;
   REPEAT        (* Read one line of input data *)
      ReadByte(InputFile,Char);
      IF Char <> 15C THEN
         InputLine[Index] := Char;
         INC(Index);
      END;
   UNTIL (Index = 100) OR (Char = 12C) OR InputFile.eof;
   InputLine[Index - 1] := 000C;
END ReadALine;



(* This procedure calls the actual copying routine after it checks *)
(* to see if there is enough room on the target floppy.  If there  *)
(* is not, it requests a blank floppy to be loaded.                *)

PROCEDURE CopyTheFile;
BEGIN
   Delete(InputLine,0,1);                  (* Remove leading blank *)
   SourceFile := InputLine;
   Insert(SourceDrive,SourceFile,0);
   Insert(':',SourceFile,1);
   TargetFile := InputLine;
   Insert(TargetDrive,TargetFile,0);
   Insert(':',TargetFile,1);
                           (* See if the file will fit on the disk *)
   PointToData := ADR(DataForFile);
   ReadFileStats(SourceFile,TRUE,PointToData,ErrorRet);
   FileSize := PointToData^.Size;
   GetDiskStatistics(TargetDrive,SectorsPerCluster,FreeClusters,
                     BytesPerSector,TotalClusters);
   RoomOnDisk := FLOAT(SectorsPerCluster) *
                 FLOAT(FreeClusters) *
                 FLOAT(BytesPerSector);
   IF RoomOnDisk >= FileSize THEN
      CopyFile(SourceFile,TargetFile,FileSize,ErrorCode);
   ELSIF RoomOnNewDisk >= FileSize THEN
      WriteString("Install a new disk, hit return to continue");
      WriteString(", or hit Q to stop backup");
      WriteLn;
      Read(Char);
      IF Char = 'Q' THEN
         EndOfCopy := TRUE;
      ELSE
         INC(DiskNumber);
         WriteString("Beginning disk number ");
         WriteCard(DiskNumber,3);
         WriteLn;
         ChangeToDirectory(WorkingDirectory,TRUE,ErrorRet);
         GetDiskStatistics(TargetDrive,SectorsPerCluster,FreeClusters,
                           BytesPerSector,TotalClusters);
         RoomOnDisk := FLOAT(SectorsPerCluster) *
```

```
                       FLOAT(FreeClusters) *
                       FLOAT(BytesPerSector);

         IF RoomOnDisk >= FileSize THEN
            CopyFile(SourceFile,TargetFile,FileSize,ErrorCode);
         ELSE
            WriteString("File too big for this system");
         END;
      END;
   ELSE
      WriteString("File too big for this system");
   END;
END CopyTheFile;



(* This procedure makes the calls to change the directories of both*)
(* the source and target directories.                               *)

PROCEDURE ChangeBothDirectories;
BEGIN
   Insert(SourceDrive,InputLine,0);
   Insert(':',InputLine,1);
   ChangeToDirectory(InputLine,FALSE,ErrorRet);
   IF ErrorRet THEN
      WriteString(InputLine);
      WriteString("  Cannot change to source directory");
      WriteLn;
   END;
   InputLine[0] := TargetDrive;
   WorkingDirectory := InputLine;
   ChangeToDirectory(InputLine,TRUE,ErrorRet);
   IF ErrorRet THEN
      WriteString(InputLine);
      WriteString("  Cannot change to target directory");
      WriteLn;
   END;
END ChangeBothDirectories;



BEGIN (* Main program *)
   DiskNumber := 1;
   EndOfCopy := FALSE;
   WriteString("Enter the target drive, one letter ---> ");
   Read(TargetDrive);
   TargetDrive := CAP(TargetDrive);
   Write(TargetDrive);
   WriteLn;
   WriteString("Beginning disk number   1");
   WriteLn;
   GetDiskStatistics(TargetDrive, SectorsPerCluster, FreeClusters,
                     BytesPerSector, TotalClusters);
   IF BytesPerSector > 0 THEN                 (* Valid drive found *)
      RoomOnNewDisk := FLOAT(SectorsPerCluster) *
                       FLOAT(TotalClusters) *
                       FLOAT(BytesPerSector);
      Copy("C:\FULLDISK.LST",0,100,SourceFile);
      CurrentDrive(SourceDrive);       (* Get current drive letter *)
      SourceFile[0] := SourceDrive;  (* Open FULLDISK.LST for read *)
      Lookup(InputFile,SourceFile,FALSE);
      IF InputFile.res = done THEN
         LOOP
            ReadALine;
            IF InputFile.eof THEN
               EXIT;
            ELSE
               IF InputLine[0] = ' ' THEN              (* Filename *)
                  CopyTheFile;
                  IF EndOfCopy THEN EXIT END;
               ELSIF InputLine[0] = 000C THEN
                              (* Empty line, not a directory entry *)
               ELSE                                   (* Directory *)
                  ChangeBothDirectories;
                  WriteString(" Directory ---> ");
                  WriteString(InputLine);
                  WriteLn;
               END
            END;
         END; (* LOOP *)
         Close(InputFile);
      ELSE
         WriteString("FULLDISK.LST not available for reading.");
         WriteLn;
         WriteString("Program terminated");
         WriteLn;
      END;
      WriteString("End of Backup copy program");
      WriteLn;
   END; (* Drive test *)
END BakCopy.
```

This program uses FULLDISK.LST to actually copy the files from the source disk to the target and requests a disk change whenever the floppy disk fills up. It will not copy a file larger than that which will fit on one disk, but will give a message of which files are not copied.

BAKRSTR.MOD

```
MODULE BakRstr;

(* This program is used to restore the files from the floppy disks *)
(* to the hard disk.  This program is loaded into the root direc-  *)
(* tory of the hard disk and executed from there.  The files are   *)
(* read from the floppy and copied into the same directory of the  *)
(* hard disk as they are in on the floppy, the directories being   *)
(* created as needed on the hard disk.  To restore additional      *)
(* disks, simply rerun this program once for each disk.             *)
(*                                                                  *)
(*               Copywrite 1987 - Coronado Enterprises              *)

FROM InOut         IMPORT WriteString,Read,Write,WriteLn;
FROM DiskDirectory IMPORT CurrentDrive,CurrentDirectory;
FROM Strings       IMPORT Concat,Copy,Insert,Delete,Length,
                          CompareStr;
FROM Storage       IMPORT ALLOCATE,DEALLOCATE;
FROM FileSystem    IMPORT File;
FROM SYSTEM        IMPORT ADDRESS,TSIZE,SIZE;
FROM DirHelps      IMPORT ReadFileStats,FileDataPointer,FileData,
                          CopyFile,ChangeToDirectory;

VAR SourceDrive  : CHAR;
    TargetDrive  : CHAR;
    StartingPath : ARRAY[0..64] OF CHAR;
    FileList     : File;
    DiskTransAdr : ARRAY[1..43] OF CHAR;




PROCEDURE Initialize();
VAR StorageFile : ARRAY[0..20] OF CHAR;
BEGIN
   WriteString("Enter the source drive, one letter ---> ");
   Read(SourceDrive);
   SourceDrive := CAP(SourceDrive);
   Write(SourceDrive);
   WriteLn;
   CurrentDrive(TargetDrive);
   TargetDrive := 'A';
   Copy("A:",0,64,StartingPath);
   StartingPath[0] := SourceDrive;
END Initialize;



(* This procedure is used to copy the files from the floppy to the *)
(* hard disk while making a list of subdirectories found in order  *)
(* to copy each of them also.                                       *)

PROCEDURE StoreData(NewData         : FileDataPointer;
                    VAR Directories : FileDataPointer);
     PROCEDURE AddToTree(VAR RootOfTree : FileDataPointer;
                         VAR NewNode    : FileDataPointer);
     VAR Result : INTEGER;
     BEGIN
        Result := CompareStr(RootOfTree^.Name,NewNode^.Name);
        IF Result = 1 THEN
           IF RootOfTree^.Left = NIL THEN
              RootOfTree^.Left := NewNode;
           ELSE
              AddToTree(RootOfTree^.Left,NewNode);
           END;
```

```
         ELSE
            IF RootOfTree^.Right = NIL THEN
               RootOfTree^.Right := NewNode;
            ELSE
               AddToTree(RootOfTree^.Right,NewNode);
            END;
         END;
      END AddToTree;

VAR  Error      : CARDINAL;
     SourceFile : ARRAY[0..20] OF CHAR;
     DestFile   : ARRAY[0..20] OF CHAR;
BEGIN
   IF NewData^.Attr = 010H THEN         (* Attr = 10 for directory *)
      IF NewData^.Name[0] <> "." THEN
         IF Directories = NIL THEN
            Directories := NewData;
         ELSE
            AddToTree(Directories,NewData);
         END;
      END;
   ELSE                                    (* Otherwise a filename *)
      WriteString("Copyfile ---> ");
      WriteString(NewData^.Name);
      WriteLn;
      Copy(NewData^.Name,0,20,SourceFile);
      Insert(SourceDrive,SourceFile,0);
      Insert(':',SourceFile,1);
      Copy(NewData^.Name,0,20,DestFile);
      Insert(TargetDrive,DestFile,0);
      Insert(':',DestFile,1);
      CopyFile(SourceFile,DestFile,NewData^.Size,Error);
      IF Error <> 0 THEN
         WriteString("Error copying file ---> ");
         WriteString(SourceFile);
         WriteLn;
      END;
   END;
END StoreData;



(* This procedure reads the file statistics from DOS and stores    *)
(* the data in a record for further use.                           *)

PROCEDURE ReadFileStatistics(VAR Directories : FileDataPointer;
                             PathToFiles : ARRAY OF CHAR);
TYPE MaskStore = ARRAY[0..70] OF CHAR;
VAR MaskAndFile  : MaskStore;         (* Used for file search name *)
    ModifiedPath : MaskStore;
    MaskAddr     : ADDRESS;
    Error        : BOOLEAN;
    Index        : CARDINAL;
    NewData      : FileDataPointer;
    FirstFile    : BOOLEAN;
BEGIN
```

```
   WriteString("Changepath ---> ");
   WriteString(PathToFiles);
   WriteLn;
   Copy(PathToFiles,0,64,ModifiedPath);
   IF ModifiedPath[2] = 000C THEN
      ModifiedPath[2] := '\';
      ModifiedPath[3] := 000C;
   END;
   ModifiedPath[0] := TargetDrive;
   ChangeToDirectory(ModifiedPath,TRUE,Error);
   IF Error THEN
      WriteString("Cannot change target directory ---> ");
      WriteString(ModifiedPath);
      WriteLn;
   END;
   ModifiedPath[0] := SourceDrive;
   ChangeToDirectory(ModifiedPath,FALSE,Error);
   IF Error THEN
      WriteString("Cannot change source directory ---> ");
      WriteString(ModifiedPath);
      WriteLn;
   END;
   ALLOCATE(NewData,TSIZE(FileData));
   FirstFile := TRUE;
   Concat(PathToFiles,"/*.*",MaskAndFile);
   ReadFileStats(MaskAndFile,FirstFile,NewData,Error);
   IF NOT Error THEN
      StoreData(NewData,Directories);
   END;

   REPEAT
      ALLOCATE(NewData,TSIZE(FileData));
      FirstFile := FALSE;
      ReadFileStats(MaskAndFile,FirstFile,NewData,Error);
      IF NOT Error THEN
         StoreData(NewData,Directories);
      END;
   UNTIL Error;

END ReadFileStatistics;



(* This procedure searches all subdirectory names found in a       *)
(* search of a subdirectory for additional files and subdirector-  *)
(* ies.  The search is recursive.                                  *)

PROCEDURE DoAllSubdirectories(StartPath : ARRAY OF CHAR;
                              Directories : FileDataPointer);
VAR NewPath : ARRAY[0..64] OF CHAR;
    Index   : CARDINAL;
BEGIN
   IF Directories <> NIL THEN
      IF Directories^.Left <> NIL THEN
         DoAllSubdirectories(StartPath,Directories^.Left);
      END;
      IF Directories^.Name[0] <> '.' THEN
         Copy(StartPath,0,64,NewPath);
         Insert('\',NewPath,Length(NewPath));
         Concat(NewPath,Directories^.Name,NewPath);
         FOR Index := (SIZE(NewPath)-1) TO 1 BY -1 DO
            IF NewPath[Index] = ' ' THEN
               NewPath[Index] := 000C;
            END;
         END;
         GetAllFilesAndDirectories(NewPath);
      END;
      IF Directories^.Right <> NIL THEN
         DoAllSubdirectories(StartPath,Directories^.Right);
      END;
   END;
END DoAllSubdirectories;



(* This procedure deletes a tree after it is no longer needed.     *)

PROCEDURE DeleteTree(Point : FileDataPointer);
BEGIN
   IF Point <> NIL THEN
      DeleteTree(Point^.Left);
      DeleteTree(Point^.Right);
      DEALLOCATE(Point,TSIZE(FileData));
   END;
END DeleteTree;



PROCEDURE GetAllFilesAndDirectories(ThisPath : ARRAY OF CHAR);
VAR DirExists   : BOOLEAN;           (* Temporary - use logic later*)
    Directories : FileDataPointer;   (* Point to root of Dir tree  *)
BEGIN
   Directories := NIL;
   ReadFileStatistics(Directories,ThisPath);
   DoAllSubdirectories(ThisPath,Directories);
   DeleteTree(Directories);
END GetAllFilesAndDirectories;


BEGIN   (* Main program - BakRstr, Backup restore *)
  Initialize;
  GetAllFilesAndDirectories(StartingPath);
END BakRstr.
```

This program will read the files from floppy back to the fixed disk to restore it. It simply copies from whatever directory they are in to the corresponding directory on the fixed disk, creating the directory if necessary.

DIRHELPS.DEF DIRHELPS.MOD

```
DEFINITION MODULE DirHelps;

(*           Copyright (c) 1987 - Coronado Enterprises           *)

EXPORT QUALIFIED ReadFileStats,
                 GetDiskStatistics,
                 ChangeToDirectory,
                 CopyFile,
                 FileDataPointer,
                 FileData;


TYPE FileDataPointer = POINTER TO FileData;
     FileData = RECORD
        Name  : ARRAY[0..13] OF CHAR;
        Attr  : CARDINAL;
        Time  : CARDINAL;
        Date  : CARDINAL;
        Size  : REAL;
        Left  : FileDataPointer;
        Right : FileDataPointer;
     END;



(*****************************************************************)
PROCEDURE ReadFileStats(FileName      : ARRAY OF CHAR;
                        FirstFile     : BOOLEAN;
                        VAR FilePt    : FileDataPointer;
                        VAR FileError : BOOLEAN);

(* This procedure is used to read the DOS data concerning a file.  *)
(* It returns a pointer to the FileData structure containing all   *)
(* of the file data.  FirstFile is set to TRUE for the first file  *)
(* and to FALSE for the remaining files in the list.  FileError    *)
(* returns TRUE if the read was successful and FALSE if it was not *)
(* with a FALSE also indicating the end of the files in this dir-  *)
(* ectory.                                                         *)



(*****************************************************************)
PROCEDURE GetDiskStatistics(Drive                 : CHAR;
                            VAR SectorsPerCluster : CARDINAL;
                            VAR FreeClusters      : CARDINAL;
                            VAR BytesPerSector    : CARDINAL;
                            VAR TotalClusters     : CARDINAL);

(* This procedure gets the disk statistics on the selected drive.  *)



(*****************************************************************)
```

```
PROCEDURE ChangeToDirectory(Directory : ARRAY OF CHAR;
                            CreateIt : BOOLEAN;
                            VAR ErrorReturn : BOOLEAN);

(* C:\DIR1\DIR2\DIR3\<000>   Example of usage                    *)
(* This procedure is used to change to a directory on the selected *)
(* drive included in the CHAR array. The directory is a complete   *)
(* path and if the CreatIt flag is TRUE, the directory will be     *)
(* created, otherwise an error return will be generated as follows.*)
(* ErrorReturn = 0  Directory created as desired.                  *)
(* ErrorReturn = 1  Directory doesn't exist and CreateIt = FALSE.  *)
(* ErrorReturn = 2  Not enough disk room to create Directory.      *)



(*****************************************************************)
PROCEDURE CopyFile(SourceFile       : ARRAY OF CHAR;
                   DestinationFile  : ARRAY OF CHAR;
                   FileSize         : REAL;
                   VAR ResultOfCopy : CARDINAL);

(* C:FILENAME.EXT<000>   Example of usage                        *)
(* This procedure copies a file from SourceDrive:SourceFile to     *)
(* DestinationDrive:DestinationFile and returns a ResultOfCopy     *)
(* indicator to signal the result of the copy.  It assumes that    *)
(* the proper subdirectory has been selected prior to a call to    *)
(* this routine.  If a file cannot be opened, it is not copied.    *)

(* ResultOfCopy = 0  Good copy made.                               *)
(* ResultOfCopy = 1  Cannot open source file.                      *)
(* ResultOfCopy = 2  Cannot open destination file.                 *)
(* ResultOfCopy = 3  Not enough room on the disk.                  *)


END DirHelps.




IMPLEMENTATION MODULE DirHelps;

(*           Copyright (c) 1987 - Coronado Enterprises            *)

FROM InOut      IMPORT WriteString,Write,WriteLn;
FROM FileSystem IMPORT Lookup, Close, File, Response,
                       ReadNBytes, WriteNBytes;
FROM SYSTEM     IMPORT AX,BX,CX,DX,DS,SWI,GETREG,SETREG,
                       ADDRESS,ADR;

VAR DiskTransAdr : ARRAY[1..43] OF CHAR;         (* Must be Global *)

(*****************************************************************)
PROCEDURE ReadFileStats(FileName      : ARRAY OF CHAR;
                        FirstFile     : BOOLEAN;
                        VAR FilePt    : FileDataPointer;
                        VAR FileError : BOOLEAN);
VAR MaskAddr     : ADDRESS;
    Error        : CARDINAL;
    Index        : CARDINAL;
BEGIN
   IF FirstFile THEN
      FOR Index := 1 TO 43 DO                 (* Clear out the DTA *)
         DiskTransAdr[Index] := " ";
      END;

      SETREG(AX,01A00H);       (* Set up the Disk Transfer Address *)
      MaskAddr := ADR(DiskTransAdr);
      SETREG(DS,MaskAddr.SEGMENT);
      SETREG(DX,MaskAddr.OFFSET);
      SWI(021H);

      MaskAddr := ADR(FileName);
      SETREG(AX,04E00H);                         (* Get first file *)
      SETREG(DS,MaskAddr.SEGMENT);
      SETREG(DX,MaskAddr.OFFSET);
      SETREG(CX,017H);                  (* Attribute for all files *)
      SWI(021H);
   ELSE
      SETREG(AX,04F00H);                   (* Get additional files *)
      SWI(021H);
   END;
   GETREG(AX, Error);
   Error := Error MOD 256;                 (* Logical AND with 255 *)
   IF Error = 0 THEN
      FileError := FALSE;  (* Good read, put data in the structure *)
      FOR Index := 0 TO 13 DO    (* Put all blanks in the filename *)
         FilePt^.Name[Index] := ' ';
      END;
      Index := 0;
      REPEAT                            (* Copy filename to record *)
         FilePt^.Name[Index] := DiskTransAdr[Index + 31];
         Index := Index + 1;
      UNTIL (Index > 11) OR (DiskTransAdr[Index + 31] = 000C);
      FilePt^.Name[12] := 000C;               (* ASCIIZ terminator *)

      FilePt^.Attr := ORD(DiskTransAdr[22]);
      FilePt^.Time := 0;                             (* Ignore Time *)
      FilePt^.Date := 0;                             (* Ignore Date *)
      FilePt^.Size := 65536.0 * FLOAT(ORD(DiskTransAdr[29]))
                    + 256.0 * FLOAT(ORD(DiskTransAdr[28]))
                    +         FLOAT(ORD(DiskTransAdr[27]));
      FilePt^.Left := NIL;
      FilePt^.Right := NIL;
   ELSE
      FileError := TRUE;
   END; (* of IF Error = 0 *)

END ReadFileStats;
```

```
(*****************************************************************)
PROCEDURE GetDiskStatistics(Drive           : CHAR;
                            VAR SectorsPerCluster : CARDINAL;
                            VAR FreeClusters      : CARDINAL;
                            VAR BytesPerSector    : CARDINAL;
                            VAR TotalClusters     : CARDINAL);
VAR DriveCode : INTEGER;
BEGIN
   DriveCode := INTEGER(ORD(Drive)) - 64;
   IF (DriveCode > 17) OR (DriveCode < 0) THEN
      WriteString("Error - Drive code invalid ---> ");
      Write(Drive);
      WriteLn;
      SectorsPerCluster := 0;
      FreeClusters := 0;
      BytesPerSector := 0;
      TotalClusters := 0;
   ELSE
      SETREG(AX,03600H);
      SETREG(DX,DriveCode);
      SWI(021H);
      GETREG(BX,FreeClusters);
      GETREG(AX,SectorsPerCluster);
      GETREG(CX,BytesPerSector);
      GETREG(DX,TotalClusters);
      IF SectorsPerCluster = 0FFFFH THEN
         WriteString("Error - Drive doesn't exist ---> ");
         Write(Drive);
         WriteLn;
         SectorsPerCluster := 0;
         FreeClusters := 0;
         BytesPerSector := 0;
         TotalClusters := 0;
      END;
   END;
END GetDiskStatistics;




(*****************************************************************)
PROCEDURE ChangeToDirectory(Directory : ARRAY OF CHAR;
                            CreateIt : BOOLEAN;
                            VAR ErrorReturn : BOOLEAN);

VAR MaskAddr : ADDRESS;
    Good     : CARDINAL;

     PROCEDURE CHDIR(Path : ARRAY OF CHAR;
                     VAR Error : CARDINAL);
     BEGIN
        MaskAddr := ADR(Path);
        SETREG(AX,03B00H);
        SETREG(DX,MaskAddr.OFFSET);
        SETREG(DS,MaskAddr.SEGMENT);
        SWI(021H);
```

```
         GETREG(AX,Error);
         Error := Error MOD 256;
      END CHDIR;

      PROCEDURE MKDIR(Path : ARRAY OF CHAR;
                      VAR Error : CARDINAL);
      BEGIN
         MaskAddr := ADR(Path);
         SETREG(AX,03900H);
         SETREG(DX,MaskAddr.OFFSET);
         SETREG(DS,MaskAddr.SEGMENT);
         SWI(021H);
         GETREG(AX,Error);
         Error := Error MOD 256;
      END MKDIR;

      PROCEDURE CreateAndChangeDirectory(Directory : ARRAY OF CHAR);
      VAR SubDir  : ARRAY[0..64] OF CHAR;
          Index   : CARDINAL;
          Correct : CARDINAL;
      BEGIN
         Index := 0;
         REPEAT                        (* Find the terminating zero *)
            SubDir[Index] := Directory[Index];
            Index := Index + 1;
         UNTIL (Directory[Index] = 000C) OR (Index = 64);
         SubDir[Index] := 000C;
         REPEAT                            (* Remove a subdirectory *)
            SubDir[Index] := 000C;
            IF Index > 2 THEN
               Index := Index - 1;
            END;
         UNTIL (Index = 2) OR (SubDir[Index] = '\');
         IF Index > 2 THEN
            SubDir[Index] := 000C;          (* Blank out trailing \ *)
         END;
         CHDIR(SubDir,Correct);
         IF (Correct <> 0) AND         (* SubDir Doesn't exist, AND *)
                   (Index > 2) THEN        (* subdirs still in list *)
            CreateAndChangeDirectory(SubDir);
            MKDIR(SubDir,Correct);         (* Make the subdirectory *)
            CHDIR(SubDir,Correct);       (* Change the subdirectory *)
         END;
      END CreateAndChangeDirectory;
BEGIN
   CHDIR(Directory,Good);
   IF Good = 0 THEN                  (* Change to dir if it exists *)
      ErrorReturn := FALSE;
   ELSIF CreateIt THEN              (* Create and change directory *)
      CreateAndChangeDirectory(Directory);
      MKDIR(Directory,Good);
      CHDIR(Directory,Good);
      ErrorReturn := FALSE;
   ELSE                       (* Dir doesn't exist, return an error *)
      ErrorReturn := TRUE;
   END;
```

```
END ChangeToDirectory;




(*****************************************************************)
PROCEDURE CopyFile(SourceFile       : ARRAY OF CHAR;
                   DestinationFile  : ARRAY OF CHAR;
                   FileSize         : REAL;
                   VAR ResultOfCopy : CARDINAL);


TYPE BufferType = ARRAY [1..1024] OF CHAR;

VAR InputFile  : File;
    OutputFile : File;
    Buffer     : BufferType;
    BufferPtr  : POINTER TO BufferType;
    BlockSize  : CARDINAL;
    Number     : CARDINAL;
BEGIN
   Lookup(InputFile,SourceFile,FALSE);
   IF InputFile.res = done THEN
      Lookup(OutputFile,DestinationFile,TRUE);
      IF OutputFile.res = done THEN
         BufferPtr := ADR(Buffer[1]);
         WHILE FileSize > 0.0 DO
            IF FileSize > 1024.0 THEN
               BlockSize := 1024;
               FileSize := FileSize - 1024.0;
            ELSE
               BlockSize := TRUNC(FileSize);
               FileSize := 0.0;
            END;
            ReadNBytes(InputFile,BufferPtr,BlockSize,Number);
            WriteNBytes(OutputFile,BufferPtr,BlockSize,Number);
         END;
         ResultOfCopy := 0;                   (* Good copy made *)
         Close(OutputFile);
      ELSE
         ResultOfCopy := 2;         (* Cannot open destination file *)
         WriteString("Unable to open Destination file ---> ");
         WriteString(DestinationFile);
         WriteLn;
      END;
      Close(InputFile);
   ELSE
      ResultOfCopy := 1;
      WriteString("Unable to open Source file ---> ");
      WriteString(SourceFile);
      WriteLn;
   END;
END CopyFile;


BEGIN
```

```
END DirHelps.
```

This global module contains several useful file handling and directory manipulation procedures. It is called by the above three example programs used for backup and restore of a fixed disk. These routines are available for your use also if you desire to use them for a file manipulation program. Their main intent however is that they be a guide for the student to observe methods used to write library functions.

BITOPS.DEF BITOPS.MOD

```
DEFINITION MODULE BitOps;

(*          Copyright (c) 1987 - Coronado Enterprises              *)

EXPORT QUALIFIED LogicalAND,      (* Note;                         *)
                 LogicalOR,       (*   All of these operations are *)
                 LogicalXOR,      (*   performed in a bitwise man- *)
                 LogicalNOT;      (*   ner with no carry to higher *)
                                  (*   level bits.                 *)

PROCEDURE LogicalAND(In1, In2 : CARDINAL) : CARDINAL;
                    (* This procedure obtains the logical AND of   *)
                    (* the arguments and returns the value.        *)

PROCEDURE LogicalOR(In1, In2 : CARDINAL) : CARDINAL;
                    (* This procedure obtains the logical OR of    *)
                    (* the arguments and returns the value.        *)

PROCEDURE LogicalXOR(In1, In2 : CARDINAL) : CARDINAL;
                    (* This procedure obtains the logical XOR of   *)
                    (* the argiments and returns the value.        *)

PROCEDURE LogicalNOT(In1 : CARDINAL) : CARDINAL;
                    (* This procedure returns the bitwise comple-  *)
                    (* ment of the argument.                       *)

END BitOps.



IMPLEMENTATION MODULE BitOps;

(*            Coptright (c) 1987 - Coronado Enterprises            *)

(* The logical operations performed here are done by converting    *)
(* the input CARDINAL values into type BITSET and using the        *)
(* resulting properties of the BITSET type to perform the required *)
(* operations.                                                     *)

PROCEDURE LogicalAND(In1, In2 : CARDINAL) : CARDINAL;
VAR Result : BITSET;
```

```
BEGIN
   Result := BITSET(In1) * BITSET(In2);
   RETURN CARDINAL(Result);
END LogicalAND;


PROCEDURE LogicalOR(In1, In2 : CARDINAL) : CARDINAL;
VAR Result : BITSET;
BEGIN
   Result := BITSET(In1) + BITSET(In2);
   RETURN CARDINAL(Result);
END LogicalOR;


PROCEDURE LogicalXOR(In1, In2 : CARDINAL) : CARDINAL;
VAR Result : BITSET;
BEGIN
   Result := BITSET(In1) / BITSET(In2);
   RETURN CARDINAL(Result);
END LogicalXOR;


PROCEDURE LogicalNOT(In1 : CARDINAL) : CARDINAL;
VAR Result : BITSET;
BEGIN
   Result := BITSET(In1) / BITSET(0177777B);
   RETURN CARDINAL(Result);
END LogicalNOT;

END BitOps.
```

This module has several generic bit operations such as logical AND, OR, etc. and shift operations. These are useful procedures that you can import and use in your programs if you are doing bit manipulations.

REAL2MON.DEF REAL2MON.MOD

```
DEFINITION MODULE Real2Mon;

(*          Copyright (c) 1987 - Coronado Enterprises            *)

EXPORT QUALIFIED WriteReal;

(* This procedure allows writing to the monitor in a fully       *)
(* formatted manner (i.e. XXXXXX.XXX) instead of the scientific  *)
(* notation which is available in the Logitech library.          *)

PROCEDURE WriteReal(DataOut  : REAL;
                    FieldSize : CARDINAL;
                    Digits    : CARDINAL);
          (* Writes a REAL to the monitor with "FieldSize" total *)
          (* columns and "Digits" significant places after the   *)
          (* decimal point.                                      *)

END Real2Mon.



IMPLEMENTATION MODULE Real2Mon;

(*            Copyright (c) 1987 - Coronado Enterprises           *)

FROM InOut IMPORT Write;

VAR OutString : ARRAY[0..80] OF CHAR;

(* This procedure uses a rather lengthy method for decomposing the *)
(* REAL number and forming it into single characters.  There is a  *)
(* procedure available in the Logitech library to do this for you  *)
(* but this method is kept as an example of how to decompose the   *)
(* number to prepare it for output.  It could be much more effi-   *)
(* cient to use the Logitech library call. The Procedure is named  *)
(* RealConversions.RealTOString, see your library reference.       *)

PROCEDURE WriteReal(DataOut  : REAL;
                    FieldSize : CARDINAL;
                    Digits    : CARDINAL);

VAR Index          : CARDINAL;
    Field          : CARDINAL;
    Count          : CARDINAL;
    WholeFieldSize : CARDINAL;
    ABSDataOut     : REAL;
    Char           : CHAR;
    RoundReal      : REAL;

BEGIN
   IF DataOut >= 0.0 THEN   (* Get the absolute value to work with *)
      ABSDataOut := DataOut;
   ELSE
      ABSDataOut := -DataOut;
   END;

                         (* Make sure the Digits field is positive *)
   IF Digits < 0 THEN
      Digits := 0;
   END;

        (* Make sure there are 3 or more digits for the whole part *)
   IF (FieldSize - Digits) < 3 THEN
      FieldSize := Digits + 3;
   END;

   RoundReal := 0.5;          (* This is used for rounding the data *)
   IF Digits = 0 THEN
      WholeFieldSize := FieldSize;
   ELSE
      WholeFieldSize := FieldSize - Digits - 1;
      FOR Count := 1 TO Digits DO
         RoundReal := RoundReal * 0.1;    (* Reduce for each digit *)
      END;
   END;
   ABSDataOut := ABSDataOut + RoundReal;    (* Add rounding amount *)

   Count := 0;
   WHILE ABSDataOut >= 1.0 DO
      Count := Count + 1;              (* Count significant digits *)
      ABSDataOut := 0.1 * ABSDataOut;
   END;

   WHILE WholeFieldSize > (Count + 1) DO  (* Output leading blanks *)
      Write(" ");
      WholeFieldSize := WholeFieldSize - 1;
   END;

   IF DataOut >= 0.0 THEN          (* Output the sign (- or blank) *)
      Write(" ");
   ELSE
      Write("-");
   END;

   WHILE Count > 0 DO       (* Output the whole part of the number *)
      ABSDataOut := 10.0 * ABSDataOut;
      Index := TRUNC(ABSDataOut);
      Char := CHR(Index + 48);                   (* 48 = ASCII '0' *)
      Write(Char);
      ABSDataOut := ABSDataOut - FLOAT(Index);
      Count := Count - 1;
   END;

   IF Digits > 0 THEN  (* Output the fractional part of the number *)
      Write('.');
      FOR Count := 1 TO Digits DO
         ABSDataOut := 10.0 * ABSDataOut;
         Index := TRUNC(ABSDataOut);
         Char := CHR(Index + 48);                (* 48 = ASCII '0' *)
         Write(Char);
         ABSDataOut := ABSDataOut - FLOAT(Index);
      END;
   END;
END WriteReal;

END Real2Mon.
```

This module has a procedure to output REAL data to the monitor in a neat, easy to read format. It is documented in the header of the source files.

REAL2FIL.DEF REAL2FIL.MOD

```
DEFINITION MODULE Real2Fil;
```

```
(*          Copyright (c) 1987 - Coronado Enterprises           *)

FROM FileSystem IMPORT File;

EXPORT QUALIFIED WriteLnFile, WriteStringFile, WriteCardFile,
                 WriteIntFile, WriteOctFile, WriteHexFile,
                 WriteRealFile;

(* These routines are used to output formatted data to a file.   *)
(* They are used much like the standard output procedures that   *)
(* are available in the module "InOut".  The only real differ-   *)
(* ence is in the REAL output procedure which allows inputting   *)
(* the total field size, and the number of digits after the      *)
(* decimal point.                                                *)

PROCEDURE WriteLnFile(VAR FileName : File);
             (* Writes a return/linefeed to the file.           *)

PROCEDURE WriteStringFile(VAR FileName : File;
                          String   : ARRAY OF CHAR);
             (* Writes the string to the file.                  *)

PROCEDURE WriteCardFile(VAR FileName : File;
                        DataOut   : CARDINAL;
                        FieldSize : CARDINAL);
             (* Writes a CARDINAL to the file.                  *)

PROCEDURE WriteIntFile(VAR FileName : File;
                       DataOut   : INTEGER;
                       FieldSize : CARDINAL);
             (* Writes an INTEGER to the file.                  *)

PROCEDURE WriteOctFile(VAR FileName : File;
                       DataOut   : CARDINAL;
                       FieldSize : CARDINAL);
             (* Writes a CARDINAL to the file in an octal format *)

PROCEDURE WriteHexFile(VAR FileName : File;
                       DataOut   : CARDINAL;
                       FieldSize : CARDINAL);
             (* Writes a CARDINAL to the file in a hex format.   *)

PROCEDURE WriteRealFile(VAR FileName : File;
                        DataOut   : REAL;
                        FieldSize : CARDINAL;
                        Digits    : CARDINAL);
             (* Writes a REAL to the file with "FieldSize" total *)
             (* columns and "Digits" significant places after    *)
             (* the decimal point.                               *)

END Real2Fil.
```

```
IMPLEMENTATION MODULE Real2Fil;

(*          Copyright (c) 1987 - Coronado Enterprises             *)

FROM ASCII       IMPORT EOL;
FROM FileSystem  IMPORT File, WriteChar;
FROM Conversions IMPORT ConvertCardinal, ConvertInteger,
                        ConvertOctal, ConvertHex;

VAR OutString : ARRAY[0..80] OF CHAR;



PROCEDURE WriteLnFile(VAR FileName : File);
BEGIN
   WriteChar(FileName,EOL);
END WriteLnFile;



PROCEDURE WriteStringFile(VAR FileName : File;
                          String   : ARRAY OF CHAR);
VAR Index : CARDINAL;
BEGIN
   Index := 0;
   WHILE String[Index] <> 000C DO
      WriteChar(FileName,String[Index]);
      Index := Index + 1;
   END;
END WriteStringFile;



PROCEDURE WriteCardFile(VAR FileName : File;
                        DataOut   : CARDINAL;
                        FieldSize : CARDINAL);
VAR Index : CARDINAL;
BEGIN
   ConvertCardinal(DataOut,6,OutString);
   WHILE FieldSize > 6 DO
      WriteChar(FileName," ");
      FieldSize := FieldSize - 1;
   END;
   FOR Index := 0 TO 5 DO
      IF (OutString[Index] <> " ") OR ((6 - Index) <= FieldSize) THEN
         WriteChar(FileName,OutString[Index]);
      END;
   END;
END WriteCardFile;



PROCEDURE WriteIntFile(VAR FileName : File;
                       DataOut   : INTEGER;
                       FieldSize : CARDINAL);
VAR Index : CARDINAL;
```

```
BEGIN
   ConvertInteger(DataOut,6,OutString);
   WHILE FieldSize > 6 DO
      WriteChar(FileName," ");
      FieldSize := FieldSize - 1;
   END;
   FOR Index := 0 TO 5 DO
      IF (OutString[Index] <> " ") OR ((6 - Index) <= FieldSize) THEN
         WriteChar(FileName,OutString[Index]);
      END;
   END;
END WriteIntFile;



PROCEDURE WriteOctFile(VAR FileName : File;
                       DataOut   : CARDINAL;
                       FieldSize : CARDINAL);
VAR Index : CARDINAL;
BEGIN
   ConvertOctal(DataOut,6,OutString);
   WHILE FieldSize > 6 DO
      WriteChar(FileName," ");
      FieldSize := FieldSize - 1;
   END;
   FOR Index := (6 - FieldSize) TO 5 DO
      WriteChar(FileName,OutString[Index]);
   END;
END WriteOctFile;



PROCEDURE WriteHexFile(VAR FileName : File;
                       DataOut   : CARDINAL;
                       FieldSize : CARDINAL);
VAR Index : CARDINAL;
BEGIN
   ConvertHex(DataOut,4,OutString);
   WHILE FieldSize > 4 DO
      WriteChar(FileName," ");
      FieldSize := FieldSize - 1;
   END;
   FOR Index := (4 - FieldSize) TO 3 DO
      WriteChar(FileName,OutString[Index]);
   END;
END WriteHexFile;


(* This procedure uses a rather lengthy method for decomposing the *)
(* REAL number and forming it into single characters.  There is a  *)
(* procedure available in the Logitech library to do this for you  *)
(* but this method is kept as an example of how to decompose the   *)
(* number to prepare it for output.  It could be much more effi-   *)
(* cient to use the Logitech library call. The Procedure is named  *)
(* RealConversions.RealTOString, see your library reference.       *)
```

```
PROCEDURE WriteRealFile(VAR FileName : File;
                        DataOut   : REAL;
                        FieldSize : CARDINAL;
                        Digits    : CARDINAL);

VAR Index          : CARDINAL;
    Field          : CARDINAL;
    Count          : CARDINAL;
    WholeFieldSize : CARDINAL;
    ABSDataOut     : REAL;
    Char           : CHAR;
    RoundReal      : REAL;

BEGIN
   IF DataOut >= 0.0 THEN   (* Get the absolute value to work with *)
      ABSDataOut := DataOut;
   ELSE
      ABSDataOut := -DataOut;
   END;

                         (* Make sure the Digits field is positive *)
   IF Digits < 0 THEN
      Digits := 0;
   END;

        (* Make sure there are 3 or more digits for the whole part *)
   IF (FieldSize - Digits) < 3 THEN
      FieldSize := Digits + 3;
   END;

   RoundReal := 0.5;         (* This is used for rounding the data *)
   IF Digits = 0 THEN
      WholeFieldSize := FieldSize;
   ELSE
      WholeFieldSize := FieldSize - Digits - 1;
      FOR Count := 1 TO Digits DO
         RoundReal := RoundReal * 0.1;    (* Reduce for each digit *)
      END;
   END;
   ABSDataOut := ABSDataOut + RoundReal;    (* Add rounding amount *)

   Count := 0;
   WHILE ABSDataOut >= 1.0 DO
      Count := Count + 1;              (* Count significant digits *)
      ABSDataOut := 0.1 * ABSDataOut;
   END;

   WHILE WholeFieldSize > (Count + 1) DO  (* Output leading blanks *)
      WriteChar(FileName," ");
      WholeFieldSize := WholeFieldSize - 1;
   END;

   IF DataOut >= 0.0 THEN           (* Output the sign (- or blank) *)
      WriteChar(FileName," ");
   ELSE
      WriteChar(FileName,"-");
   END;

   WHILE Count > 0 DO       (* Output the whole part of the number *)
      ABSDataOut := 10.0 * ABSDataOut;
      Index := TRUNC(ABSDataOut);
      Char := CHR(Index + 48);                    (* 48 = ASCII '0' *)
      WriteChar(FileName,Char);
      ABSDataOut := ABSDataOut - FLOAT(Index);
      Count := Count - 1;
   END;

   IF Digits > 0 THEN  (* Output the fractional part of the number *)
      WriteChar(FileName,'.');
      FOR Count := 1 TO Digits DO
         ABSDataOut := 10.0 * ABSDataOut;
         Index := TRUNC(ABSDataOut);
         Char := CHR(Index + 48);                 (* 48 = ASCII '0' *)
         WriteChar(FileName,Char);
         ABSDataOut := ABSDataOut - FLOAT(Index);
      END;
   END;
END WriteRealFile;

END Real2Fil.
```

This module has several procedures to output REAL and other data type to a file using the FileSystem MODULE. The various procedures are documented in their respective headers.