920*i*

Programmable Indicator

iRite

Programming Reference

RICE LAKE WEIGHING SYSTEMS
Industrial Solutions on a Global Scale®

ISO 9001 REGISTERED

67888

Contents

About This Manual

This manual is intended for use by programmers who write *iRite* applications for *920i* digital weight indicators.

This manual should be used in conjunction with the *920i Installation Manual*. See that manual for detailed descriptions of indicator capability and operation.

Warning *All programs should be thoroughly tested before implementation in a live system. To prevent personal injury and equipment damage, software-based interrupts must always be supplemented by emergency stop switches and other safety devices necessary for the application.*

Authorized distributors and their employees can view or download this manual from the Rice Lake Weighing Systems distributor site at www.rlws.com.

1.0 Introduction

1.1 What is iRite?

iRite is a programming language developed by Rice Lake Weighing Systems and used for the purpose of programming the *920i* programmable indicator. Similar to other programming languages, *iRite* is a set of rules, called syntax, for composing instructions in a format that a compiler can understand.

An *iRite* program is nothing more than a text file, which contains statements composed following the *iRite* language syntax. The text file created using the *iRite* programming language isn't much use until it is compiled. Compiling is done using a compiler program.

The compiler reads the text file written in *iRite* and translates the program's intent into commands that are understandable to the *920i*'s serial interface. In addition, with an ample amount of appropriate comments, the same *iRite* program that is understandable to the compiler should also relate, to any person reading the file, what the program is meant to accomplish.

1.2 Why iRite?

Although there are many different programming languages already established in the programming world, some of which you may already be familiar with, none of them were "the right tool for the job."

Most other programming languages are very general and try to maximize flexibility in unknown or unforeseen applications; hence they carry a lot of overhead and functionality that the *920i* programmer might not ever use.

Considering the varying backgrounds and experiences of the people that will be doing most of the *iRite* programming, we wanted a language that was easy to learn and use for the first-time programmer, but also familiar in syntax to an experienced programmer. Furthermore, we wanted to eliminate some of the unnecessary features that are troublesome in other languages, namely the *pointer* data type. In addition, we added some items that are very useful when programming the *920i*, the *database* data type and the handler subprogram, for example.

Also by creating a new language, we had the luxury of picking the best features from other languages, with the advantage of hindsight. The result is *iRite*: a compact language (only six discrete statement types, three data types) with a general syntax similar to Pascal and Ada, the string manipulation of Basic, and a rich set of function calls and built-in types specific to the weighing and batching industry. A Pascal-like syntax was adopted because Pascal was originally developed as a teaching language and its syntax is unambiguous.

1.3 About iRite Programs

The *920i* indicator has, at any given moment, many time critical tasks it must accomplish. It is always calculated new weight from new analog information, updating the display, watching for key press events, running the setpoint engine, watching for serial input, streaming weight data, or sending print data out one or more serial ports. In addition to these tasks, it also runs user programmed custom event handlers, i.e. an *iRite* program.

Writing custom event handlers is what *iRite* is for. Each of the *920i* tasks share processor time, but some tasks have higher priorities than other tasks. If a low priority task is taking more than its share of processor time (1/1200 of a second, it will be suspended so a higher priority task can be given processor time when it needs it. Then, when all the other higher priority tasks have completed, the low priority task will be resumed.

Gathering analog weight signals and converting it to weight data is the *920i*'s highest priority. Running a user-defined program has a very low priority. Streaming data out a serial port is the lowest priority task, because of its minimal computational requirements. This means that if your *iRite* program "hangs", the task of streaming out the serial ports will never get and CPU time and streaming will never happen. An example of interrupting a task would be if a user program included an event handler for SP1Trip (Setpoint 1 Trip Event) and this event "fired".

Let's assume the logic for the SP1Trip event is executing at a given moment in time. In this example, the programmer wanted to display the message "Setpoint 1 Tripped" on the display. If the SP1Trip event logic doesn't complete by the time the *920i* needs to calculate a new weight, for example, the SP1Trip handler will be interrupted immediately, a new weight will be calculated, and the SP1Trip event will resume executing exactly where it was interrupted. In most circumstances, this happens so quickly the user will never know that the SP1Trip handler was ever interrupted.

How Do I write and Compile iRite Programs?

You can use any flat ASCII text editor to create *iRite* source files, but use of the *iRev* Editor is strongly recommended because of its many *iRite*-specific utilities. An editor like Microsoft Word should not be used unless you specifically tell it to save the files as text only (*.txt).

iRite source files are named with the *.src* extension. Once you have the *iRev* Editor open, you are ready to start writing a program.

At this point it is worth mentioning the templates and sample programs RLWS has made available. In order to get started with the skeleton of a working program, a program that demonstrates the recommended indentation and commenting, it would be best to start by opening an existing program with your editor.

In addition to writing *.src* files you may write include files with an extension *.iri*. The *iRite* language doesn't have the ability to include files, but when using *iRev* you can. An include file can be helpful in keeping your *.src* program from getting cluttered with small unrelated functions and procedures that get used in many different programs. For example, you could create a file named *math.iri* and put only functions that perform some kind of math operation not supported in the *iRite* library already. When the program is compiled through *iRev*, the *.iri* file is placed where you told it to be placed in *iRev*. Because *iRite* enforces "declaration before use", the iri file needs to be placed before any of the subprograms in your *.src* file.

When you are ready to compile your program, use the "Compile" feature from the "Tools" menu in the *iRev* Editor. If the program compiles without errors a new text file is created. This new text file has the same name but an extension of *.cod*. The new file named *your_program.cod* is a text file containing commands that can be sent to the *920i* via an RS232 serial communication connection between your computer and the *920i*. Although the *.cod* file is a text file, most of it will not be understandable. There is really no reason to edit the *.cod* file and we strongly discourage doing so.

How Do I Get My Program into the 920i?

The *920i* indicator must be in configuration mode before the *.cod* file can be sent. The easiest way to send the *.cod* file to the *920i* is to use *iRev*. You can use the send the *Send .COD file to Indicator* option under the *Tools* menu in the *iRev* Editor, or you can send the *.cod* file directly from *iRev* by using the *Download Configuration...* selection on the *Communications* menu and specifying that you want to send the *.cod* file.

If the *920i* indicator is not in configuration mode, *iRev* will pop-up a message informing you of this condition. It is strongly recommended that you use *iRev* or the *iRev* Editor to send the compiled program to the *920i*. This method implements error checking on each string sent to the indicator and helps protect from data transmission errors corrupting the program.

You can also send the *.cod* file with a communication program like Procomm Plus or Terminal. Simply use the send file tool and set the protocol to RAW ASCII. However, with this method there is no error checking and you can't be sure that the program was sent correctly.

1.4 Running Your Program

A program written for the *920i* is simply a collection of one or more custom event handlers and their supporting subprograms. A custom event handler is run whenever the associated event occurs. The *ProgramStartup* event is called whenever the indicator is powered up, is taken out of configuration mode, or is sent the RS serial command. It should be straightforward when the other event handlers are called. For example, the *DotKeyPressed* event handler is called when ever the "." key is pressed.

All events have built-in intrinsic functionality associated with them, although, the intrinsic functionality may be to do nothing. If you write a custom event handler for an event, your custom event handler will be called instead of the intrinsic function, and the default action will be suppressed.

For example, the built-in intrinsic function of the UNITS key is to switch between primary, secondary, and tertiary units. If the handler *UnitsKeyPressed* was defined in a user program, then the UNITS key no longer switches between primary, secondary, and tertiary units, but instead does what ever is written in the handler *UnitsKeyPressed*. The ability to turn off the custom event handler and return to the intrinsic functionality is provided by the *DisableHandler* function.

It is important to note that only one event handler can be running at a time. This means that if an event occurs while another event handler is running, the new event will not be serviced immediately but instead will be placed in a queue and serviced after the current event is done executing.

This means that if you are executing within an infinite loop in an event handler, then no other event handlers will ever get serviced. This doesn't mean that the indicator will be totally locked-up: The *920i* will still be executing its other tasks, like calculating current weights, and running the setpoint engine. But it will not run any other custom event handlers while one event is executing in an infinite loop.

There are some fatal errors that an *iRite* program can make that will completely disable the *920i*. Some of these errors are "...divide by zero", "string space exhausted", and "array bounds violation". When they occur, the *920i* stops processing and displays a fatal error message on the display. Power must be cycled to reset the indicator.

After the indicator has been restarted, it should be put into setup mode, and a new version (without the fatal error) of the *iRite* program should be loaded. If you are unfortunate enough to program a fatal error in your ProgramStartup Handler, then cycling power to the unit will only cause the ProgramStartup Handler to be run again and repeat the fatal error.

In this case you must perform a RESETCONFIGURATION. Your program, along with the configuration, will be erased and set to the defaults. This will allow you to reload your *iRite* program after you have corrected the code that generated the fatal error and re-compiled the program.

1.5 Sound Programming Practices

The most important thing to remember about writing source code is that it has two very important functions: it must work, and it must clearly communicate how it works. At first glance, especially to a beginning programmer, it may seem that getting the program to work is more important than clearly commenting and documenting *how* it works.

As a professional programmer, you will realize that a higher quality product is produced, which is less costly to maintain, when the source code is well documented. You, somebody else at your organization, the customer, or RLWS Support Personnel, may need to look at some *iRite* source code, months or years from now, long after the original author has forgotten how the program worked or isn't around to ask. This is why we advocate programming to a specific standard. The template programs, example programs, and purchased custom programs that are available from RLWS follow a single standard. You are welcome to download this standard from our website, or you can write your own.

The purpose of a standard is to document the way all programmers will create software for the *920i* indicator. When the standard is followed, the source code will be easy to follow and understand. The standard will document: the recommended style and form for module, program, and subprogram headers, proper naming conventions for variables and functions, guidelines for function size and purpose, commenting guidelines, and coding conventions.

NOTE: *Some of the examples in this manual have had some or all the comments removed. This was done only for the purpose of emphasizing the actual iRite language source code. The English description accompanying each example should be sufficient to explain the example.*

2.0 Tutorial

2.1 Getting Started

Traditionally, the first program a programmer writes in every language is the famous “Hello World!” program. Being able to write, compile, download, and run even the simple “Hello World!” program is a major milestone. Once you have accomplished this, the basics components will be in place, and the door will be open for you and your imagination to start writing real world solutions to some challenging tasks.

Here is the “Hello World!” program in *iRite*:

```
01   program HelloWorld;
02
03   begin
04     DisplayStatus("Hello, world!");
05   end HelloWorld;
```

This program will display the text *Hello, world!* on the *920i*’s display in the status message area, every time the indicator is turned on, taken out of configuration mode, or reset. Let’s take a closer look at each line of the program.

Line 1: **`program`** `HelloWorld;`

The first line is the program header. The program header consists of the keyword **program** followed by the name of the program. The name of the program is arbitrary and made up by the programmer. The program name; however, must follow the identifier naming rules (i.e. an identifier can’t start with a number or contain a space).

The second line is an optional blank line. Blank lines can be placed anywhere in the program to separate important lines and to make the program easier to read and understand.

Line 3: **`begin`**

The **begin** keyword is the start of the optional main code body. The optional main code body is actually the ProgramStartup event handler. The ProgramStartup handler is the only event handler that doesn’t have to be specifically named.

Line 4:

```
    DisplayStatus("Hello, world!");
```

The statement `DisplayStatus("Hello, world!")` is the only statement in the main code body. It is a call to the built-in procedure DisplayStatus with the string constant “Hello, world!” passed as a parameter. The result is the text, "Hello, world!" will be shown in the status area of the display (lower left corner), whenever the startup event is fired.

Line 5: **`end`** `HelloWorld;`

The keyword **end** followed by the same identifier for the program name used in line one, HelloWorld, is required to end the program.

From this analysis, you may have gathered that only the first and last lines were required. This is true, the program would compile, but it would do nothing and be totally useless. At a minimum, a working program must have at least one event handler, though it doesn’t have to be the ProgramStartup handler. We could have written the HelloWorld program to display “Hello, world!” whenever any key on the keypad was pressed. It would look like this:

```
program HelloWorld;

  handler KeyPressed;
  begin
    DisplayStatus("Hello, world!");
  end;

end HelloWorld;
```

In this version, we chose to use the KeyPressed event handler to call the DisplayStatus procedure. The KeyPressed event will fire any time any key on the keypad is pressed. Also notice that the **begin** keyword that started the main code body, and the DisplayStatus call have been removed and replaced with the four lines making up the KeyPressed event handler definition.

Using the *iRev* Editor, write the original version of the “Hello, world!” program on your system. After you have compiled the program successfully, download it to your *920i*. After the program has been downloaded and the indicator is put back in run mode, then the text *Hello, world!* should appear on the display.

2.2 Program Example with Constants and Variables

The “Hello, world!” program didn’t use any explicitly declared constants or variables (the string “Hello, world!” is actually a constant, but not explicitly declared). Most useful programs use many constants and variables. Let’s look at a program that will calculate the area of a circle for various length radii. The program, named “PrintCircleAreas”, is listed below.

```
01   program PrintCircleAreas;
02
03     -- Declare constants and aliases here.
04     g_ciPrinterPort : constant integer := 2;
05
06     -- Declare global variables here.
07     g_iCount : integer := 1;
08     g_rRadius : real;
09     g_rArea : real;
10     g_sPrintText: string;
11
12
13     function CircleArea(rRadius : real) : real;
14       crPie : constant real := 3.141592654;
15     begin
16       -- The area of a circle is defined by: area = pie*(r^2).
17       return (crPie * rRadius * rRadius);
18     end;
19
20
21   begin
22
23     for g_iCount := 1 to 10
24     loop
25
26       g_rRadius := g_iCount;
27       g_rArea := CircleArea(g_rRadius);
28
29      g_sPrintText := "The area of a circle with radius " + RealToString(g_rRadius, 4, 1)
30                         + " is " + RealToString(g_rArea, 7, 2);
31
32       WriteLn(g_ciPrinterPort, g_sPrintText);
33
34     end loop;
35
36   end PrintCircleAreas;
```

The PrintCircleAreas program demonstrates variables and constants as well as introducing these important ideas: **for** loop, assignment statement, function declarations, function calling and return parameters, string concatenation, WriteLn procedure, a naming convention, comments, and a couple of data conversion functions.

You probably know by now that this program will calculate the areas of circles with radius from 1 to 10 (counting by 1s) and send text like, “The area of a circle with radius 1 is 3.14,” once for each radius, out the communication port 2.

```
01   program PrintCircleAreas;
```

Line 1 is the program header with the keyword **program** and the program identifier “PrintCircleAreas”. This is the same in theory as the “HelloWorld” program header.

Line 3 is a comment. In *iRite* all comments are started with a `--` (double dash). All text after the double dash up to the end of the line is considered a comment. Comments are used to communicate to any reader what is going on in the program on the specific lines with the comment or immediately following the comment. The `--` can start on any column in a line and can be after, on the same line, as other valid program statements.

Line 4 is a global constant declaration for the communication port that a printer may be connected to. This simple line has many important parts:

```
04   g_ciPrinterPort : constant integer := 2;
```

First, an identifier name is given. Identifier names are made up by the programmer and should accurately describe what the identifier is used for. In the name g_ciPrinterPort the "PrinterPort" part tells us that this identifier will hold the value of a port where a printer should be connected. The "g_ci" is a prefix used to describe the type of the identifier. When "g_ciPrinterPort" is used later on in the program, the prefix may help someone reading the program, even the program's author, to easily determine the identifier's data type without having to look back at the declaration.

The "g_" in the prefix helps tell us that the identifier is "global". Global identifiers are declared outside of any subprogram (handler, function, procedure) and have global scope. The term "scope" refers to the region of the program text in which the identifier is known and understood. The term "global" means that the identifier is "visible" or "known" everywhere in the program. Global identifiers can be used within an event handler body, or any procedure or function body. Global identifiers also have "program duration". The duration of an identifier refers to when or at what point in the program the identifier is understood, and when their memory is allocated and freed. Identifiers with global duration, in a *920i* program, are understood in all text regions of the program, and their memory is allocated at program start-up and is re-allocated when the indicator is powered up.

The "c" in the prefix helps us recognize that the identifier is a constant. Constants are a special type of identifier that are initialized to a specific value in the declaration and may not be changed anytime or anywhere in the program. Constants are declared by adding the keyword **constant** before the type.

Constants are very useful and make the program more understandable. In this example, we defined the printer port as port 2. If we would have just used the number 2 in the call to WriteLn, then a reader of the program would not have any idea that the programmer intended a printer to be connected to the *920i*'s port 2.

Also, in a larger program, port 2 may be used hundreds of times in Write and WriteLn calls. Then, if it were decided to change the printer port from port 2 to port 3, hundreds of changes would have to be made. With port 2 being a constant, only one change in the declaration of g_ciPrinterPort would be required to change the printer port from 2 to 3.

The type of the constant is an integer. The "i" in the prefix helps us identify g_ciPrinterPort as an integer. The keyword **integer** follows the keyword **constant** and specifies the type compatibility of the identifier as an integer and also determines how much memory will be required to store the value (a value of 2 in this example). In the *iRite* programming language, there are only 3 basic data types: integer, real and string.

The initialization of the constant is accomplished with the ":= 2" part of the statement. Initialization of constants is done in the declaration, with the assignment operator, **:=**, followed by the initial value.

Finally, the statement is terminated by a semicolon. The ";" is used in *iRite* and other languages as a statement terminator and separator. Every *statement* must be terminated with a semicolon. Don't read this to mean "every *line* must end in a semicolon"; this is not true. A statement may be written on one line, but it is usually easier to read if the statement is broken down into enough lines to make some keywords stand out and to keep the length of each line less than 80 characters.

Some statements contain one or more other statements. In our example, the statement:

```
g_ciPrinterPort : constant integer := 2;
```

is an example of a simple statement that easily fit on one line of code. The **`loop`** statement in the program startup handler (main code body) is spread out over several lines and contains many additional statements. It does, however, end with line **`end loop`**;, and ends in a semicolon.

```
06     -- Declare global variables here.
07     g_iCount : integer := 1;
08     g_rRadius : real;
09     g_rArea : real;
10     g_sPrintText: string;
```

Line 6 is another comment to let us know that the global variables are going to be declared.

Lines 7—10 are global variable declarations. One integer, g_iCounter, two reals, g_rRadius and g_rArea, and one string, g_sPrintText, are needed during the execution of this program. Like the constant g_ciPrinterPort, these identifiers are global in scope and duration; however, they are not constants. They may have an optional initial value assigned to them, but it is not required. Their value may be changed any time they are "in scope", they may be changed in every region of the program anytime the program is loaded in the 920i.

Lines 13—18 are our first look at a function declaration. A function is a subprogram that can be invoked (or called) by other subprograms. In the PrintCircleAreas program, the function CircleArea is invoked in the program startup event handler. The radius of a circle is passed into the function when it is invoked. In *iRite* there are three types of subprograms: functions, procedures, and handlers.

```
13    function CircleArea(rRadius : real) : real;
14      crPie : constant real := 3.141592654;
15    begin
16      -- The area of a circle is defined by: area = pie*(r^2).
17      return (crPie * rRadius * rRadius);
18    end;
```

On line 13, the function declaration starts with the keyword **function** followed by the function name. The function name is an identifier chosen by the programmer. We chose the name "CircleArea" for this function because the name tells us that we are going to return the area of a circle. Our function CircleArea has an optional formal arguments (or parameters) list. The formal argument list is enclosed in parenthesis, like this: `(rRadius : real)`. Our example has one argument, but functions and procedures may have zero or more.

Argument declarations must be separated by a semicolon. Each argument is declared just like any other variable declaration: starting with an identifier followed by a colon followed by the data type. The exception is that no initialization is allowed. Initialization wouldn't make sense, since a value is passed into the formal argument each time the function is called (invoked).

The rRadius parameters are passed by value. This means that the radius value in the call is copied in rRadius. If rRadius is changed, there is no effect on the value passed into the function. Unlike procedures, functions may return a value. Our function CircleArea returns the area of a circle. The area is a real number. The data type of the value returned is specified after the optional formal argument list. The type is separated with a colon, just like in other variable declarations, and terminated with a semicolon.

Up to this point in our program, we have only encountered global declarations. On line 14 we have a local declaration. A local declaration is made inside a subprogram and its scope and duration are limited. So the declaration: `crPie : constant real := 3.141592654;` on line 14 declares a constant real named crPie with a value of 3.141592654. The identifier crPie is only known—and only has meaning—inside the text body of the function CircleArea. The memory for crPie is initialized to the value 3.141592654 each time the function is called.

Line 15 contains the keyword **begin** and signals the start of the function code body. A function code body contains one or more statements.

Line 16 is a comment that explains what we are about to do in line 17. Comments are skipped over by the compiler, and are not considered part of the code. This doesn't mean they are not necessary; they are, but are not required by the compiler.

Every function must return a value. The value returned must be compatible with the return type declared on line 14. The keyword **return** followed by a value, is used to return a value and end execution of the function. The **return** statement is always the last statement a function runs before returning. A function may have more then one return statement, one in each conditional execution path; however, it is good programming practice to have only one return statement per function and use a temporary variable to hold the value of different possible return values.

The function code body, or statement lists, is terminated with the **end** keyword on line 18.

In this program we do all the work in the program startup handler. We start this unnamed handler with the **begin** keyword on line 21.

```
23    for g_iCount := 1 to 10
24    loop
```

```
25
26      g_rRadius := g_iCount;
27      g_rArea := CircleArea(g_rRadius);
28
29      g_sPrintText := "The area of a circle with radius " + RealToString(g_rRadius, 4, 1)
30                + " is " + RealToString(g_rArea, 7, 2);
31
32      WriteLn(g_ciPrinterPort, g_sPrintText);
33
34    end loop;
```

On line 23 we see a **for** loop to start the first statement in the startup handler. In *iRite* there are two kinds of looping constructs. The **for** loop and the **while** loop. **For** loops are generally used when you want to repeat a section of code for a predetermined number of times. Since we want to calculate the area of 10 different circles, we chose to use a **for** loop.

For loops use an optional iteration clause that starts with the keyword **for** followed by the name of variable, followed by an assignment statement, followed by the keyword **to**, then an expression, and finally an optional step clause. Our example doesn't use a step clause, but instead uses the implicit step of 1. This means that lines 26 through 32 will be executed ten times. The first time g_iCount will have a value of 1, and during the last iteration, g_iCount will have a value of 10.

All looping constructs (the **for** and the **while**) start with the keyword **loop** and end with the keywords **end loop,** followed by a semicolon. In our example, **loop** is on line 24 and **end loop** is on line 34. In between these two, are found, the statements that make up the body of the loop.

Line 26 is an assignment of an integer data type into a real data type. This line is unnecessary and the assignment could have been made automatically if the integer g_iCount was passed into the function CircleArea directly on line 27, since CircleArea is expecting a real value. Calls to functions like CircleArea are usually done in an assignment statement if the functions return value need to be used later in the program. The return value of CircleArea (the area of a circle with radius g_rRadius) is stored in g_rArea.

The assignment on lines 29 and 30 uses two lines strictly for readability. This single assignment statement does quite a bit. We are trying to create a string of plain English text that will say: "`The area of a circle with radius xx.x is yyyy.yy`", where the radius value will be substituted for `xx.x` and the calculated area will be substituted for `yyyy.yy`. The global variable g_sPrintText is a string data type. The constants (or literals): "`The area of a circle with radius `" and "` is `" are also strings.

However, g_rRadius and g_iArea are real values. We had to use a function from the API to convert the real values to strings. The API function RealToString is passed a real and a width integer and a precision integer. The width parameter specifies the minimum length to reserve in the string for the value. The precision parameter specifies how many places to report to the right of the decimal place. To concatenate all the small strings into one string we use the string concatenation operator, "+".

Finally, we want to send the new string we made to a printer. The Write and WriteLn procedures from the API send text data to a specified port. Earlier in the program we decided the printer port will be stored in g_ciPrinterPort. So the WriteLn call on line 32 send the text stored in g_sPrintText, followed by a carriage return character, out port 2.

If we had a printer connected to port 2 on the *920i*, every time the program startup handler is fired, we would see the following printed output:

```
The area of a circle with radius  1.0 is    3.14
The area of a circle with radius  2.0 is   12.57
The area of a circle with radius  3.0 is   28.27
The area of a circle with radius  4.0 is   50.27
The area of a circle with radius  5.0 is   78.54
The area of a circle with radius  6.0 is  113.10
The area of a circle with radius  7.0 is  153.94
The area of a circle with radius  8.0 is  201.06
The area of a circle with radius  9.0 is  254.47
The area of a circle with radius 10.0 is  314.16
```

3.0 Language Syntax

3.1 Lexical Elements

3.1.1 Identifiers

An identifier is a sequence of letters, digits, and underscores. The first character of an identifier must be a letter or an underscore, and the length of an identifier cannot exceed 100 characters. Identifiers are not case-sensitive: "`HELLO`" and "`hello`" are both interpreted as "`HELLO`".

Examples:

Valid identifiers:	`Variable12`	
	`_underscore`	
	`Std_Deviation`	
Not valid identifiers:	`9abc`	First character must be a letter or an underscore.
	`ABC DEF`	Space (blank) is not a valid character in an identifier.

Identifiers are used by the programmer to name programs, data types, constants, variables, and subprograms. You can name your identifiers anything you want as long as they follow the rules above and the identifiers is not already used as a keyword or as a built-in type or built-in function. Identifiers provide the "name" of an entity. Names are bound to program entities by declarations and provide a simple method of entity reference. For example, an integer variable iCounter (declared `iCounter : integer`) is referred to by the name iCounter.

3.1.2 Keywords

Keywords are special identifiers that are reserved by the language definition and can only be used as defined by the language. The keywords are listed below for reference purposes. More detail about the use of each keyword is provided later in this manual.

and	**array**	**begin**	**builtin**	**constant**	**database**
else	**elsif**	**end**	**exit**	**for**	**function**
handler	**if**	**integer**	**is**	**loop**	**mod**
not	**of**	**or**	**procedure**	**program**	**real**
record	**return**	**step**	**stored**	**string**	**then**
to	**type**	**var**	**while**		

3.1.3 Constants

Constants are tokens representing fixed numeric or character values and are a necessary and important part of writing code. Here we are referring to constants placed in the code when a value or string is known at the time of programming and will never change once the program is compiled. The compiler automatically figures out the data type for each constant.

NOTE*: Be careful not to confuse the constants in this discussion with identifiers declared with the keyword* constant*, although they may both be referred to as constants.*

Three types of constants are defined by the language:

Integer Constants: An integer constant is a sequence of decimal digits. The value of an integer constant is limited to the range 0…$2^{31} - 1$. *Any values outside the allowed range are silently truncated.*

Literally, any time a whole number is used in the text of the program, the compiler creates an integer constant. The following gives examples of situations where an integer constant is used:

```
iCount : integer := 25;
for iIndex := 1 to 3
sResultString := IntegerToString(12345);
sysResult := StartTimer(4);
```

Real Constants:A real constant is an integer constant immediately followed by a decimal point and another integer constant. Real constants conform to the requirements of IEEE-754 for double-precision floating point values. When the compiler “sees” a number in the format *n.n* then a real constant is created. The value .56 will generate a compiler error. Instead compose real constants between –1 and +1 with a leading zero like this: 0.56 and –0.667. The following gives examples of situations where a real constant is used:

```
rLength := 9.25;
if rValue <= 0.004 then
sResultString := RealToString(98.765);
rLogResult := Log(345.67);
```

String Constants:A string constant is a sequence of printable characters delimited by quotation marks (double quotes, " "). The maximum length allowed for a string constant is 1000 characters, including the delimiters. The following gives examples of situations where a string constant (or string literal) is used:

```
sUserPrompt := "Please enter the maximum barrel weight:";
WriteLn(iPrinter, "Production Report (1st Shift));
if sUserEntry = "QUIT" then
  DisplayStatus("Thank You!");
```

3.1.4 Delimiters

Delimiters include all tokens other than identifiers and keywords, including the arithmetic operators listed below:

```
>=   <=   <>   :=   <>   =   +   –   *   /
.    ,    ;    :    (    )   [   ]   "
```

Delimiters include all tokens other than identifiers and keywords. Below is a functional grouping of all of the delimiters in *iRite*.

Punctuation

Parentheses

() (open and close parentheses) group expressions, isolate conditional expressions, and indicate function parameters:

```
iFarenheit := ((9.0/5.0) * iCelcius) + 32;    -- enforce proper precedence
if (iVal >= 12)  and (iVal <= 34) or (iMaxVal > 200)   -- conditional expr.
EnableSP(5);  -- function parameters
```

Brackets

[ ] (open and close brackets) indicate single and multidimensional array subscripts:

```
type CheckerBoard is array [8, 8] of recSquare;
iThirdElement := aiValueArray[3];
```

Comma

The comma(,) separates the elements of a function argument list and elements of a multidimensional array:

```
type Matrix is array [4,8] of integer;
GetFilteredCount(iScale, iCounts);
```

Semicolon

The semicolon (;) is a statement terminator. Any legal *iRite* expression followed by a semicolon is interpreted as a statement. Look around at other examples, its used all over the place.

Colon

The colon (:) is used to separate an identifier from its data type. The colon is also used in front of the equal sign (=) to make the assignment operator:

```
function GetAverageWeight(iScale : integer) : real;
iIndex : integer;
csCopyright : constant string := "2002 Rice Lake Weighing Systems";
```

Quotation Mark

Quotation marks ("") are used to signal the start and end of string constants:

```
if sCommand = "download data" then
  Write(iPCPort, "Data download in progress.  Please wait…");
```

Relational Operators

Greater than (>)

Greater than or equal to (>=)

Less than (<)

Less than or equal to (<=)

Equality Operators

Equal to (=)

Not equal to (<>)

The relational and equality operators are only used in an **if** expression. They may only be used between two objects of compatible type, and the resulting construct will be evaluated by the compiler to be either true or false;

```
if iPointsScored = 6 then
if iSpeed > 65 then
if rGPA <= 3.0 then
if sEntry <> "2" then
```

NOTE: *Be careful when using the equal to (=) operator with real data. Because of the way real data is stored and the amount of precision retained, it may not contain what you would expect. For example, given a real variable named rTolerance:*

```
rTolerance := 10.0 / 3.0
…
if rTolerance * 3 = 10 then
  -- do something
end if;
```

The evaluation of the if *statement will resolve to false. The real value assigned to rTolerance by the expression 10.0 / 3.0 will be a real value (3.333333) that, when multiplied by 3, is not quite equal to 10.*

Logical Operators

Although they are keywords and not delimiters, this is a good place to talk about "Logical Operators". In *iRite* the logical operators are **and**, **or**, and **not** and are named "logical and", "logical or", and "logical negation" respectively. They too are only used in an **if** expression. They can only be used with expressions or values that evaluate to true or false:

```
if (iSpeed > 55) and (not flgInterstate) or (strOfficer = "Cranky") then
  sDriverStatus := "Busted";
```

Arithmetic Operators

The arithmetic operators (+, – ,* , /, and **mod**) are used in expression to add, subtract, multiply, and divide integer and real values. Multiplication and division take precedence over addition and subtraction. A sequence of operations with equal precedence is evaluated from left to right.

The keyword **mod** is not a delimiter, but is included here because it is also an arithmetic operator. The modulus (or remainder) operator returns the remainder when operand 1 is divided by operand 2. For example:

```
rResult : 7 mod 3;   -- rResult should equal 1
```

NOTE: *Both division (/) and mod operations can cause the fatal divide-by-zero error if the second operand is zero.*

When using the divide operator with integers, be careful of losing significant digits. For example, if you are dividing a smaller integer by a larger integer then the result is an integer zero: 4/7 = 0. If you were hoping to assign the result to a real like in the following example:

```
rSlope : real;
rSlope := 4/7;
```

rSlope will still equal 0, not 0.571428671 as might be expected. This is because the compiler does integer math when both operands are integers, and stores the result in a temporary integer. To make the previous statement work in *iRite*, one of the operands must be a real data type or one of the operands must evaluate to a real. So we could write the assignment statement like:

```
rSlope := 4.0/7;
```

If we were dividing two integer variables, we could multiply one of the operands by 1.0 to force the compile to resolve the expression to a real:

```
rSlope : real;
iRise : integer := 4;
iRun : integer := 7;

rSlope := (iRise * 1.0) / iRun;
```

Now rSlope will equal 0.571428671.

NOTE: *The plus sign (+) is also used as the string concatenation operator. The minus sign (–) is also used as a unary minus operator that has the result equal to the negative of its operand.*

Assignment Operator (:=)

The assignment operator is used to assign a value to a compatible program variable or to initialize a constant. The value on the left of the ":=" must be a modifiable value. The following are some invalid examples:

```
3 := 1 + 1;  -- not valid
ciMaxAge := 67; -- where ciMaxAge was declared with keyword constant
iInteger := "This is a string, not an integer!";  -- incompatible types
```

Structure Member Operator ("dot")

The "dot" (.) is used to access the name of a field of a record or database types.

3.2 Program Structure

A program is delimited by a program header and a matching end statement. The body of a program contains a declarations section, which may be empty, and an optional main code body. The declaration section and the main code body may not both be empty.

```
<program>:
  program IDENTIFIER ';'
    <decl-section>
    <optional-main-body>
  end IDENTIFIER ';'
  ;
<optional-main-body>:
    /* NULL */
  | begin <stmt-list>
  ;
```

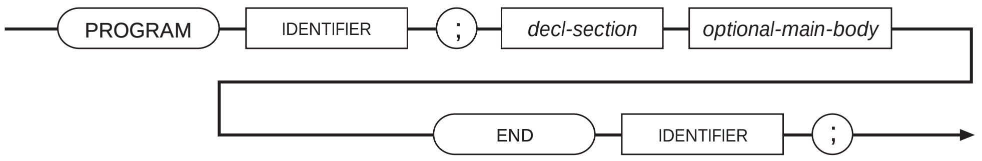

Figure 3-1. Program Statement Syntax

The declaration section contains declarations defining global program types, variables, and subprograms. The main code body, if present, is assumed to be the declaration of the program startup event handler. A program startup event is generated when the instrument personality enters operational mode at initial power-up and when exiting setup mode.

Example:

```
program MyProgram;
  KeyCounter : Integer;
  handler AnyKeyPressed;
```

```
        begin
          KeyCounter := KeyCounter + 1;
        end;

    begin
      KeyCounter := 0
    end MyProgram;
```

The *iRite* language requires declaration before use so the order of declarations in a program is very important. The "declaration before use" requirement is imposed to prevent recursion, which is difficult for the compiler to detect.

In general, it make sense for certain types of declarations to always come before others types of declarations. For example, functions and procedures must always be declared before the handlers. Handlers cannot be called or invoked from within the program, only by the event dispatching system. But functions and procedures can be called from within event handlers; therefore, always declare the functions and procedures before handlers.

Another example would be to always declare constants before type definitions. This way you can size an array with named constants.

Here is an example program with a logical ordering for various elements:

```
program Template;   -- program name is always first!

-- Put include (.iri) files here.
#include template.iri

          -- Constants and aliases go here.
          g_csProgName : constant string := "Template Program";
          g_csVersion : constant string := "0.01";
          g_ciArraySize : integer := 100;

          -- User defined type definitions go here.
          type tShape is (Circle, Square, Triangle, Rectangle, Octagon, Pentagon, Dodecahedron);

          type tColor is (Blue, Red, Green, Yellow, Purple);

          type tDescription is
            record
              eColor : tColor;
              eShape : tShape;
            end record;

          type tBigArray is array [g_ciArraySize] of tDescription;


          -- Variable declarations go here.
          g_iBuild : integer;
          g_srcResult : SysCode;
          g_aArray : tBigArray;
          g_rSingleRecord : tDescription;

           -- Start functions and procedures definitions here.

           function MakeVersionString : string;
             sTemp : string;
           begin
             if g_iBuild > 9 then
               sTemp := ("Ver " + g_csVersion + "." + IntegerToString(g_iBuild, 2));
             else
               sTemp := ("Ver " + g_csVersion + ".0" + IntegerToString(g_iBuild, 1));
             end if;

             return sTemp;
            end;

            procedure DisplayVersion;
            begin
              DisplayStatus(g_csProgName + "  " + MakeVersionString);
            end;
```

```
    -- Begin event handler definitions here.
            handler User1KeyPressed;
            begin
              DisplayVersion;
            end;

-- This chunk of code is the system startup event handler.

begin

     -- Initialize all global variables here.
     -- Increment the build number every time you make a change to a new version.
     g_iBuild := 3;

     -- Display the version number to the display.
     DisplayVersion;

end Template;
```

3.3 Declarations

3.3.1 Type Declarations

Type declarations provide the mechanism for specifying the details of enumeration and aggregate types. The identifier representing the type name must be unique within the scope in which the type declaration appears. All user-defined types must be declared prior to being used.

```
     <type-declaration>:
          type IDENTIFIER is <type-definition> ';'
        ;
     <type-definition>:
          <record-type-definition>
        | <array-type-definition>
        | <database-type-definition>
        | <enum-type-definition>
        ;
```

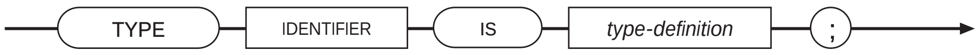

Figure 3-2. Type Declaration Syntax

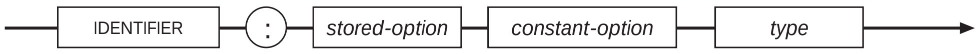

Figure 3-3. Identifier Syntax

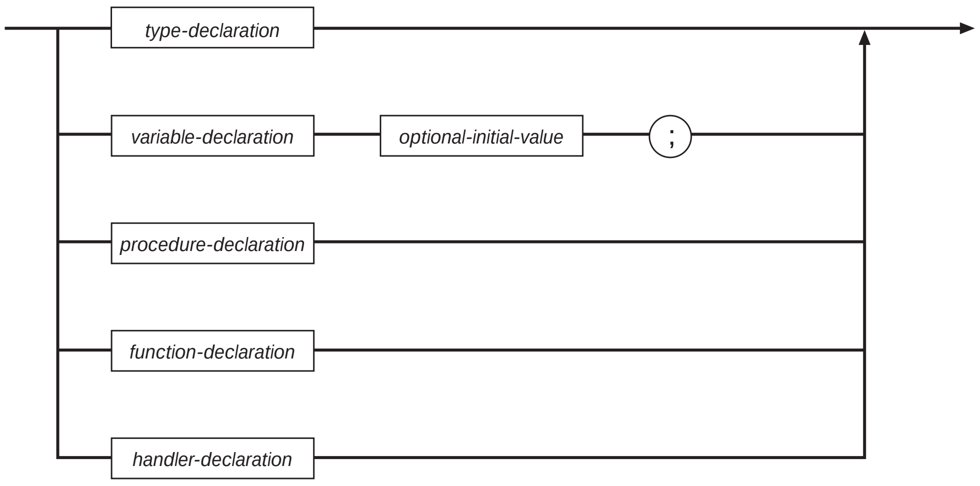

Figure 3-4. Type Declaration Syntax

Enumeration Type Definitions

An enumeration type definition defines a finite ordered set of values. Each value, represented by an identifier, must be unique within the scope in which the type definition appears.

```
<enum-type-definition>:
    '(' <identifier-list> ')'
  ;
<identifier-list>:
    IDENTIFIER
  | <identifier-list> ',' IDENTIFIER
  ;
```

Examples:

```
type StopLightColors is (Green, Yellow, Red);

type BatchStates is (NotStarted, OpenFeedGate, CloseGate, WaitforSS, PrintTicket, AllDone);
```

Record Type Definitions

A record type definition describes the structure and layout of a record type. Each field declaration describes a named component of the record type. Each component name must be unique within the scope of the record; no two components can have the same name. Enumeration, record and array type definitions are not allowed as the type of a component: only previously defined user- or system-defined type names are allowed.

```
<record-type-definition>:
    record
      <field-declaration-list>
    end record
  ;
<field-declaration-list>:
    <field-declaration>
  | <field declaration-list>
    <field declaration>
  ;
<field-declaration>:
      IDENTIFIER ':' <type> ';'
  ;
```

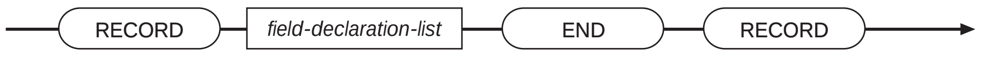

Figure 3-5. Record Type Definition Syntax

Examples:

```
type MyRecord is
  record
    A : integer;
    B : real;
  end record;
```

The EmployeeRecord record type definition, below, incorporates two enumeration type definitions, tDepartment and tEmptype:

```
type tDepartment is (Shipping, Sales, Engineering, Management);

type tEmptype is (Hourly, Salaried);

type EmployeeRecord is
  record
    ID : integer;
    Last : string;
    First : string;
    Dept : tDepartment;
    EmployeeType : tEmptype;
  end record;
```

Database Type Definitions

A database type definition describes a database structure, including an alias used to reference the database.

```
<database-type-definition>:
    database (STRING_CONSTANT)
      <field-declaration-list>
    end database
  ;
<field-declaration-list>:
    <field-declaration>
  | <field declaration-list>
    <field declaration>
  ;
<field-declaration>:
      IDENTIFIER ':' <type> ';'
  ;
```

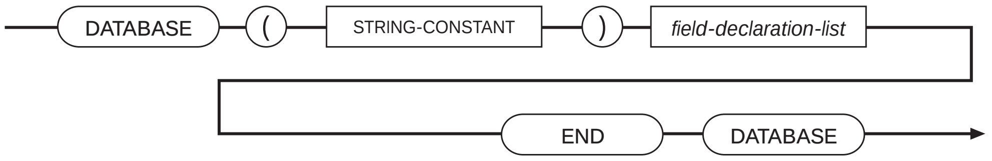

Figure 3-6. Database Type Definition Syntax

Example: A database consisting of two fields, an integer field and a real number, could be defined as follows:

```
type MyDB is
  database ("DBALIAS")
    A : integer
    B : real
  end database;
;
```

Array Type Definitions

An array type definition describes a container for an ordered collection of identically typed objects. The container is organized as an array of one or more dimensions. All dimensions begin at index 1.

```
<array-type-definition>:
    array '[' <expr-list> ']' of <type>
  ;
```

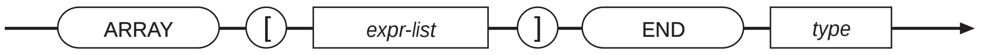

Figure 3-7. Array Type Definition Syntax

Examples:

```
type Weights is array [25] of Real;
```

`An array consisting of user-defined records could be defined as follows:`

```
type Employees is array [100] of EmployeeRecord;
```

A two-dimensional array in which each dimension has an index range of 10 (1…10), for a total of 100 elements could be defined as follows:

```
type MyArray is array [10,10] of Integer;
```

NOTE: *In all of the preceding examples, no variables (objects) are created, no memory is allocated by the type definitions. The type definition only defines a type for use in a later variable declaration, at which time memory is allocated.*

3.3.2 Variable Declarations

A variable declaration creates an object of a particular type. The type specified must be a previously defined user- or system-defined type name. The initial value, if specified, must be type-compatible with the declared object type. All user-defined variables must be declared before being used.

Variables declared with the keyword **stored** cause memory to be allocated in battery-backed RAM. Stored data values are retained even after the indicator is powered down.

Variables declared with the keyword **constant** must have an initial value.

```
    <variable-declaration>:
        IDENTIFIER ':' <stored-option> <constant-option> <type>
        <optional-initial-value>
      ;
    <stored-option>:
        /* NULL */
      | stored
      ;
    <constant-option>:
        /* NULL */
      | constant
      ;
    <optional-initial-value>:
        /* NULL */
      | := <expr>
      ;
```

Example:

```
MyVariable : StopLightColor;
```

3.3.3 Subprogram Declarations

A subprogram declaration defines the formal parameters, return type, local types and variables, and the executable code of a subprogram. Subprograms include handlers, procedures, and functions.

Handler Declarations

A handler declaration defines a subprogram that is to be installed as an event handler. An event handler does not permit parameters or a return type, and can only be invoked by the event dispatching system.

```
    <handler-declaration>:
        handler IDENTIFIER ';'
          <decl-section>
        begin
          <stmt-list>
        end ';'
      ;
```

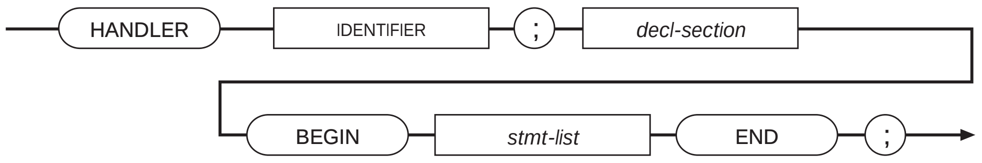

Figure 3-8. Handler Declaration Syntax

Example:

```
    handler SP1Trip;

    I : Integer;

    begin
      for I := 1 to 10
      loop
        Writeln (1, "Setpoint Tripped!");
        if I=2 then
          return;
        endif;
      end loop;
    end;
```

Procedure Declarations

A procedure declaration defines a subprogram that can be invoked by other subprograms. A procedure allows parameters but not a return type. A procedure must be declared before it can be referenced; recursion is not supported.

```
<procedure-declaration>:
        procedure IDENTIFIER
        <optional-formal-args> ';'
        <decl-section>
        begin
        <stmt-list>
        end ';'
    ;
<optional-formal-args>:
        /* NULL */
    | <formal-args>
    ;
<formal-args>:
        '(' <arg-list> ')'
    ;
<arg-list>:
        <optional-var-spec>
        <variable-declaration>
    | <arg-list> ';' <optional-var-spec>
        <variable-declaration>
    ;
<optional-var-spec>:
        /* NULL */
    | var
    ;
```

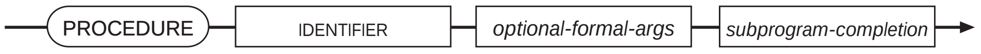

Figure 3-9. Procedure Declaration Syntax

Examples:

```
procedure PrintString (S : String);
begin
  Writeln (1, "The String is => ",S);
end;

procedure ShowVersion;
begin
  DisplayStatus ("Version 1.42");
end;

procedure Inc (var iVariable : Integer);
begin
  iVariable := iVariable + 1;
end;
```

Function Declarations

A function declaration defines a subprogram that can be invoked by other subprograms. A function allows parameters and requires a return type. A function must be declared before it can be referenced; recursion is not supported. A function must return to the point of call using a return-with-value statement.

```
<function-declaration>:
        function IDENTIFIER
        <optional-formal-args> ':' <type> ';'
        <decl-section>
        begin
        <stmt-list>
        end ';'
    ;
```

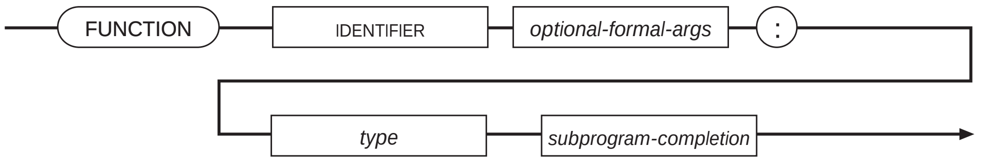

Figure 3-10. Function Declaration Syntax

Examples:

```
function Sum (A : integer; B : integer) : Integer;
begin
  return A + B;
end;

function PoundsPerGallon : Real;
begin
  return 8.34;
end;
```

3.4 Statements

There are only six discrete statements in *iRite*. Some statements, like the **if**, **call**, and assignment (:=) are used extensively even in the simplest program, while the **exit** statement should be used rarely. The **if** and the **loop** statements have variations and can be quite complex. Let's take a closer look at each of the six:

```
<stmt>:
    <assign-stmt>
  | <call-stmt>
  | <if-stmt>
  | <return-stmt>
  | <loop-stmt>
  | exit-stmt>
  ;
```

3.4.1 Assignment Statement

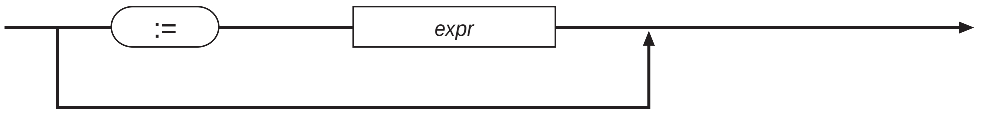

Figure 3-11. Assignment Statement Syntax

The assignment statement uses the assignment operator (:=) to assign the expression on the right-hand side to the object or component on the left-hand side. The types of the left-hand and right-hand sides must be compatible. The value on the left of the ":=" must be a modifiable value. Here are some examples:

Simple assignments:

```
iMaxPieces := 12000;
rRotations := 25.3456;
sPlaceChickenPrompt := "Please place the chicken on the scale…";
```

Assignments in declarations (initialization):

```
iRevision : integer := 1;
rPricePerPound : real := 4.99;
csProgramName : constant string := "Pig and Chicken Weigher";
```

Assignments in **for** loop initialization:

```
for iCounter := 1 to 25
for iTries := ciFirstTry to ciMaxTries
```

Assignment of function return value:

```
sysReturn := GetSPTime(4, dtDateTime);
rCosine := Cos(1.234);
```

Assignment with complex expression on right-hand side:

```
iTotalLivestock := iNumChickens + iNumPigs + GetNumCows;
rTotalCost := ((iNumBolt * rBoltPrice) + (iNumNuts * rNutPrice)) * (1 + rTaxRate);
sOutputText := The total cost is : " + RealToString(rTotalCost, 4, 2) + " dollars.";
```

Assignment of different but compatible types:

```
iValue := 34.867; -- Loss of significant digits! iValue will equal 34, no rounding!
rDegrees := 212; -- No problem! rDegrees will equal 212.000000000000000000
```

3.4.2 Call Statement

The call statement is used to initiate a subprogram invocation. The number and type of any actual parameters are compared against the number and type of the formal parameters that were defined in the subprogram declaration. The number of parameters must match exactly. The types of the actual and formal parameters must also be compatible. Parameter passing is accomplished by copy-in, or by copy-in/copy-out for **var** parameters.

```
<call-stmt>:
     <name> ';'
;
```

Copy-in refers to the way value parameters are copied into their corresponding formal parameters. The default way to pass a parameter in *iRite* is “by value”. By value means that a copy of actual parameter is made to use in the function or procedure. The copy may be changed inside the function or procedure but these changes will never affect the value of the actual parameter outside of the function or procedure, since only the copy may be changed.

The other way to pass a parameter is to use a copy-in/copy-out method. To specify the copy-in/copy-out method, a formal parameter must be preceded by the keyword **var** in the subprogram declaration. **Var** stands for “variable”, which means the parameter may be changed. Just like with a **value** parameter, a copy is made. However, when the function or procedure is done executing, the value of the copy is then copied, or assigned, back into the actual parameter. This is the copy-out part. The result is that if the formal **var** parameter was changed within the subprogram, then the actual parameter will also be changed after the subprogram returns. Actual **var** parameters must be values: a constant cannot be passed as a **var** parameter.

One potentially troublesome issue occurs when passing a global parameter as a **var** parameter. If a global parameter is passed to a function or procedure as a **var** parameter, then the system makes a copy of it to use in the function body. Let’s say that the value of the formal parameter is changed and then some other function or procedure call is made after the change to the formal parameter. If the function or procedure called uses, by name, the same global parameter that was passed into the original function, then the value of the global parameter in the second function will be the value of the global when it was pass into the original function. This is because the changes made to the formal parameter (only a copy of the actual parameter passed in) have not yet been copied-out, since the function or procedure has not returned yet. This is better demonstrated with an example:

```
program GlobalAsVar;

g_ciPrinterPort : constant integer := 2;

g_sString : string := "Initialized, not changed yet";

  procedure PrintGlobalString;
  begin
    WriteLn(g_ciPrinterPort, g_sString);
  end;


  procedure SetGlobalString (var vsStringCopy : string);
  begin

    vsStringCopy := "String has been changed";

    Write(g_ciPrinterPort, "In function call: ");
    PrintGlobalString;

  end;
begin
  Write(g_ciPrinterPort, "Before function call: ");
  PrintGlobalString;

  SetGlobalString(g_sString);

  Write(g_ciPrinterPort, "After function call: ");
  PrintGlobalString;

end GlobalAsVar;
```

When run, the program prints the following:

```
Before function call: Initialized, not changed yet
In function call: Initialized, not changed yet
After function call: String has been changed
```

3.4.3 If Statement

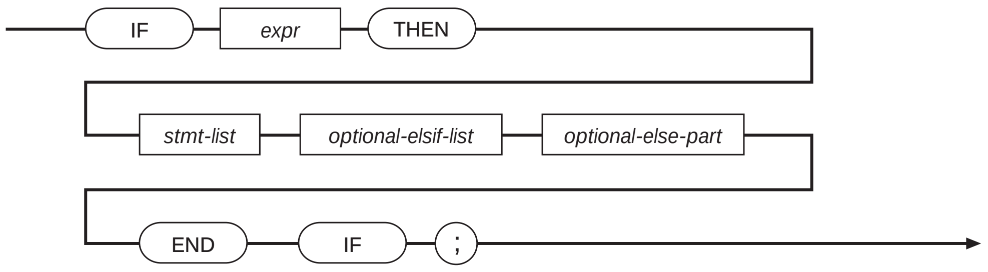

Figure 3-12. If Statement Syntax

The **if** statement is one of the programmer's most useful tools. The **if** statement is used to force the program to execute different paths based on a decision. In its simplest form, the **if** statement looks like this:

```
if <expression> then
  <statement list>
end if;
```

The decision is made after evaluating the expression. The expression is most often a "conditional expression". If the expression evaluates to true, then the statements in <*statement list*> are executed. This form of the **if** statement is used primarily when you only want to do something if a certain condition is true. Here is an example:

```
if iStrikes = 3 then
  sResponse := "You're out!";
end if;
```

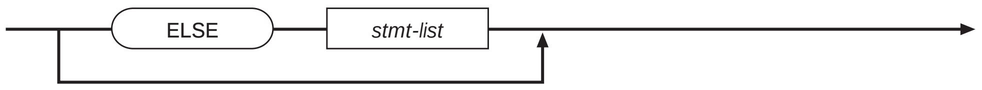

Figure 3-13. Optional Else Statement Syntax

Another form of the **if** statement, known as the **if-else** statement has the general form:

```
if <expression> then
  <statement list 1>
else
  <statement list 2>
end if;
```

The **if-else** is used when the program must decide which of exactly two different paths of execution must be executed. The path that will execute the statement or statements in <*statement list 1*> will be chosen if <*expression*> evaluates to true. Here is an example:

```
if iAge => 18 then
  sStatus := "Adult";
else
  sStatus := "Minor";
  end if;
```

If the statement is false, then the statement or statements in <*statement list 2*> will be executed. Once the expression is evaluated and one of the paths is chosen, the expression is not evaluated again. This means the statement will terminate after one of the paths has been executed.

For example, if the expression was true and we were executing <*statement list 1*>, and within the code in <*statement list 1*> we change some part of <*expression*> so it would at that moment evaluate to false, <*statement list 2*> would still not be executed. This point is more relevant in the next form called the **if-elsif**.

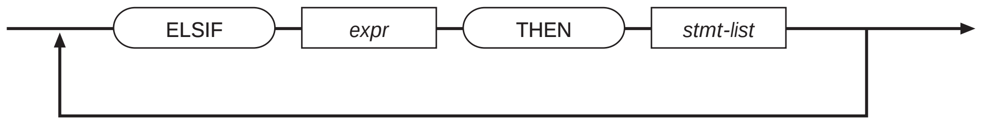

Figure 3-14. Optional Else-If Statement Syntax

The **if-elsif** version is used when a multi-way decision is necessary and has this general form:

```
if <expression> then
  <statement list 1>
elsif <expression> then
  <statement list 2>
elsif <expression> then
  <statement list 3>
elsif <expression> then
  <statement list 4>
else
  <statement list 5>
end if;
```

Here is an example of the **if-elsif** form:

```
if rWeight <= 2.0 then
  iGrade := 1;
elsif (rWeight > 2.0) and (rWeight < 4.5) then
  iGrade := 2;
elsif (rWeight > 4.5) and (rWeight < 9.25) then
  iGrade := 3;
elsif (rWeight > 9.25) and (rWeight < 11.875) then
  iGrade := 4;
else
  iGrade := 0;
  sErrorString := "Invalid Weight!";
end if;
```

3.4.4 Loop Statement

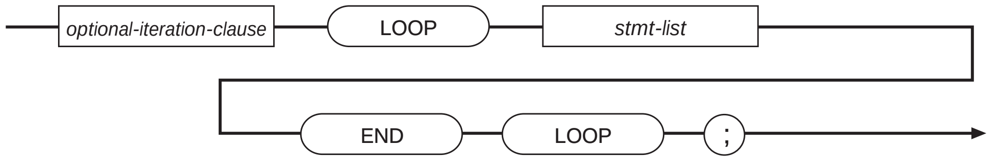

Figure 3-15. Loop Statement Syntax

The **loop** statement is also quite important in programming. The **loop** statement is used to execute a statement list 0 or more times. An optional expression is evaluated and the statement list is executed. The expression is then re-evaluated and as long as the expression is true the statements will continue to get executed. The **loop** statement in *iRite* has three general forms. One way is to write a loop with no conditional expression. The loop will keep executing the loop body (the statement list) until the **exit** statement is encountered. The **exit** statement can be used in any **loop**, but is most often used in this version without a conditional expression to evaluate. It has this form:

```
loop
<statement list>
end loop;
```

This version is most often used with an **if** statement at the end of the statement list. This way the statement list will always execute at least once. This is referred to as a **loop-until**. Here is an example:

```
rGrossWeight : real;

loop
  WriteLn(2, "I'm in a loop.");
  GetGross(1, Primary, rGrossWeight);
  if rGrossWeight > 200 then
    exit;
  end if;
end loop;
```

A similar version uses an optional **while** clause at the start of the loop. The **while-loop** version is used when you want the loop to execute zero or more times. Since the expression is evaluated before the loop is entered, the statement list may not get executed even once. Here is the general form for the **while-loop** statement:

```
while <expression>
loop
  <statement list>
end loop;
```

Here is the same example from above, but with a **while** clause. Keep in mind that if the gross weight is greater than 200 pounds, then the loop body will never execute:

```
rGrossWeight : real;

GetGross(1, Primary, rGrossWeight);

while rGrossWeight <= 200
loop
    WriteLn(2, "I'm in a loop.");
    GetGross(1, Primary, rGrossWeight);
end loop;
```

Here we see that we had to get the weight before we could evaluate the expression. In addition we have to get the weight in the loop. In this example, it would be better programming to use the **loop-until** version.

Another version is known as the **for-loop**. The **for-loop** is best used when you want to execute a chunk of code for a known or predetermined number of times. In its general form the **for-loop** looks like this:

```
for <name> := <expression> to <expression> step <expression>
loop
  <statement list>
end loop;
```

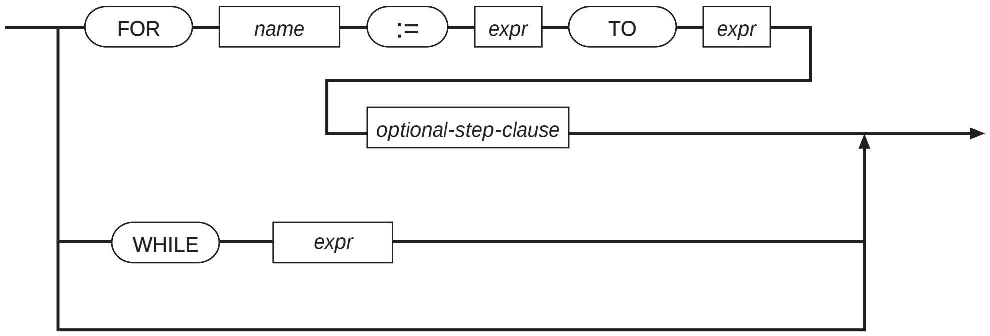

Figure 3-16. Optional Loop Iteration Clause Syntax

The optional step clause can be omitted if you want <*name*> to increment by 1 after each run of the statement list. If you want to increment <*name*> by 2 or 3, or decrement it by 1 or 2, then you have to use the step clause. The step expression (–1 in the second example below) must be a constant.

```
for iCount := 97 to 122
loop
    strAlpha := strAlpha + chr$(iCount);
end loop;

for iCount := 10 to 0 step -1
loop
    if iCount = 0 then
      strMissionControl := "Blast off!";
    else
      strMissionControl := IntegerToString(iCount, 2);
    end if;
end loop;
```

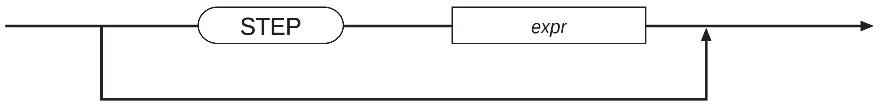

Figure 3-17. Optional Step Clause Syntax

3.4.5 Return Statement

The **return** statement can only be used inside of subprograms (functions, procedures, and event handlers). The return statement in procedures and handlers cannot return a value. An explicit return statement inside a procedure or handler is not required since the compiler will insert one if the **return** statement is missing. If you want to return from a procedure or handler before the code body is done executing, then you can use the **return** statement to exit at that point.

```
procedure DontDoMuch;
begin
if PromptUser("circle: ") <> SysOK then
    return;
  end if;
end;
```

Functions must return a value and an explicit **return** statement is required. The data type of the expression returned must be compatible with the return type specified in the function declaration.

```
function Inc(var viNumber : integer) : integer;
begin
  viNumber := viNumber + 1;
  return viNumber;
end;
```

It is permissible to have more then one **return** statement in a subprogram, but not recommended. In most instances it is better programming practice to use conditional execution (using the **if** statement) with one **return** statement at the end of the function than it is to use a **return** statement multiple times. **Return** statements liberally dispersed through a subprogram body can result in “dead code” (code that never gets executed) and hard-to-find bugs.

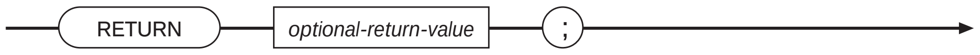

Figure 3-18. Return Statement Syntax

3.4.6 Exit Statement

The **exit** statement is only allowed in loops. It is used to immediately exit any loop (loop-until, for-loop, while-loop) it is called from. Sometimes it is convenient to be able to exit from a loop instead of testing at the top. In the case of nested loops (a loop inside another loop), only the innermost enclosing loop will be exited. See the loop examples in Section 3.4.4 on page 23 for the **exit** statement in action.

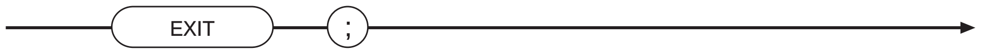

Figure 3-19. Exit Statement Syntax

4.0 Built-in Types and Functions

This section describes the built-in types and functions provided for programming the *920i* indicator. Functions are grouped according to the kinds of operations they support.

4.1 Built-in Types

The following built-in types are used in parameters passed to and from the functions described in this section. Most built-in types are declared in the system.src file found in the *iRev* application directory. Some built-in types are defined by the compiler and are not declared in the system.src file.

```
type SysCode is (SysOK,
   SysLFTViolation,
   SysOutOfRange,
   SysPermissionDenied,
   SysInvalidScale,
   SysBatchRunning,
   SysBatchNotRunning,
   SysNoTare,
   SysInvalidPort,
   SysQFull,
   SysInvalidUnits,
   SysInvalidSetpoint,
   SysInvalidRequest,
   SysInvalidMode,
   SysRequestFailed,
   SysInvalidKey,
   SysInvalidWidget,
   SysInvalidState,
   SysInvalidTimer,
   SysNoSuchDatabase,
   SysNoSuchRecord,
   SysDatabaseFull,
   SysNoSuchColumn,
   SysInvalidCounter,
   SysDeviceError);
type Mode is (GrossMode, NetMode);
type Units is (Primary, Secondary, Tertiary);
type TareType is (NoTare, PushButton, Keyed);
type BatchingMode is (Off, Manual, Auto);
type BatchStatus is (BatchComplete, BatchStopped, BatchRunning, BatchPaused);
type PrintFormat is (GrossFmt, NetFmt,
   AuxFmt,
   TrWInFmt, TrRegFmt, TrWoutFmt,
   SPFmt,
   AccumFmt);
type TimerMode is (TimerOneShot, TimerContinuous);
type Keys is (Soft4Key, Soft5Key, GrossNetKey, UnitsKey,
   Soft3Key, Soft2Key, Soft1Key, ZeroKey,
   Undefined3Key, Undefined4Key, TareKey, PrintKey,
   N1Key, N4Key, N7Key, DecpntKey,
   NavUpKey, NavLeftKey, EnterKey, Undefined5Key,
   N2Key, N5Key, N8Key, N0Key,
   Undefined1Key, Undefined2Key, NavRightKey, NavDownKey,
   N3Key, N6Key, N9Key, ClearKey);
type DT Component is (DateTimeYear,
   DateTimeMonth,
   DateTimeDay,
   DateTimeHour,
   DateTimeMinute,
   DateTimeSecond);
type OnOffType (Voff,Von);
```

4.1.1 Using SysCode Data

SysCode data can be used to take some action based on whether or not a function completed successfully. For example, the following code checks the SysCode result following a `GetTare` function. If the function completed successfully, the retrieved tare weight is written to Port 1:

```
Scale1 : constant Integer := 1;
Port1 : constant Integer := 1;
SysResult : SysCode;
TareWeight : Real;
…
SysResult:= GetTare (Scale1, Primary, TareWeight);
if SysResult = SysOK then
  WriteLn (Port1, "The current tare weight is ", TareWeight)'
end if;
```

4.2 Scale Data Acquisition

NOTE: Unless otherwise stated, when an API with a VAR parameter returns a SysCode value other than *SysOK*, the VAR parameter is not changed.

4.2.1 Weight Acquisition

See the GetStableWeight function on page 72 for an example of using the GetGross, GetNet, and GetTare functions in a program.

GetGross

Sets W to the current gross weight value of scale S, in the units specified by U. ***W*** will contain a weight value even if the scale is in programmed overload.

Syntax:

```
function GetGross (S : Integer; U : Units; VAR W : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysInvalidRequest	The requested value is not available.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
GrossWeight : Real;
…
GetGross (Scale1, Primary, GrossWeight);
WriteLn (Port1, "Current gross weight is", GrossWeight);
```

GetNet

Sets ***W*** to the current net weight value of scale S, in the units specified by U. ***W*** will contain a weight value even if the scale is in programmed overload.

Syntax:

```
function GetNet (S : Integer; U : Units; VAR W : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysInvalidRequest	The requested value is not available.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
NetWeight : Real;
…
GetNet (Scale2, Secondary, NetWeight);
WriteLn (Port1, "Current net weight is", NetWeight);
```

GetTare
Sets ***W*** to the tare weight of scale ***S*** in weight units specified by ***U***.

```
function GetTare (S : Integer; U : Units; VAR W : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysInvalidRequest	The requested value is not available.
SysNoTare	The specified scale has no tare. ***W*** is set to 0.0.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:
```
TareWeight : Real;
…
GetTare (Scale3, Tertiary, TareWeight);
WriteLn (Port1, "Current tare weight is ", TareWeight);
```

4.2.2 Tare Manipulation

AcquireTare
Acquires a pushbutton tare from scale ***S***.

Syntax:
```
function AcquireTare (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist
SysLFTViolation	The tare operation would violate configured legal-for-trade restrictions for the specified scale. No tare is acquired.
SysOutOfRange	The tare operation would acquire a tare that may cause a display overload. No tare is acquired.
SysPermissionDenied	The tare operation would violate configured tare acquisition restrictions for the specified scale. No tare is acquired.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:
```
AcquireTare (Scale1);
```

SetTare
Sets the tare weight for the specified channel.

Syntax:
```
function SetTare (S : Integer; U : Units; W : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysLFTViolation	The tare operation would violate configured legal-for-trade restrictions for the specified scale. No tare is acquired.
SysOutOfRange	The tare operation would acquire a tare that may cause a display overload. No tare is acquired.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:
```
DesiredTare : Real;
…
DesiredTare := 1234.5;
SetTare (Scale1, Primary, DesiredTare);
```

GetTareType

Sets T to indicate the type of tare currently on scale S.

Syntax:

```
function GetTareType (S : Integer; VAR T : TareType) : SysCode;
```

TareType values returned:

NoTare	There is no tare value associated with the specified scale.
PushbuttonTare	The current tare was acquired by pushbutton.
Keyed	The current tare was acquired by key entry or by setting the tare.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist. ***T*** is unchanged.
SysOK	The function completed successfully.

ClearTare

Removes the tare associated with scale ***S*** and sets the tare type associated with the scale to ***NoTare*** .

Syntax:

```
function ClearTare (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysNoTare	The scale specified by ***S*** has no tare.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
ClearTare (Scale1);
```

4.2.3 Rate of Change

GetROC

Sets ***R*** to the current rate-of-change value of scale S. Syntax:

```
function GetROC (S : Integer; VAR R : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
ROC : Real;
…
GetROC (Scale3, ROC);
WriteLn (Port1, "Current ROC is", ROC);
```

4.2.4 Accumulator Operations

GetAccum

Sets ***W*** to the value of the accumulator associated with scale ***S***, in the units specified by ***U***.

Syntax:

```
function GetAccum (S : Integer; U : Units; VAR W ; Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysDeviceError	The scale is reporting an error condition.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysOK	The function completed successfully.

Example:

```
AccumValue : Real;
…
GetAccum (Scale1, AccumValue);
```

SetAccum

Sets the value of the accumulator associated with scale ***S*** to weight ***W***, in units specified by ***U***.

Syntax:

```
function SetAccum (S : Integer; U : Units; W : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysDeviceError	The scale is reporting an error condition.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysOK	The function completed successfully.

Example:

```
AccumValue : Real;
…
AccumValue := 110.5
SetAccum (Scale1, Primary, AccumValue);
```

GetAvgAccum

Sets ***W*** to the average accumulator value associated with scale ***S***, in the units specified by ***U***, since the accumulator was last cleared.

Syntax:

```
function GetAvgAccum (S : Integer; U : Units; VAR W ; Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not valid.
SysDeviceError	The scale is reporting an error condition.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysOK	The function completed successfully.

Example:

```
AvgAccum : Real;
…
GetAvgAccum (Scale1, AvgAccum);
```

GetAccumCount

Sets ***N*** to the number of accumulations performed for scale ***S*** since its accumulator was last cleared.

Syntax:

```
function GetAccumCount (S : Integer; VAR N ; Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
NumAccums : Integer;
…
GetAccumCount (Scale1, NumAccums);
```

GetAccumDate

Sets ***D*** to the date of the most recent accumulation performed by scale ***S***.

Syntax:

```
function GetAccumDate (S : Integer; VAR D ; String) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
AccumDate : String;
…
GetAccumDate (Scale1, AccumDate);
```

GetAccumTime

Sets ***T*** to the time of the most recent accumulation performed by scale ***S***.

Syntax:

```
function GetAccumTime (S : Integer; VAR T ; String) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
AccumTime : String;
…
GetAccumTime (Scale1, AccumTime);
```

ClearAccum

Sets the value of the accumulator for scale ***S*** to zero.

Syntax:

```
function ClearAccum (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysPermissionDenied	The accumulator is not enabled for the specified scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
ClearAccum (Scale1);
```

4.2.5 Scale Operations

CurrentScale

Sets ***S*** to the numeric ID of the currently displayed scale.

Syntax:

```
function CurrentScale : Integer;
```

Example:

```
ScaleNumber : Integer;
…
ScaleNumber=CurrentScale;
```

SelectScale

Sets scale ***S*** as the current scale.

Syntax:

```
function SelectScale (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist. The current scale is not changed
SysOK	The function completed successfully.

Example:

```
SelectScale (Scale1);
```

GetMode

Sets ***M*** to the value representing the current display mode for scale ***S***.

Syntax:

```
function GetMode (S : Integer; VAR M : Mode) : SysCode;
```

Mode values returned:

GrossMode	Scale ***S*** is currently in gross mode.
NetMode	Scale ***S*** is currently in net mode.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
CurrentMode : Mode;
…
GetMode (Scale1, CurrentMode);
```

Example:

```
TT : TareType;
…
GetTareType (Scale1, TT);
if TT=KeyedTare then …
```

GetUnits

Sets ***U*** to the value representing the current display units for scale ***S***.

Syntax:

```
function GetUnits (S : Integer; VAR U : Units) : SysCode;
```

Units values returned:

Primary	Primary units are currently displayed on scale ***S***.
Secondary	Secondary units are currently displayed on scale ***S***.
Tertiary	Tertiary units are currently displayed on scale ***S***.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
CurrentUnits : Units;
…
GetUnits (Scale1, CurrentUnits);
```

GetUnitsString

Sets ***V*** to the text string representing the current display units for scale ***S***.

Syntax:

```
function GetUnitsString (S : Integer; U : Units; VAR V : String) : SysCode;
```

Units values sent:

Primary	Primary units are currently displayed on scale ***S***.
Secondary	Secondary units are currently displayed on scale ***S***.
Tertiary	Tertiary units are currently displayed on scale ***S***.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units value specified by ***U*** does not exist.
SysOK	The function completed successfully.

Example:

```
CurrentUnitsString : Units;
…
GetUnitsString (Scale1, Primary, CurrentUnitsString);
```

InCOZ

Sets ***V*** to a non-zero value if scale ***S*** is within 0.25 grads of gross zero. If the condition is not met, ***V*** is set to zero.

Syntax:

```
function InCOZ (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully

Example:

```
ScaleAtCOZ : Integer;
…
InCOZ (Scale1, ScaleAtCOZ);
```

InMotion
Sets ***V*** to a non-zero value if scale ***S*** is in motion. Otherwise, ***V*** is set to zero.

Syntax:
```
function InMotion (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully

Example:
```
ScaleInMotion : Integer;
…
InMotion (Scale1, ScaleInMotion);
```
See the GetStableWeight function on page 72 for an example of using the InMotion function in a program.

InRange
Sets ***V*** to zero value if scale ***S*** is in an overload or underload condition. Otherwise, ***V*** is set to a non-zero value.

Syntax:
```
function InRange (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully

Example:
```
ScaleInRange : Integer;
…
InRange (Scale1, ScaleInRange);
```

SetMode
Sets the current display mode on scale ***S*** to ***M***.

Syntax:
```
function SetMode (S : Integer; M : Mode) : SysCode;
```

Mode values sent:

GrossMode	Scale ***S*** is set to gross mode.
NetMode	Scale ***S*** is set to net mode.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidMode	The mode value ***M*** is not valid.
SysDeviceError	The scale is reporting an error condition. ***M*** is not changed.
SysOK	The function completed successfully.

Example:
```
SetMode (Scale1, Gross);
```

SetUnits
Sets the current display units on scale ***S*** to ***U***.

Syntax:
```
function SetUnits (S : Integer; U : Units) : SysCode;
```

Units values sent:

Primary	Primary units will be displayed on scale ***S***.
Secondary	Secondary units will be displayed on scale ***S***.
Tertiary	Tertiary units will be displayed on scale ***S***.

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units value ***U*** is not valid.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:
```
SetUnits (Scale1, Secondary);
```

ZeroScale

Performs a gross zero scale operation for ***S***.

Syntax:

```
function ZeroScale (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist
SysLFTViolation	The zero operation would violate configured legal-for-trade restrictions for the specified scale. No zero is performed.
SysOutOfRange	The zero operation would exceed the configured zeroing limit. No zero is acquired.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
ZeroScale (Scale1);
```

4.2.6 A/D and Calibration Data

GetFilteredCount

Sets ***C*** to the current filtered A/D count for scale ***S***.

Syntax:

```
function GetFilteredCount (S : Integer; VAR C : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
FilterCount : Integer;
…
GetFilteredCount (1; FilterCount);
```

GetRawCount

Sets ***C*** to the current raw A/D count for scale ***S***.

Syntax:

```
function GetRawCount (S : Integer; VAR C : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

Example:

```
RawCount : Integer;
…
GetRawCount (1; RawCount);
```

GetLCCD

Sets ***V*** to the calibrated deadload count for scale ***S***.

Syntax:

```
function GetLCCD (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysOK	The function completed successfully.

GetLCCW

Sets ***V*** to the calibrated span count for scale ***S***.

Syntax:
```
function GetLCCW (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysOK	The function completed successfully.

GetWVal

Sets ***V*** to the configured WVAL (test weight value) for scale ***S***.

Syntax:
```
function GetWVal (S : Integer; VAR V : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysOK	The function completed successfully.

GetZeroCount

Sets ***V*** to the acquired zero count for scale ***S***.

Syntax:
```
function GetWVal (S : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** is not an A/D-based scale.
SysOK	The function completed successfully.

4.3 System Support

Date$

Returns a string representing the system date contained in ***DT***.

Syntax:
```
function Date$ (DT : DateTime) : String;
```

DisplayIsSuspended

Returns a true (non-zero) value if the display is suspended (using the SuspendDisplay procedure), or a false (zero) value if the display is not suspended.

Syntax:
```
function DisplayIsSuspended : Integer;
```

EventChar

Returns a one-character string representing the character received on a communications port that caused the *PortxCharReceived* event. If EventChar is called outside the scope of a *PortxCharReceived* event, EventChar returns a string of length zero. See Section 6.1 on page 90 for information about the *PortxCharReceived* event handler.

Syntax:
```
function EventChar : String;
```

Example:
```
handler Port4CharReceived;
  strOneChar : string;
begin
  strOneChar := EventChar;
end;
```

EventKey

Returns an enumeration of type Keys with the value corresponding to the key press that generated the event. See Section 4.1 on page 26 for a definition of the Keys data type.

Syntax:
```
function EventKey : Keys;
```

Example:
```
handler KeyPressed;
```

```
begin
  if EventKey = ClearKey then
  …
  end if;
end;
```

EventPort

Returns the communications port number that received an F#*x* serial command. This function extracts data from the Cmd*x*Handler event for the F#*x* command, if enabled. (The Cmd*x*Handler, if enabled, runs whenever a F#*x* command is received on any serial port.) If the Cmd*x*Handler is not enabled, this function returns 0 as the port number. See the Cmd1Handler definition in the HelloWorld program on page 59 for an example of the EventPort function.

Syntax:

```
function EventPort : Integer;
```

EventString

Returns the string sent with an F#*x* serial command.This function extracts data from the Cmd*x*Handler event for the F#*x* command, if enabled. (The Cmd*x*Handler, if enabled, runs whenever a F#*x* command is received on any serial port.) If the Cmd*x*Handler is not enabled, or if no string is defined for the F#*x* command, this function returns a string of length zero. See the Cmd1Handler definition in the HelloWorld program on page 59 for an example of the EventString function.

Syntax:

```
function EventString : String;
```

GetConsecNum

Returns the value of the consecutive number counter.

Syntax:

```
function GetConsecNum : Integer;
```

GetCountBy

Sets ***C*** to the real count-by value on scale ***S***, in units ***U***.

Syntax:

```
function GetCountBy (S : Integer; U : Units; VAR C : Real) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidUnits	The units specified by ***U*** is not recognized.
SysInvalidRequest	The scale specified by ***S*** does not support this operation (serial scale).
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

GetDate

Extracts date information from ***DT*** and places the data in variables ***Year*** , ***Month*** , and ***Day*** .

Syntax:

```
procedure GetDate (DT : DateTime; VAR Year : Integer; VAR Month : Integer;
VAR Day : Integer);
```

GetGrads

Sets ***G*** to the configured grad value of scale ***S***.

Syntax:

```
function GetGrads (S : Integer; VAR G : Integer) : SysCode;
```

SysCode values returned:

SysInvalidScale	The scale specified by ***S*** does not exist.
SysInvalidRequest	The scale specified by ***S*** does not support this operation (serial scale).
SysDeviceError	The scale is reporting an error condition.
SysOK	The function completed successfully.

GetSoftwareVersion

Returns the current software version.

Syntax:

```
function GetSoftwareVersion : String;
```

GetTime
Extracts time information from ***DT*** and places the data in variables ***Hour*** , ***Minute*** , and ***Second*** .

Syntax:
```
procedure GetTime (DT : DateTime; VAR Hour : Integer; VAR Minute : Integer;
VAR Second : Integer);
```

GetUID
Returns the current unit identifier.

Syntax:
```
function GetUID : Integer;
```

LockKey
Disables the specified front panel key. Possible values are: ZeroKey, GrossNetKey, TareKey, UnitsKey, PrintKey, Soft1Key, Soft2Key, Soft3Key, Soft4Key, Soft5Key, NavUpKey, NavRightKey, NavDownKey, NavLeftKey, EnterKey, N1Key, N2Key, N3Key, N4Key, N5Key, N6Key, N7Key, N8Key, N9Key, N0Key, DecpntKey, ClearKey.

Syntax:
```
function LockKey (K : Keys) : SysCode;
```

SysCode values returned:

SysInvalidKey	The key specified is not valid.
SysOK	The function completed successfully.

LockKeypad
Disables operation of the entire front panel keypad.

Syntax:
```
function LockKeypad : SysCode;
```

SysCode values returned:

SysPermissionDenied	
SysOK	The function completed successfully.

ResumeDisplay
Resumes a suspended display.

Syntax:
```
procedure ResumeDisplay
```

SetConsecNum
Sets ***V*** to the value of the consecutive number counter.

Syntax:
```
function SetConsecNum (V : Integer) : SysCode;
```

SysCode values returned:

SysOutOfRange	The value specified is not in the allowed range. The consecutive number is not changed.
SysOK	The function completed successfully.

SetDate
Sets the date in ***DT*** to the values specified by ***Year*** , ***Month*** , and ***Day*** .

Syntax:
```
function SetDate (VAR DT : DateTime; VAR Year : Integer; VAR Month : Integer;
VAR Day : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	Year, month, or day entry not valid.
SysOK	The function completed successfully.

SetSoftkeyText
Sets the text of softkey ***K*** (representing F1–F10) to the text specified by ***S*** .

Syntax:
```
function SetSoftkeyText (K : Integer; S : String) : SysCode;
```

SysCode values returned:

SysInvalidRequest	The value specified for K is less than 1 or greater than 10, or does not represent a configured softkey.
SysOK	The function completed successfully.

SetTime
Sets the time in ***DT*** to the values specified by ***Hour*** , ***Minute*** , and ***Second*** .

Syntax:
```
function SetTime (VAR DT : DateTime; VAR Hour : Integer; VAR Minute :
Integer; VAR Second : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	Hour or minute entry not valid.
SysOK	The function completed successfully.

SetUID
Sets the unit identifier.

Syntax:
```
function SetUID (newid : Integer) : SysCode;
```

SysCode values returned:

SysOutOfRange	The unit identifier specified for ***newid*** is not in the allowed range. The UID is not changed.
SysOK	The function completed successfully.

STick
Returns the number of system ticks, in 1/1200th of a second intervals, since the indicator was powered on (1200 = 1 second).

Syntax:
```
function STick;
```

SuspendDisplay
Suspends the display.

Syntax:
```
procedure SuspendDisplay
```

SystemTime
Returns the current system date and time.

Syntax:
```
function SystemTime : DateTime;
```

Time$
Returns a string representing the system time contained in ***DT***.

Syntax:
```
function Time$ (DT : DateTime) : String;
```

UnlockKey
Enables the specified front panel key. Possible values are: ZeroKey, GrossNetKey, TareKey, UnitsKey, PrintKey, Soft1Key, Soft2Key, Soft3Key, Soft4Key, Soft5Key, NavUpKey, NavRightKey, NavDownKey, NavLeftKey, EnterKey, N1Key, N2Key, N3Key, N4Key, N5Key, N6Key, N7Key, N8Key, N9Key, N0Key, DecpntKey, ClearKey.

Syntax:
```
function UnlockKey (K : Keys) : SysCode;
```

SysCode values returned:

SysInvalidKey	The key specified is not valid.
SysOK	The function completed successfully.

UnlockKeypad
Enables operation of the entire front panel keypad.

Syntax:
```
function UnlockKeypad : SysCode;
```

SysCode values returned:

SysPermissionDenied	
SysOK	The function completed successfully.

4.4 Setpoints and Batching

NOTE: Unless otherwise stated, when an API with a VAR parameter returns a SysCode value other than *SysOK*, the VAR parameter is not changed.

DisableSP

Disables operation of setpoint ***SP***.

Syntax:

```
function DisableSP (SP : Integer) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysBatchRunning	Setpoint ***SP*** cannot be disabled while a batch is running.
SysInvalidRequest	The setpoint specified by ***SP*** cannot be enabled or disabled.
SysOK	The function completed successfully.

Example:

```
DisableSP (4);
```

EnableSP

Enables operation of setpoint ***SP***.

Syntax:

```
function EnableSP (SP : Integer) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysBatchRunning	Setpoint ***SP*** cannot be enabled while a batch is running.
SysInvalidRequest	The setpoint specified by ***SP*** cannot be enabled or disabled.
SysOK	The function completed successfully.

Example:

```
EnableSP (4);
```

GetBatchingMode

Returns the current batching mode (BATCHNG parameter).

Syntax:

```
function GetBatchingMode : BatchingMode;
```

BatchingMode values returned:

Off	Batching mode is off.
Auto	Batching mode is set to automatic.
Manual	Batching mode is set to manual.

GetBatchStatus

Sets ***S*** to the current batch status.

Syntax:

```
function GetBatchStatus (VAR S : BatchStatus) : SysCode;
```

BatchStatus values returned:

BatchComplete	The batch is complete.
BatchStopped	The batch is stopped.
BatchRunning	A batch routine is in progress.
BatchPaused	The batch is paused.

SysCode values returned:

SysInvalidRequest	The BATCHNG configuration parameter is set to OFF.
SysOK	The function completed successfully.

GetCurrentSP

Sets ***SP*** to the number of the current setpoint.

Syntax:

```
function GetCurrentSP (VAR SP : Integer) : Syscode;
```

SysCode values returned:

SysInvalidRequest	The BATCHNG configuration parameter is set to OFF.
SysBatchNotRunning	No batch routine is running.
SysOK	The function completed successfully.

Example:

```
CurrentSP : Integer;
…
GetCurrentSP (CurrentSP);
WriteLn (Port1, "Current setpoint is", CurrentSP);
```

GetSPBand

Sets ***V*** to the current band value (BANDVAL parameter) of the setpoint ***SP***.

Syntax:

```
function GetSPBand (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no hysteresis (BANDVAL) parameter.
SysOK	The function completed successfully.

Example:

```
SP7Bandval : Real;
…
GetSPBand (7, SP7BAndval);
WriteLn (Port1, "Current Band Value of SP7 is", SP7Bandval);
```

GetSPCaptured

Sets ***V*** to the weight value that satisfied the setpoint ***SP***.

Syntax:

```
function GetSPCaptured (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint number specified by ***SP*** is less than 1 or greater than 100.
SysInvalidRequest	The setpoint has no captured value.
SysOK	The function completed successfully.

GetSPHyster

Sets ***V*** to the current hysteresis value (HYSTER parameter) of the setpoint ***SP***.

Syntax:

```
function GetSPHyster (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no hysteresis HYSTER) parameter.
SysOK	The function completed successfully.

Example:

```
SP5Hyster : Real;
…
GetSPHyster (5, SP5Hyster);
WriteLn (Port1, "Current Hysteresis Value of SP5 is", SP5Hyster);
```

GetSPPreact

Sets ***V*** to the current preact value (PREACT parameter) of the setpoint ***SP***.

Syntax:

```
function GetSPPreact (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no preact (PREACT) parameter.
SysOK	The function completed successfully.

Example:

```
SP2Preval : Real;
…
GetSPPreact (2, SP2Preval);
WriteLn (Port1, "Current Preact Value of SP2 is", SP2Preval);
```

GetSPValue

Sets ***V*** to the current value (VALUE parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPValue (SP : Integer; VAR V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VALUE parameter.
SysOK	The function completed successfully.

Example:
```
SP4Val : Real;
…
GetSPValue (4, SP4Val);
WriteLn (Port1, "Current Value of SP4 is", SP4Val);
```

GetSPNSample

For averaging (AVG) setpoints, sets ***N*** to the current number of samples (NSAMPLE parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPNSample (SP : Integer; VAR N : Integer) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no NSAMPLE parameter.
SysOK	The function completed successfully.

Example:
```
SP5NS : Integer;
…
GetSPNSample (5, SP5NS);
WriteLn (Port1, "Current NSample Value of SP5 is", SP5NS);
```

GetSPTime

For time of day (TOD) setpoints, sets ***DT*** to the current trip time (TIME parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPTime (SP : Integer; VAR DT : DateTime) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no TIME parameter.
SysOK	The function completed successfully.

Example:
```
SP2TIME : DateTime;
…
GetSPTime (2, SP2TIME);
WriteLn (Port1, "Current Trip Time of SP2 is", SP2TIME);
```

GetSPDuration

For time of day (TOD) setpoints, sets ***DT*** to the current trip duration (DURATION parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPDuration (SP : Integer; VAR DT : DateTime) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no DURATION parameter.
SysOK	The function completed successfully.

Example:
```
SP3DUR : DateTime;
…
GetSPTime (3, SP3DUR);
WriteLn (Port1, "Current Trip Duration of SP3 is", SP3DUR);
```

GetSPVover
For checkweigh (CHKWEI) setpoints, sets ***V*** to the current overrange value (VOVER parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPVover (SP : Integer; VAR V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VOVER parameter.
SysOK	The function completed successfully.

Example:
```
SP3VOR : Real;
…
GetSPVover (3, SP3VOR);
WriteLn (Port1, "Current Overrange Value of SP3 is", SP3VOR);
```

GetSPVunder
For checkweigh (CHKWEI) setpoints, sets ***V*** to the current overrange value (VUNDER parameter) of the setpoint ***SP***.

Syntax:
```
function GetSPVunder (SP : Integer; VAR V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VUNDER parameter.
SysOK	The function completed successfully.

Example:
```
SP4VUR : Real;
…
GetSPVunder (4, SP4VUR);
WriteLn (Port1, "Current Underrange Value of SP4 is", SP4VUR);
```

PauseBatch
Pauses a currently running batch.

Syntax:
```
function PauseBatch : SysCode;
```

SysCode values returned:

SysBatchRunning	No batch routine is running.
SysOK	The function completed successfully.

ResetBatch
Resets a currently running batch.

Syntax:
```
function ResetBatch : SysCode;
```

SysCode values returned:

SysBatchRunning	No batch routine is running.
SysOK	The function completed successfully.

SetBatchingMode
Sets the batching mode (BATCHNG parameter) to the value specified by ***M***.

BatchingMode values sent:

Off	Batching mode is off.
Auto	Batching mode is set to automatic.
Manual	Batching mode is set to manual.

Syntax:
```
function SetBatchingMode (M : BatchingMode) : SysCode;
```

SysCode values returned:

SysInvalidMode	The batching mode specified by ***M*** is not valid.
SysOK	The function completed successfully.

SetSPBand

Sets the band value (BANDVAL parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:

```
function SetSPBand (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no band value (BANDVAL) parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOK	The function completed successfully.

Example:

```
SP7Bandval : Real;
...
SP7Bandval := 10.0
SetSPBand (7, SP7Bandval);
```

SetSPHyster

Sets the hysteresis value (HYSTER parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:

```
function SetSPHyster (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no hysteresis (HYSTER) parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOK	The function completed successfully.

Example:

```
SP5Hyster : Real;
...
SP5Hyster := 15.0;
SetSPHyster (5, SP5Hyster);
```

SetSPPreact

Sets the preact value (PREACT parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:

```
function SetSPPreact (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no preact (PREACT) parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOK	The function completed successfully.

Example:

```
SP2PreVal : Real;
...
SP2PreVal := 30.0;
SetSPPreact (2, SP2PreVal);
```

SetSPValue

Sets the value (VALUE parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:

```
function SetSPValue (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VALUE parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOutOfRange	The value specified for ***V*** is not in the allowed range for setpoint ***SP***.
SysOK	The function completed successfully.

Example:

```
SP4Val : Real;
...
SP4Val := 350.0;
SetSPValue (4, SP4Val);
```

SetSPNSample
For averaging (AVG) setpoints, sets the number of samples (NSAMPLE parameter) of setpoint ***SP*** to the value specified by ***N***.

Syntax:
```
function SetSPNSample (SP : Integer; N : Integer) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no NSAMPLE parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOutOfRange	The value specified for ***N*** is not in the allowed range for setpoint ***SP***.
SysOK	The function completed successfully.

Example:
```
SP5NS : Integer;
...
SP5NS := 10
SetSPNSample (5, SP5NS);
```

SetSPVover
For checkweigh (CHKWEI) setpoints, sets the overrange value (VOVER parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:
```
function SetSPVover (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VOVER parameter.
SysOK	The function completed successfully.

Example:
```
SP3VOR : Real;
...
SP3VOR := 35.5
SetSPVover (3, SP3VOR);
```

SetSPVunder
For checkweigh (CHKWEI) setpoints, sets the underrange value (VUNDER parameter) of setpoint ***SP*** to the value specified by ***V***.

Syntax:
```
function SetSPVunder (SP : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no VUNDER parameter.
SysOK	The function completed successfully.

Example:
```
SP4VUR : Real;
...
SP4VUR := 26.4
SetSPVunder (4, SP4VUR);
```

SetSPTime
For time of day (TOD) setpoints, sets the trip time (TIME parameter) of setpoint ***SP*** to the value specified by ***DT***.

Syntax:
```
function SetSPTime (SP : Integer; DT : DateTime) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no TIME parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOutOfRange	The value specified for ***DT*** is not in the allowed range for setpoint ***SP***.
SysOK	The function completed successfully.

Example:

```
SP2TIME : DateTime;
…
SP2TIME := 08:15:00
SetSPTime (2, SP2TIME);
```

SetSPDuration

For time of day (TOD) setpoints, sets the trip duration (DURATION parameter) of setpoint ***SP*** to the value specified by ***DT***.

Syntax:

```
function SetSPDuration (SP : Integer; DT : DateTime) : SysCode;
```

SysCode values returned:

SysInvalidSetpoint	The setpoint specified by ***SP*** does not exist.
SysInvalidRequest	The setpoint specified by ***SP*** has no DURATION parameter.
SysBatchRunning	The value cannot be changed because a batch process is currently running.
SysOutOfRange	The value specified for ***DT*** is not in the allowed range for setpoint ***SP***.
SysOK	The function completed successfully.

Example:

```
SP3DUR : DateTime;
…
SP3DUR := 00:3:15
SetSPDuration (3, SP3DUR);
```

StartBatch

Starts or resumes a batch run.

Syntax:

```
function StartBatch : SysCode;
```

SysCode values returned:

SysPermissionDenied	The BATCHNG configuration parameter is set to OFF.
SysBatchRunning	A batch process is already in progress.
SysOK	The function completed successfully.

StopBatch

Stops a currently running batch.

Syntax:

```
function StopBatch : SysCode;
```

SysCode values returned:

SysPermissionDenied	The BATCHNG configuration parameter is set to OFF.
SysBatchNotRunning	No batch process is running.
SysOK	The function completed successfully.

PauseBatch

Initiates a latched pause of a running batch process.

Syntax:

```
function PauseBatch : SysCode;
```

SysCode values returned:

SysPermissionDenied	The BATCHNG configuration parameter is set to OFF.
SysBatchRunning	A batch process is already in progress.
SysOK	The function completed successfully.

ResetBatch

Terminates a running, stopped, or paused batch process and resets the batch system.

Syntax:

```
function ResetBatch : SysCode;
```

SysCode values returned:

SysPermissionDenied	The BATCHNG configuration parameter is set to OFF.
SysOK	The function completed successfully.

4.5 Serial I/O

Print

Requests a print operation using the print format specified by ***F***. Output is sent to the port specified in the print format configuration.

Syntax:
```
function Print (F : PrintFormat) : SysCode;
```

PrintFormat values sent:

GrossFmt	Gross format
NetFmt	Net format
TrWInFmt	Truck weigh-in format
TrRegFmt	Truck register format (truck IDs and tare weights)
TrWOutFmt	Truck weigh-out format
SPFmt	Setpoint format
AccumFmt	Accumulator format
AuxFmt	Auxiliary format

SysCode values returned:

SysInvalidRequest	The print format specified by ***F*** does not exist.
SysQFull	The request could not be processed because the print queue is full.
SysOK	The function completed successfully.

Example:
```
Fmtout : PrintFormat;
...
Fmtout := NetFmt
Print (Fmtout);
```

StartStreaming

Starts data streaming for the port number specified by ***P***.

Syntax:
```
function StartStreaming (P : Integer) : SysCode;
```

SysCode values returned:

SysInvalidPort	The port number specified for ***P*** is not valid.
SysInvalidRequest	The port specified for ***P*** is not configured for streaming.
SysOK	The function completed successfully.

Example:
```
StartStreaming (1);
```

StopStreaming

Stops data streaming for the port number specified by ***P***.

Syntax:
```
function StopStreaming (P : Integer) : SysCode;
```

SysCode values returned:

SysInvalidPort	The port number specified for ***P*** is not valid.
SysInvalidRequest	The port specified for ***P*** is not configured for streaming.
SysOK	The function completed successfully.

Example:
```
StopStreaming (1);
```

Write

Writes the text specified in the ***<arg-list>*** to the port specified by ***P***. A subsequent ***Write*** or ***WriteLn*** operation will begin where this ***Write*** operation ends; a carriage return is not included at the end of the data sent to the port.

Syntax:
```
procedure Write (P : Integer; <arg-list>);
```

Example:
```
Write (Port1, "This is a test.");
```

WriteLn

Writes the text specified in the ***<arg-list>*** to the port specified by ***P***, followed by a carriage return and a line feed (CR/LF). The line feed (LF) can be suppressed by setting the indicator TERMIN parameter for the specified port to *CR* in the SERIAL menu configuration. A subsequent ***Write*** or ***WriteLn*** operation begins on the next line. See Section 5.8 on page 78 for a programming example that uses the WriteLn function.

Syntax:

```
procedure Write (P : Integer; <arg-list>);
```

Example:

```
WriteLn (Port1, "This is another test.");
```

4.6 Digital I/O Control

In the following digital I/O control functions, slot 0 represents the J2 connector on the indicator CPU board and supports four digital I/O bits (1–4). Digital I/O on expansion boards (slots 1–14) each support 24 bits of I/O (bits 1–24).

See the CheckWeigher program in Section 5.7 on page 67 for examples of using digital I/O control in a program.

GetDigin

Sets ***V*** to the value of the digital input assigned to slot ***S***, bit ***D***. GetDigin sets the value of ***V*** to 0 if the input is on, to 1 if the input is off. Note that the values returned are the reverse of those used when setting an output with the SetDigout function.

Syntax:

```
function GetDigin (S : Integer; D : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	The slot and bit assignment specified is not a valid digital input.
SysOK	The function completed successfully.

Example:

```
DIGINS0B3 : Integer;
…
GetDigin (0, 3, DIGINS0B3);
WriteLn (Port1, "Digin S0B3 status is", DIGINS0B3);
```

GetDigout

Sets ***V*** to the value of the digital output assigned to slot ***S***, bit ***D***. GetDigout sets the value of ***V*** to 0 if the output is on, to 1 if the output is off. Note that the values returned are the reverse of those used when setting an output with the SetDigout function.

Syntax:

```
function GetDigout (S : Integer; D : Integer; VAR V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	The slot and bit assignment specified is not a valid digital output.
SysOK	The function completed successfully.

Example:

```
DIGOUTS0B2 : Integer;
…
GetDigout (0, 2, DIGOUTS0B2);
WriteLn (Port1, "Digout S0B2 status is", DIGOUTS0B2);
```

SetDigout

Sets value of the digital output assigned to slot ***S***, bit ***D,*** to the value specified by ***V***. Set ***V*** to 1 to turn the specified output on; set ***V*** to 0 to turn the output off.

Syntax:

```
function SetDigout (S : Integer;  D : Integer; V : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	The slot and bit assignment specified is not a valid digital output.
SysOutOfRange	The value ***V*** must be 0 (inactive) or 1 (active).
SysOK	The function completed successfully.

Example:

```
DIGOUTS0B2 : Integer;
…
DIGOUTS0B2 := 0;
SetDigout (0, 2, DIGOUTS0B2);
```

4.7 Analog Output Operations

SetAlgout

Sets the analog output card in slot ***S*** to the percentage ***P***. Negative ***P*** values are set to zero; values greater than 100.0 are set to 100.0.

Syntax:

```
function SetAlgout (S : Integer;  P : Real) : SysCode;
```

SysCode values returned:

SysInvalidPort	The specified slot (***S***) is not a valid analog output.
SysInvalidRequest	The analog output is not configured from program control.
SysOK	The function completed successfully.

4.8 Pulse Input Operations

PulseRate

Sets ***R*** to the current pulse rate (in pulses per second) of the pulse input card in slot ***S***.

Syntax:

```
function PulseRate (S : Integer;  VAR R : Integer) : SysCode;
```

SysCode values returned:

SysInvalidCounter	The specified counter (***S***) is not a valid pulse input.
SysOK	The function completed successfully.

PulseCount

Sets ***C*** to the current pulse count of the pulse input card in slot ***S***.

Syntax:

```
function PulseCount (S : Integer;  VAR C : Integer) : SysCode;
```

SysCode values returned:

SysInvalidCounter	The specified counter (***S***) is not a valid pulse input.
SysOK	The function completed successfully.

ClearPulseCount

Sets the pulse count of the pulse input card in slot ***S*** to zero.

Syntax:

```
function ClearPulseCount (S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidCounter	The specified counter (***S***) is not a valid pulse input.
SysOK	The function completed successfully.

4.9 Display Operations

DisplayStatus

Displays the string ***msg*** in the front panel status message area. The length of string ***msg*** should not exceed 32 characters.

Syntax:

```
procedure DisplayStatus (msg : String);
```

PromptUser

Opens the alpha entry box and places the string ***msg*** in the user prompt area.

Syntax:

```
function PromptUser (msg : String) : SysCode;
```

SysCode values returned:

SysRequestFailed	The prompt could not be opened.
SysOK	The function completed successfully.

ClosePrompt

Closes a prompt opened by the PromptUser function.

Syntax:

```
procedure ClosePrompt;
```

GetEntry
Retrieves the user entry from a programmed prompt.

Syntax:
```
procedure GetEntry : String;
```

SelectScreen
Selects the configured screen, ***N***, to show on the indicator display.

Syntax:
```
function SelectScreen (N : Integer) : SysCode;
```

SysCode values returned:

SysInvalidRequest	The value specified for ***N*** is less than 1 or greater than 10.
SysOK	The function completed successfully.

SetEntry
Sets the user entry for a programmed prompt. This procedure can be used to provide a default value for entry box text when prompting the operator for input. Up to 1000 characters can be specified.

Syntax:
```
procedure SetEntry : String;
```

4.10 Display Programming

SetBargraphLevel
Sets the displayed level of bargraph widget ***W*** to the percentage (0–100%) specified by ***Level*** .

Syntax:
```
function SetBargraphLevel (W : Integer; Level : Integer) : SysCode;
```

SysCode values returned:

SysInvalidWidget	The bargraph widget specified by ***W*** does not exist.
SysOK	The function completed successfully.

SetLabelText
Sets the text of label widget ***W*** to ***S***.

Syntax:
```
function SetLabelText (W : Integer; S : String) : SysCode;
```

SysCode values returned:

SysInvalidWidget	The label widget specified by ***W*** does not exist.
SysOK	The function completed successfully.

SetNumericValue
Sets the value of numeric widget ***W*** to ***V***.

Syntax:
```
function SetNumericValue (W : Integer; V : Real) : SysCode;
```

SysCode values returned:

SysInvalidWidget	The numeric widget specified by ***W*** does not exist.
SysOK	The function completed successfully.

SetSymbolState
Sets the state of symbol widget ***W*** to ***S***. The widget state determines the variant of the widget symbol displayed. All widgets have at least two states (values 1 and 2); some have three (3). See Section 9.0 of the *920i Installation Manual* for descriptions of the symbol widget states.

Syntax:
```
function SetSymbolState (W : Integer; S : Integer) : SysCode;
```

SysCode values returned:

SysInvalidWidget	The symbol widget specified by ***W*** does not exist.
SysOK	The function completed successfully.

SetWidgetVisibility
Sets the visibility state of widget ***W*** to ***V***.

Syntax:
```
function SetWidgetVisibility (W : Integer; V : OnOffType) : SysCode;
```

SysCode values returned:

SysInvalidWidget	The widget specified by ***W*** does not exist.
SysOK	The function completed successfully.

4.11 Database Operations

For examples of the following database functions, see the example program in Section 5.9 on page 80.

<DB>.Add

Adds a record to the referenced database. Using this function invalidates any previous sort operation.

Syntax:

```
function <DB>.Add : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysDatabaseFull	There is no space in the specified database for this record.
SysOK	The function completed successfully.

<DB>.Clear

Clears all records from the referenced database.

Syntax:

```
function <DB>.Clear : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysOK	The function completed successfully.

<DB>.Delete

Deletes the current record from the referenced database. Using this function invalidates any previous sort operation.

Syntax:

```
function <DB>.Delete : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

*The following **<DB.Find>** functions allow a database to be searched. Column **I** is an alias for the field name, generated by the "Generate iRev import file" operation. The value to be matched is set in the working database record, in the field corresponding to column **I**, before a call to **<DB>.FindFirst** or **<DB>.FindLast**.*

<DB>.FindFirst

Finds the first record in the referenced database that matches the contents of ***<DB>*** column ***I***.

Syntax:

```
function <DB>.FindFirst (I : Integer) : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysNoSuchColumn	The column specified by ***I*** does not exist.
SysOK	The function completed successfully.

<DB>.FindLast

Finds the last record in the referenced database that matches the contents of ***<DB>*** column ***I***.

Syntax:

```
function <DB>.FindLast (I : Integer) : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysNoSuchColumn	The column specified by ***I*** does not exist.
SysOK	The function completed successfully.

<DB>.FindNext

Finds the next record in the referenced database that matches the criteria of a previous FindFirst or FindLast operation.

Syntax:

```
function <DB>.FindNext : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.FindPrev
Finds the previous record in the referenced database that matches the criteria of a previous FindFirst or FindLast operation.

Syntax:
```
function <DB>.FindLast : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.GetFirst
Retrieves the first logical record from the referenced database.

Syntax:
```
function <DB>.GetFirst : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.GetLast
Retrieves the last logical record from the referenced database.

Syntax:
```
function <DB>.GetLast : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.GetNext
Retrieves the next logical record from the referenced database.

Syntax:
```
function <DB>.GetNext : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.GetPrev
Retrieves the previous logical record from the referenced database.

Syntax:
```
function <DB>.GetPrev : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.Sort
Sorts database ***<DB>*** into ascending order based on the contents of column ***I***. The sort table supports a maximum of 30 000 elements. Databases with more than 30 000 records cannot be sorted.

Syntax:
```
function <DB>.Sort (I : Integer) : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

<DB>.Update

Updates the current record in the referenced database with the contents of ***<DB>*** . Using this function invalidates any previous sort operation.

Syntax:
```
function <DB>.Update : SysCode;
```

SysCode values returned:

SysNoSuchDatabase	The referenced database cannot be found.
SysNoSuchRecord	The requested record is not contained in the database.
SysOK	The function completed successfully.

4.12 Timer Controls

Thirty-two timers, configurable as either continuous and one-shot timers, can be used to generate events at some time in the future. The shortest interval for which a timer can be set is 10 ms. See the system startup event handler for the CheckWeigher program on page 77 for an example of how to set up timers.

ResetTimer

Resets the value of timer ***T*** (1–32) by stopping the timer, setting the timer mode to TimerOneShot, and setting the timer time-out to 1.

Syntax:
```
function ResetTimer (T : Integer) : Syscode;
```

SysCode values returned:

SysInvalidTimer	The timer specified by ***T*** a not valid timer.
SysOK	The function completed successfully.

SetTimer

Sets the time-out value of timer ***T*** (1–32). Timer values are specified in 0.01-second intervals (1= 10 ms, 100 = 1 second). For one-shot timers, the SetTimer function must be called again to restart the timer once it has expired.

Syntax:
```
function SetTimer (T : Integer ; V : Integer) : Syscode;
```

SysCode values returned:

SysInvalidTimer	The timer specified by ***T*** a not valid timer.
SysOK	The function completed successfully.

SetTimerMode

Sets the mode value, ***M***, of timer ***T*** (1–32). This function, normally included in a program startup handler, only needs to be called once for each timer unless the timer mode is changed.

Syntax:
```
function SetTimer (T : Integer ; M : TimerMode) : Syscode;
```

TimerMode values sent:

TimerOneShot	Timer mode is set to one-shot.
TimerContinuous	Timer mode is set to continuous.

SysCode values returned:

SysInvalidTimer	The timer specified by ***T*** a not valid timer.
SysInvalidState	The timer mode specified by ***M*** a not valid timer mode.
SysOK	The function completed successfully.

StartTimer

Starts timer ***T*** (1–32). For one-shot timers, this function must be called each time the timer is used. Continuous timers are started only once; they do not require another call to StartTimer unless stopped by a call to the StopTimer function.

Syntax:
```
function StartTimer (T : Integer) : Syscode;
```

SysCode values returned:

SysInvalidTimer	The timer specified by ***T*** a not valid timer.
SysOK	The function completed successfully.

StopTimer

Stops timer ***T*** (1–32).

Syntax:
```
function StopTimer (T : Integer) : Syscode;
```

SysCode values returned:

SysInvalidTimer	The timer specified by ***T*** a not valid timer.
SysOK	The function completed successfully.

4.13 Mathematical Operations

Abs
Returns the absolute value of ***x***.

Syntax:
```
function Abs (x : Real) : Real;
```

ATan
Returns a value between –π/2 and π/2, representing the arctangent of ***x*** in radians.

Syntax:
```
function Atan (x : Real) : Real;
```

Ceil
Returns the smallest integer greater than or equal to ***x***.

Syntax:
```
function Ceil (x : Real) : Integer;
```

Cos
Returns the cosine of ***x***. ***x*** must be specified in radians.

Syntax:
```
function Cos (x : Real) : Real;
```

Exp
Returns the value of e^x.

Syntax:
```
function Exp (x : Real) : Real;
```

Log
Returns the value of $\log_e(x)$.

Syntax:
```
function Log (x : Real) : Real;
```

Log10
Returns the value of $\log_{10}(x)$.

Syntax:
```
function Log10 (x : Real) : Real;
```

Sign
Returns the sign of the numeric operand. If $x < 0$, the function returns a value of –1; otherwise, the value returned is 1.

Syntax:
```
function Sign (x : Real) : Integer;
```

Sin
Returns the sine of ***x***. ***x*** must be specified in radians.

Syntax:
```
function Sin (x : Real) : Real;
```

Sqrt
Returns the square root of ***x***.

Syntax:
```
function Sqrt (x : Real) : Real;
```

Tan
Returns the tangent of ***x***. ***x*** must be specified in radians.

Syntax:
```
function Tan (x : Real) : Real;
```

4.14 Bit-wise Operations

BitAnd
Returns the bit-wise AND result of ***X*** and ***Y***.

Syntax:
```
function BitAnd (X : Integer; Y : Integer) : Integer;
```

BitOr
Returns the bit-wise OR result of ***X*** and ***Y***.

Syntax:
```
function BitOr (X : Integer; Y : Integer) : Integer;
```

BitNot

Returns the bit-wise NOT result of ***X***.

Syntax:
```
function BitNOT (X : Integer) : Integer;
```

BitXor
Returns the bit-wise exclusive OR (XOR) result of ***X*** and ***Y***.

Syntax:
```
function BitXor (X : Integer; Y : Integer) : Integer;
```

4.15 String Operations

Asc
Returns the ASCII value of the first character of string ***S***. If ***S*** is an empty string, the value returned is 0.

Syntax:
```
function Asc (S : String) : Integer;
```

Chr$
Returns a one-character string containing the ASCII character represented by ***I***.

Syntax:
```
function Chr$ (I : Integer) : String;
```

Hex$
Returns an eight-character hexadecimal string equivalent to ***I***.

Syntax:
```
function Hex$ (I : Integer) : String;
```

LCase$
Returns the string ***S*** with all upper-case letters converted to lower case.

Syntax:
```
function LCase$ (S : String) : String;
```

Left$
Returns a string containing the leftmost ***I*** characters of string ***S***. If ***I*** is greater than the length of ***S***, the function returns a copy of ***S***.

Syntax:
```
function Left$ (S : String; I : Integer) : String;
```

Len
Returns the length (number of characters) of string ***S***.

Syntax:
```
function Len (S : String) : Integer;
```

Mid$
Returns a number of characters (specified by ***length***) from string ***s***, beginning with the character specified by ***start***. If ***start*** is greater than the string length, the result is an empty string. If ***start + length*** is greater than the length of ***S***, the returned value contains the characters from ***start*** through the end of ***S***.

Syntax:
```
function Mid$ (S : String; start : Integer; length : Integer) : String;
```

Oct$
Returns an 11-character octal string equivalent to ***I***.

Syntax:
```
function Oct$ (I : Integer) : String;
```

Right$
Returns a string containing the rightmost ***I*** characters of string ***S***. If ***I*** is greater than the length of ***S***, the function returns a copy of ***S***.

Syntax:
```
function Right$ (S : String; I : Integer) : String;
```

Space$
Returns a string containing ***N*** spaces.

Syntax:
```
function Space$ (N : Integer) : String;
```

UCase$
Returns the string ***S*** with all lower-case letters converted to upper case.

Syntax:
```
function UCase$ (S : String) : String;
```

4.16 Data Conversion

IntegerToString
Returns a string representation of the integer ***I*** with a minimum length of ***W***. If ***W*** is less than zero, zero is used as the minimum length. If ***W*** is greater than 100, 100 is used as the minimum length.

Syntax:
```
function IntegerToString (I : Integer; W : Integer) : String;
```

RealToString
Returns a string representation of the real number ***R*** with a minimum length of ***W***, with ***P*** digits to the right of the decimal point. If ***W*** is less than zero, zero is used as the minimum length; if ***W*** is greater than 100, 100 is used as the minimum length. If ***P*** is less than zero, zero is used as the precision; if ***P*** is greater than 20, 20 is used.

Syntax:
```
function RealToString (R : Real; W : Integer; P: Integer) : String;
```

StringToInteger
Returns the integer equivalent of the numeric string ***S***. If ***S*** is not a valid string, the function returns the value 0.

Syntax:
```
function StringToInteger (S : String) : Integer;
```

StringToReal
Returns the real number equivalent of the numeric string ***S***. If ***S*** is not a valid string, the function returns the value 0.0.

Syntax:
```
function StringToReal (S : String) : Real;
```

5.0 Programming Examples

5.1 Handler Examples

The following simple programs show examples of event handlers used to initiate other indicator functions. See the list of event handlers in Section 6.1 on page 90.

Example 1: Batch Paused by Digital Input

In the following program, the event handler, `DiginS0B3Activate` is used to pause a running batch whenever the digital input assigned to slot 0, bit 3, goes low. The `StartBatch` function or other input would be required to restart the batch.

```
program DigPause

  handler DiginS0B3Activate;

  begin
    PauseBatch;
  end;
end DigPause;
```

Example 2: Batching Mode Set by Digital Input

In the following program, the event handler `DiginS0B3Activate` is used to set the batching mode to AUTO whenever the digital input assigned to slot 0, bit 3, goes low (active).

The `DiginS0B3Deactivate` handler is used to set the batching state to MANUAL whenever the digital input goes high (inactive). The batching mode change takes place immediately, even if a batch sequence is in progress.

```
program AutoMan

  handler DiginS0B3Activate;

  begin
    SetBatchingMode(AUTO);
  end;

  handler DiginS0B3Deactivate;

  begin
    SetBatchingMode(MANUAL);
  end;

end AutoMan;
```

Example 3: Serial Notification of Setpoint Trip

The following program uses the event handler `SP1Trip` to send a notification to Port 1 that the setpoint has tripped.

```
program SPDebug

  handler SP1Trip;

  begin
    Writeln (1, "SP 1 TRIPPED");
  end;
end SPDebug;
```

Example 4: Setpoints Toggled by Softkeys

In the following program, the event handlers `User2KeyPressed` and `User4KeyPressed` are used to turn groups of setpoints on and off. When the second softkey is pressed, setpoints 1 and 3 are turned off, setpoints 2 and 4 are turned on. When the fourth softkey is pressed, setpoints 1 and 3 are turned on, 2 and 4 are turned off.

```
program SPTURN

  handler User2KeyPressed;

  begin
    DisableSP(1);
    DisableSP(3);
    EnableSP(2);
    EnableSP(4);

  end;

  handler User4KeyPressed;

  begin
    DisableSP(2);
    DisableSP(4);
    EnableSP(1);
    EnableSP(3);
  end;
```

```
end SPTURN;
```

5.2 "Hello": A Simple Handler Program

```
program Hello;

  --
  -- Define constants for use in the program
  --

  Port1 : constant integer := 1;
  Greeting : constant string := "HELLO 920i USER!!!";

--
--  User1KeyPressed handler to display message on
--  920i Status line and write the message out RS232
--  Port 1.
--

handler User1KeyPressed;

  begin
    DisplayStatus(Greeting);
    WriteLn(Port1, Greeting);
  end;

--
--  Clear the status line with the User2KeyPressed handler
--

handler User2KeyPressed;

  begin
    DisplayStatus("");
  end;

end Hello;
```

5.3 "Looptest" Program

```
Program LoopTest

Port1 : contant integer := 1;

handler User1KeyPressed;

  LoopCounter : integer;

  begin
    for LoopCounter := 1 to 10
    loop
      WriteLn(Port1, IntegerToString(LoopCounter, 0));
    end loop;
  end;
--------------------------------------------------------
-- Second loop test counts from 10 to 1 with a 'while' loop
-- and writes the values out a RS232 port.
--------------------------------------------------------
handler User2KeyPressed;

  LoopCounter : integer;

  begin
    LoopCounter := 10;
    while LoopCounter > 1
    loop
```

```
      WriteLn(Port1, LoopCounter);
      LoopCounter := LoopCounter - 1;
    end loop;
  end;

--------------------------------------------------------
--  Third loop test counts from 10 to 1 with no 'for' or 'while'
--  condition. Notice the 'exit' statement is required to stop
--  the loop.  Otherwise the system will stay in the loop: NOT GOOD!
--------------------------------------------------------
handler User3KeyPressed;

  LoopCounter : integer;

  begin
    LoopCounter := 10;
    loop
      WriteLn(Port1, LoopCounter);
      LoopCounter := LoopCounter - 1;
      if LoopCounter < 1 then
        exit;
      end if;
    end loop;
  end;

end LoopTest;
```

5.4 "HelloWorld" Program

```
------------------------------------------------------------------------------
-- Program Name: HelloWorld
--
-- Copyright 2002 RLWS as an unpublished work.
-- All Rights Reserved.
--
-- First written on January 16, 2002 by RLWS.
--
-- Module Description:
--   This file is the iRite source code for a 920i program.
--
--   This program will display the message "Hello, world!" on the 920i display
--   in the status message area when ever any one of the possible event types
--   occur. This program will demonstrate the 8 different types of events and
--   what it takes for the event to be actually be fired.
--
------------------------------------------------------------------------------

program HelloWorld;

  -- Soft key 1 (the first one on the left of the indicator) was pressed. The
  -- location of this softkey is always the first softkey on the left.  No
  -- additional configuration or programming is necessary for this event to
  -- be "active".
  handler Soft1KeyPressed;
    begin
      DisplayStatus("Hello, world! [Soft key]");
    end;


  -- User key 1 will be defined in the softkeys menu under "Features" in
  -- configuration mode.  This key's actual physical location can change since
  -- the user can put it anywhere in the list of softkeys.  In this program
  -- example it is required that we don't put it at the same physical location
  -- as softkey 1, because the Soft1KeyPressed handler would have visiblity
  -- over this (User1KeyPressed) handler, then this handler would never get
  -- to execute.
  handler User1KeyPressed;
    begin
      DisplayStatus("Hello, world! [User key]");
    end;


  -- Setpoint 1 was tripped.  To get it Setpoint 1 to trip, it will require
  -- some configuration to set it up. This event makes no assumptions about
  -- what kind sepoint 1 is, or how it is tripped: just that it WAS tripped.
  handler SP1Trip;
    begin
      DisplayStatus("Hello, world! [Setpoint]");
    end;


  -- DiginS0B1 is the digital input 1 on slot 0.  Slot zero is the onboard
  -- relays and bit 1 is the first relay.  We need to configure this I/O point
  -- as type "Programmability" thru setup mode or in iRev. To actually cause
  -- the event to fire this I/O point needs to be pulled to low (jump pin 2
  -- and pin 3 together on connector J2)
  handler DiginS0B1Activate;
    begin
      DisplayStatus("Hello, world! [Digital I/O]");
    end;


  -- The Port1CharReceived event will happen when ever a character is received
  -- on Port1.  Port one is the RS232 port access through connector J11. This
  -- ports "INPUT" type must be configured for "PROGIN" under the Serial menu
  -- for this to work.  Once the port has been configured as input type
  -- "PROGIN" all functionality for that port has been lost, except for the
  -- functionality the programmer adds to handler Port1CharReceived below.
  handler Port1CharReceived;
    begin
      DisplayStatus("Hello, world! [Port character]");
    end;


  -- The Cmd1Handler event will happen when ever the command F#1 is received
  -- on any communication port.  The command can be sent with an optional
  -- value like F#1=Hello<CR>.  We will only display "Hello, world!" if the
  -- command comes from port 2 and has a value of "Hello". (In this program
  -- we have written an event handler for character received on port1.  This
  -- will prevent the default char input handler from processing a "F#1"
  -- command [or any other command] on port 1. So this event will not fire if
  -- the command is sent to port 1).  Also it is important to note that there
  -- will not be any "OK" or any other response sent from the indicator when
  -- a valid F#1 thru F#32 command is received regardless of whether a handler
  -- like this one was used or not.  If a response is required, then it is the
  -- programmer's responsibility to send a response from within the handler.
  handler Cmd1Handler;
    begin
      if (EventPort = 2) and (EventString = "Hello") then
        DisplayStatus("Hello, world! [Command]");
      end if;
    end;


  -- The Timer1Trip event will happen every time timer 1 expires.  This event
  -- will require some additional programming to setup and start the timer. It
  -- can be programmed to trip only once or trip every x0 miliseconds or y.z
  -- seconds.
  handler Timer1Trip;
    begin
      DisplayStatus("Hello, world! [Timer]");
    end;
```

```
-- This is the ProgramStartup event even though it isn't explicitly called
-- that by name. It fires every time the 920i is powered-up, or the indicator
-- is taken out of setup (configuration) mode, or is sent the "RS" serial cmd.
begin

  DisplayStatus("Hello, world! [Startup]");

  -- We need to setup Timer 1 so it will fire 5.0 seconds after the startup
  -- handler runs to display the "Hello, world!" message.
  SetTimerMode(1, TimerOneShot);
  SetTimer(1, 500);
  StartTimer(1);

end HelloWorld;
```

5.5 "CircAreaVol": Computation Program

The following program uses some of *iRite*'s mathematical functions to calculate the circumference, area, and volume of a circle, based on prompted user input for the circle' s radius. This program uses functions defined in "NumTools" (see Section 5.6 on page 63).

```
------------------------------------------------------------------------------
-- Program Name: CircAreaVol
--
-- Copyright 2002 RLWS as an unpublished work.
-- All Rights Reserved.
--
-- The information contained herein is confidential property of RLWS.
-- The use, copying, transfer or disclosure of such information is prohibited
-- except by express written agreement with RLWS.
--
-- First written on January 17, 2002 by RLWS.
--
-- Module Description:
-- This file is the iRite source code for a 920i program.
--
-- This program will prompt the user for the radius of a circle then
-- calculate and display the circumferance, area and volume.  A user defined
-- softkey must be setup to initiate the prompting sequence.
------------------------------------------------------------------------------

program CircAreaVol;

#include numtools.iri

  strProgName : constant string := "CircAreaVol";
  rVersion : constant real := 0.01;
  iBuild : constant integer := 7;  -- Inc. by 1 after each compile.

  ----------------------------------------------------------------------------
  --  Function Name     : MakeVersionString
  --  Created by        : Company Name or Programmer Name
  --  Last modified on  : Jan 10, 2002
  --
  --  Purpose           : This function will format the version and build into
  --                      a string and return the string.
  --  Value Parameters  : none
  --  Variable Params   : none
  --  Return Parameter  : a string representation of the version and build in
  --                      the format xx.xx.xx
  --  Side Effects      : none
  ----------------------------------------------------------------------------

  function MakeVersionString : string;
    begin
      return (RealToString(rVersion, 4, 2) + "." + IntegerToString(iBuild, 2));
    end;
```

```
----------------------------------------------------------------------------
--  Procedure Name    : ShowVersion
--  Created by        : Company Name or Programmer Name
--  Last modified on  : Jan 10, 2002
--
--  Purpose           : This procedure will show the program name and
--                      version number in the display Status window.
--  Value Parameters  : none
--  Variable Params   : none
--  Side Effects      : none
----------------------------------------------------------------------------

procedure ShowVersion;
  begin
    DisplayStatus(strProgName + " Version: " + MakeVersionString);
  end;




----------------------------------------------------------------------------
-- Function Name     : Circum
-- Created By        : RLWS
-- Last Modified on  : 1/17/2002
-- Purpose           : This function will calculate the circumference of a
--                     circle with radius iRadius and return the result.
-- Value Parameters  : iRadius is the radius of a circle (1/2 the diameter).
-- Return Parameter  : Returns the circumerence.
-- Side Effects      : none
----------------------------------------------------------------------------
function Circum(rRadius : real) : real;
  const_rPie : constant real := 3.141592654;
  begin
    -- The circumferance of a circle is defined by: circum. = 2*pie*r.
    return (2 * const_rPie * rRadius);
  end;


----------------------------------------------------------------------------
-- Function Name     : CircleArea
-- Created By        : RLWS
-- Last Modified on  : 1/17/2002
-- Purpose           : This function will calculate the area of a
--                     circle with radius iRadius and return the result.
-- Value Parameters  : iRadius is the radius of a circle (1/2 the diameter).
-- Return Parameter  : Returns the area.
-- Side Effects      : none
----------------------------------------------------------------------------
function CircleArea(rRadius : real) : real;
  const_rPie : constant real := 3.141592654;
  begin
    -- The area of a circle is defined by: area = pie*(r^2).
    return (const_rPie * Power(rRadius, 2));
  end;


----------------------------------------------------------------------------
-- Function Name     : CircleVolume
-- Created By        : RLWS
-- Last Modified on  : 1/17/2002
-- Purpose           : This function will calculate the volume of a
--                     circle with radius iRadius and return the result.
-- Value Parameters  : iRadius is the radius of a circle (1/2 the diameter).
-- Return Parameter  : Returns the volume.
-- Side Effects      : none
----------------------------------------------------------------------------
```

```
function CircleVolume (rRadius : real) : real;
  const_rPie : constant real := 3.141592654;
  begin
    -- The volume of a circle is defined by: volume = (4/3)*pie*(r^3).
    return ((4.0/3.0) * const_rPie * Power(rRadius, 3));
  end;


----------------------------------------------------------------------------
-- Handler Name      : User1KeyPressed
-- Created By        : RLWS
-- Last Modified on  : 1/17/2002
-- Purpose           : User key 1 will be defined in the softkeys menu under
--                     "Features" in configuration mode.  This key's actual
--                     physical location can change since the user can put
--                     it anywhere in the list of softkeys.
--                     We will ask the user for the radius of a circle and
--                     then calculate the circ., area, and volume.
-- Side Effects      : none permanent
----------------------------------------------------------------------------

handler User1KeyPressed;

  strEntry : string := "";

  begin

    if PromptUser("circle: ") = SysOK then
      DisplayStatus("Please enter the radius of any");
    end if;
  end;



----------------------------------------------------------------------------
-- Handler Name      : UserEntry
-- Created By        : RLWS
-- Last Modified on  : 1/21/2002
-- Purpose           : This event will fire when ever the 920 is in entry
--                     mode and the user presses one of the event keys. So
--                     far, the only event keys defined are EnterKey and
--                     CancelKey.  Do determine which key generated the
--                     event, we call EventKey.
-- Side Effects      : none permanent
----------------------------------------------------------------------------

handler UserEntry;

  strEntry : string := "";
  rRadius : real;

  begin

    -- We need to find out which key caused the this event.
    if EventKey = EnterKey then

      -- Call GetEntry to get the string entered by the user before the
      -- "ENTER" key was pressed.
      strEntry := GetEntry;

      -- This will remove the numeric entry info and the entry prompt from
      -- the display.
      ClosePrompt;

      -- We need to make sure that they entered something, and that the
      -- something they entered is a numeric value (may contain a decimal).
      if (len(strEntry) > 0) and IsNumeric(strEntry, TRUE) then
```

```
          -- Convert the user entry into a real value representing the radius.
          rRadius := StringToReal(strEntry);

          -- Calculate the circumference, area and volume and show it on the
          -- display.
          DisplayStatus("C:" + RealToString(Circum(rRadius),7,2) +
                        "   A:" + RealToString(CircleArea(rRadius),7,2) +
                        "   V:" + RealToString(CircleVolume(rRadius),7,2));
        else
          DisplayStatus("Invalid Entry!");
        end if;

      else
        --  With all other keys that generate this event, we should just close
        -- the prompt.
        ClosePrompt;
        DisplayStatus("Entry Cancelled!");
      end if;
    end;


begin

  -- Initialize all global variables here.



  ShowVersion;

end CircAreaVol;
```

5.6 "NumTools": iRev Import File

"NumTools" is an *iRev* import file. It contains several mathematical functions used by program "CircAreaVol" (see Section 5.5 on page 60).

```
------------------------------------------------------------------------------
-- File Name: numtools.iri
--
-- Copyright 2002 Company Name as an unpublished work.
-- All Rights Reserved.
--
-- The information contained herein is confidential property of Company Name.
-- The use, copying, transfer or disclosure of such information is prohibited
-- except by express written agreement with Company Name.
--
-- First written on Month Day, Year by FirstName LastName.
--
-- Module Description:
-- This file is the iRev import file for a 920i program.
--
-- This module is (fill in detailed description of what functions and defines
-- are contained in this module).
------------------------------------------------------------------------------


  ----------------------------------------------------------------------------
  -- START: Constants and aliases here.


  -- END: Constants and aliases.
  ----------------------------------------------------------------------------


  ----------------------------------------------------------------------------
  -- START: defining all user created types (enumerations, records, arrays).
```

```
type boolean is (FALSE, TRUE);  -- false/true enumeration


-- END: defining all user created types (enumerations, records, arrays).
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- START: Defining all global variables here.


-- END: Defining all global variables.
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- START: Defining all functions and procedures here.


----------------------------------------------------------------------------
-- Function Name     : Inc
-- Created By        : RLWS
-- Last Modified on  : 01/121/02
-- Purpose           : This function will add 1 to the value of the number
--                     passed in iNumber.  The new incremented value is also
--                     returned so the number can be used and incremented
--                     in the same line of code.
-- Value Parameters  : none
-- Variable Params   : iNumber, a number that will get incremented by 1.
-- Return Parameter  : The new (incremented) value of iNumber is returned.
-- Side Effects      : None
----------------------------------------------------------------------------

function Inc(var iNumber : integer) : integer;

  begin
    iNumber := iNumber + 1;
    return iNumber;
  end;




----------------------------------------------------------------------------
-- Function Name     : Dec
-- Created By        : RLWS
-- Last Modified on  : 01/121/02
-- Purpose           : This function will subtract 1 from the value of the
--                     number passed in iNumber.  The new decremented value
--                     is also returned so the number can be used and
--                     decremented in the same line of code.
-- Value Parameters  : none
-- Variable Params   : iNumber, a number that will get decremented by 1.
-- Return Parameter  : The new (decremented) value of iNumber is returned.
-- Side Effects      : None
----------------------------------------------------------------------------

function Dec(var iNumber : integer) : integer;

  begin
    iNumber := iNumber - 1;
    return iNumber;
  end;
```

```
----------------------------------------------------------------------------
-- Function Name     : IsNumeric
-- Created By        : RLWS
-- Last Modified on  : 01/15/02
--
-- Purpose           : This function will determine if the string passed in
--                     is composed of only numeric characters with the
--                     option to allow one decimal point.
--
-- Value Parameters  : strEntry : string - a string (probably from user
--                       keyboard input) that we are looking into.
--                     boDecimalAllowed : boolean - if TRUE then one of the
--                       characters in strEntry may be a decimal place.
-- Variable Params   : none
-- Return Parameter  : TRUE if strEntry is numeric.
-- Side Effects      : None
----------------------------------------------------------------------------

function IsNumeric(strEntry         : string;
                   boDecimalAllowed : boolean) : boolean;

  boDone : boolean := FALSE;
  boDecFound : boolean := FALSE;
  strOneChar : string;
  iIndex : integer := 1;
  boReturn : boolean := TRUE;


  begin

    -- We want to step through the string strEntry and look at each
    -- character to determine if it is a number.  If boDecimalAllowed is
    -- TRUE then the character can also be a decimal point once.

    while (boDone <> TRUE) and (iIndex <= len(strEntry))
    loop
      -- We need to grab only one character of the string.
      strOneChar := Mid$(strEntry, iIndex, 1);
      if (strOneChar < "0") or (strOneChar > "9") then
        -- The character is not a number, but we need to check if it is a
        -- ".' decimal point if decimal points are allowed and there hasn't
        -- been a decimal point found yet.
        if boDecimalAllowed and (strOneChar = ".") and (boDecFound = FALSE) then
           -- This means that decimal are allowed, the character IS a
           -- decimal point, and we waven't found one yet.

           -- Now we have found one so set the flag to TRUE.
           boDecFound := TRUE;
        else
          -- If we get here then that means we have found a non numeric
          -- character, or a decimal place when decimals aren't allowed or
          -- a second decimal place.  We want to stop and return FALSE.
          boReturn := FALSE;
          boDone := TRUE;
        end if;
      end if;

      -- Increment the character index into the string
      Inc(iIndex);

    end loop;
    -- This loop will execute until boDone = TRUE or until iIndex is greater
    -- then the lenth of the string strEntry.

    return boReturn;
  end;
```

```
----------------------------------------------------------------------------
-- Function Name     : Power
-- Created By        : RLWS
-- Last Modified on  : 1/17/2002
-- Purpose           : This function raises the value of rNumber to the power
--                     of iPower and returns the result.
-- Value Parameters  : rNumber is any real number.
--                     iPower is any integer.
-- Return Parameter  : Returns the calculated value as a real number.
-- Side Effects      : none
----------------------------------------------------------------------------

function Power (rNumber : real; iPower : integer) : real;
  rResult : real := 1;
  iCounter : integer;

  begin
    -- The number raised to a power is calculated by rNumber^iPower
    -- There are some special cases that we need to trap or the results will
    -- be undefined.
    if (rNumber >= -0.000000001) and (rNumber <= 0.000000001) then
      -- If rNumber is zero then the results could get ugly if iPower is
      -- less then or equal to zero.
      rResult := 0;
    elsif iPower = 0 then
      -- Any number raised to a power of zero is 1, except 0.
      rResult := 1;
    elsif iPower < 0 then
      -- iPower is negative.  This means that we will have to take the
      -- inverse of the result.

      -- Use the absolute value of iPower to eliminate the negative sign.
      iPower := abs(iPower);

      -- Multiply rNumber by 1 the first iteration and then multiply it by
      -- itself for each time after that up to iPower.
      for iCounter := 1 to iPower
      loop
        rResult := rResult * rNumber;
      end loop;

      -- We will have to take the inverse of the result to make the (-)
      -- in the exponent meaningful.
      rResult := 1/rResult;
    else
      -- Multiply rNumber by 1 the first iteration and then multiply it by
      -- itself for each time after that up to iPower.
      for iCounter := 1 to iPower
      loop
        rResult := rResult * rNumber;
      end loop;
    end if;

    return rResult;
  end;


-- END: Defining all functions and procedures.
----------------------------------------------------------------------------
```

5.7 Simple Checkweigher Program

```
-------------------------------------------------------------------------------
-- Program Name: CheckWeigher
--
-- Copyright 2002 RLWS as an unpublished work.
-- All Rights Reserved.
--
-- The information contained herein is confidential property of
-- RLWS.  The use, copying, transfer or disclosure of such
-- information is prohibited except by express written agreement with
-- RLWS.
--
-- First written on February 04, 2002 by EPD.
--
-- Module Description:
-- This file is the iRite source code for a 920i program.
--
-- This program will prompt the user for a min, max, and thresh weight for a
-- particular part.  When the user starts the check weighing a converyor will
-- start and move a piece onto the scale (with the help of a photoeye).  When
-- the piece is in position, the conveyor is stopped.  The weight of the piece
-- is captured (at standstill).  If the piece is over the theshold but also
-- over/under the tolerance values then an output is
-- turned on to move a ram to push the piece into an "reject" bin.  The
-- converyor will restart and move the next piece onto the scale.  The user
-- can move a switch to select "Run" or "Pause".
-------------------------------------------------------------------------------

-- Please keep all lines 79 characters or less.
-- This line is exactly 79 characters long. ----------------------------------


-- The name of the program is always the first line of code.
program CheckWeigher;

-- Put Include (.iri) file here.
#include common.iri
#include numtools.iri

  -----------------------------------------------------------------------------
  -- Begin constants and aliases definitions here.

  g_csProgName : constant string := "EZ Check Weigher";
  g_csVersion : constant string := "0.01";

  g_ciDebugOn : constant integer := 1;

  g_ciScale : constant integer := 1;

  -- Digital I/O Aliases
  g_ciOnBoardIO : constant integer := 0;
  g_ciPhotoEye  : constant integer := 1;  -- input
  g_ciRunPause  : constant integer := 2;  -- input
  g_ciConveyor  : constant integer := 3;  -- output
  g_ciRejectRam : constant integer := 4;  -- output


  -- This is the timer for holding on the reject ram.
  g_ciRejectRamOnTimer : constant integer := 1;
  g_ciRamTime : constant integer := 100;  -- 1 second

  -- This is the timer for checking the state of the process very often.
  g_ciProcessTimer : constant integer := 2;
  g_ciCheckProcTime : constant integer := 1;  -- 10 milliseconds

  -- End of constants and aliases.
  -----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
-- Begin user created types (enumerations, records, arrays) definitions.

type EntryMode is (NONE, MINVAL, MAXVAL, THRESHOLD);

-- End of user created types (enumerations, records, arrays).
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- Start global variables definitions here.  Don't forget to initialize
-- them in the system startup event handler.

g_iBuild       : integer;
g_flgFatalError : boolean;

g_flgStart : boolean;
g_flgRunning : boolean;
g_flgConveyorOn : boolean;
g_flgPhotoEyeMade : boolean;
g_flgWaitingForClear : boolean;

g_rLowTolerance : stored real;
g_rHighTolerance : stored real;
g_rThreshold : stored real;

g_eEntryMode : EntryMode;

-- End of global variables.
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- Start functions and procedures definitions here.

----------------------------------------------------------------------------
--  Function Name     : MakeVersionString
--  Created by        : Company Name or Programmer Name
--  Last modified on  : Jan 10, 2002
--
--  Purpose           : This function will format the version and build into
--                      a string and return the string.
--  Value Parameters  : none
--  Variable Params   : none
--  Return Parameter  : a string representation of the version and build in
--                      the format xx.xx.xx
--  Side Effects      : none
----------------------------------------------------------------------------
function MakeVersionString : string;
  sTemp : string;
begin
  if g_iBuild > 9 then
    sTemp := ("Ver " + g_csVersion + "." + IntegerToString(g_iBuild, 2));
  else
    sTemp := ("Ver " + g_csVersion + ".0" + IntegerToString(g_iBuild, 1));
  end if;

  return sTemp;
end;
```

```
----------------------------------------------------------------------------
--  Procedure Name    : DisplayVersion
--  Created by        : Company Name or Programmer Name
--  Last modified on  : Jan 15, 2002
--
--  Purpose           : This procedure will diplay the program name and
--                      version number to the diplay.
--  Value Parameters  : none
--  Variable Params   : none
--  Side Effects      : The program name and version is put on the display.
----------------------------------------------------------------------------
procedure DisplayVersion;
begin
  DisplayStatus(g_csProgName + "  " + MakeVersionString);
end;


----------------------------------------------------------------------------
-- Procedure Name    : StartConveyor
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : The digital output defined by Conveyor will be turned
--                     on.  The global flag g_flgConveyorOn will be set to
--                     TRUE.  The conveyor must run for a short while to
--                     clear any item that is breaking the photoeye when
--                     the conveyor is started.  To accomplish this the
--                     global flag g_flgWaitingForClear must be set to TRUE.
-- Value Parameters  : none
-- Variable Params   : none
-- Side Effects      : Global variables g_flgConveyorOn and
--                     g_flgWaitingForClear are changed.
----------------------------------------------------------------------------
procedure StartConveyor;
begin
  g_flgConveyorOn := TRUE;

  -- This flag gets set in the Digin active/deactive handlers.
  if g_flgPhotoEyeMade then
    -- Something is blocking the photoeye.  We need to start the conveyor and
    -- let it run until the piece blocking the photoeye is cleared and then
    -- the next piece breaks the photoeye again.

    g_flgWaitingForClear := TRUE;

  else
    g_flgWaitingForClear := FALSE;
  end if;

  if g_ciDebugOn = 1 then
    DisplayStatus("Starting Conveyor");
  end if;

  -- Start the conveyor.
  if SetDigout(g_ciOnBoardIO, g_ciConveyor, DIO_ON) <> SysOK then
    g_flgFatalError := TRUE;
  end if;

end;


----------------------------------------------------------------------------
-- Procedure Name    : StopConveyor
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : The digital output defined by Conveyor will be turned
--                     off.  The global flag g_flgConveyorOn will be set to
--                     FALSE.
-- Value Parameters  : none
-- Variable Params   : none
-- Side Effects      : Global variable g_flgConveyorOn is set to TRUE
----------------------------------------------------------------------------
procedure StopConveyor;

begin

  if g_ciDebugOn = 1 then
    DisplayStatus("Stopping Conveyor");
  end if;

  if SetDigout(g_ciOnBoardIO, g_ciConveyor, DIO_OFF) <> SysOK then
    g_flgFatalError := TRUE;
  end if;

  g_flgConveyorOn := FALSE;
  g_flgWaitingForClear := FALSE;
end;


----------------------------------------------------------------------------
-- Function Name     : CheckConveyorRunning
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : This function will get called by the process timer
--                     are fairly rapid intervals (every 10 ms or so). We
--                     need to see if it is time to turn off the conveyor.
--                     The global flags are the only thing we need to look
--                     at to determine if the conveyor needs to be shut off.
-- Value Parameters  : none
-- Variable Params   : none
-- Return Parameter  : Returns True if the conveyor is running.
-- Side Effects      : The conveyor I/O may get shut off.
----------------------------------------------------------------------------
function CheckConveyorRunning : boolean;

  boConveyorRunning : boolean := FALSE;
  rWeight : real;

begin

  if g_flgConveyorOn then

    -- We need to check to see if we are waiting for the last piece to clear
    -- the photoeye or for a new piece to break the photoeye.
    if g_flgWaitingForClear then

      if g_flgPhotoEyeMade = FALSE then

        if g_ciDebugOn = 1 then
          DisplayStatus("Photoeye Clear");
        end if;

        -- We were waiting for the photoeye to clear and now it is clear. We
        -- need to reset the flag.
        g_flgWaitingForClear := FALSE;

      end if;

      boConveyorRunning := TRUE;

    else

      -- We are waiting for the photoeye to be made, indicating the next
      -- piece may have broken the beam.
      if g_flgPhotoEyeMade then

        -- The photoeye is made, but we have to check the weight before it
        -- is safe to shut off the conveyor.

        GetGross(g_ciScale, Primary, rWeight);

        -- if the weight isn't over the threshold weigh then we don't
        -- really have a piece on the scale.
        if rWeight > g_rThreshold then
          StopConveyor;
        end if;

      else
        boConveyorRunning := TRUE;
      end if;

    end if;

  end if;

  return boConveyorRunning;

end;


----------------------------------------------------------------------------
-- Procedure Name    : RejectPiece
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : The digital output defined by RejectRam will be turned
--                     on for a short period of time to push the out-of-tol
--                     piece into the reject bin.  A timer is used to turn
--                     the RejectRam off so we don't wait in this procedure.
-- Value Parameters  : none
-- Variable Params   : none
-- Side Effects      : The Reject ram is energized momentarily.
----------------------------------------------------------------------------
procedure RejectPiece;
begin

  if g_ciDebugOn = 1 then
    DisplayStatus("Rejecting Piece");
  end if;

  -- Energize the reject ram.
  if SetDigout(g_ciOnBoardIO, g_ciRejectRam, DIO_ON) <> SysOK then
    g_flgFatalError := TRUE;
  end if;

  -- Set and start a timer that will deactivate the reject ram when the
  -- timer expires.
  if SetTimer(g_ciRejectRamOnTimer, g_ciRamTime) = SysOK then

    if StartTimer(g_ciRejectRamOnTimer) <> SysOK then
      g_flgFatalError := TRUE;
    end if;

  else
    g_flgFatalError := TRUE;
  end if;

end;


----------------------------------------------------------------------------
-- Procedure Name    : GetStableWeight
-- Created By        : RLWS
-- Last Modified on  : 2/6/2002
-- Purpose           : The motion of a scale is checked until it is out of
--                     motion, then the weight is captured.
-- Value Parameters  : iScale is the scale we want to get the weight from.
--                     eUnits is a enumerated type for the type of units.
--                     eMode is an enumerated type for the kind of weight
--                     we want to get, either GROSS, NET, or TARE.
-- Variable Params   : vrWeight, the actual weight is returned in this var.
-- Return Parameter  : The SysCode returned from any calls that don't
--                     return a SysOK.  If everything goes well, a SysOK
--                     is returned.
-- Side Effects      : none
----------------------------------------------------------------------------
function GetStableWeight(iScale : integer; eMode : WeightMode;
                         eUnits : Units; var vrWeight : real) : SysCode;
  iInMotion : integer;
  sysReturnValue : SysCode;

begin

  iInMotion := 1;
  sysReturnValue := SysOK;

  while (iInMotion = 1) and (sysReturnValue = SysOK)
  loop
    sysReturnValue := InMotion(iScale, iInMotion);
  end loop;

  if iInMotion = 0 then

    if g_ciDebugOn = 1 then
      DisplayStatus("Getting Stable Weight");
    end if;

    -- Yes, the scale is out of motion.  Now get the weight.  The type of
    -- weight we want is stored in eMode.
    if eMode = GROSS then
      sysReturnValue := GetGross(iScale, eUnits, vrWeight);
    elsif eMode = TARE then
      sysReturnValue := GetTare(iScale, eUnits, vrWeight);
    elsif eMode = NET then
      sysReturnValue := GetNet(iScale, eUnits, vrWeight);
    end if;

  end if;

  return(sysReturnValue);

end;


-- End of functions and procedures.
----------------------------------------------------------------------------


----------------------------------------------------------------------------
-- Begin event handler definitions here.

----------------------------------------------------------------------------
-- Handler Name      : User1KeyPressed
-- Created By        : RLWS
-- Last Modified on  : 2/6/2002
-- Purpose           : Will prompt the user for the minimum acceptable low
--                     weight for a piece being check-weighed.
--                     User key 1 will be defined in the softkeys menu under
--                     "Features" in configuration mode.  This key's actual
--                     physical location can change since the user can put
--                     it anywhere in the list of softkeys.
-- Side Effects      : The indicator is put into entry mode and the prompt
--                     defined below is displayed.
----------------------------------------------------------------------------
```

```
handler User1KeyPressed;
begin
  if PromptUser("weight: ") = SysOK then
    DisplayStatus("Please enter the minimum piece");
  end if;

  g_eEntryMode := MINVAL;

end;



----------------------------------------------------------------------------
-- Handler Name      : User2KeyPressed
-- Created By        : RLWS
-- Last Modified on  : 2/6/2002
-- Purpose           : Will prompt the user for the maximum acceptable high
--                     weight for a piece being check-weighed.
--                     User key 2 will be defined in the softkeys menu under
--                     "Features" in configuration mode.  This key's actual
--                     physical location can change since the user can put
--                     it anywhere in the list of softkeys.
-- Side Effects      : The indicator is put into entry mode and the prompt
--                     defined below is displayed.
----------------------------------------------------------------------------
handler User2KeyPressed;
begin
  if PromptUser("weight: ") = SysOK then
    DisplayStatus("Please enter the maximum piece");
  end if;

  g_eEntryMode := MAXVAL;

end;



----------------------------------------------------------------------------
-- Handler Name      : User3KeyPressed
-- Created By        : RLWS
-- Last Modified on  : 2/6/2002
-- Purpose           : Will prompt the user for the minimum weight to detect
--                     that a piece is actually on the scale.
--                     User key 3 will be defined in the softkeys menu under
--                     "Features" in configuration mode.  This key's actual
--                     physical location can change since the user can put
--                     it anywhere in the list of softkeys.
-- Side Effects      : The indicator is put into entry mode and the prompt
--                     defined below is displayed.
----------------------------------------------------------------------------
handler User3KeyPressed;
begin
  if PromptUser("threshold weight: ") = SysOK then
    DisplayStatus("Please enter the minimum");
  end if;

  g_eEntryMode := THRESHOLD;

end;



----------------------------------------------------------------------------
-- Handler Name      : UserEntry
-- Created By        : RLWS
-- Last Modified on  : 1/21/2002
-- Purpose           : This event will fire when ever the 920 is in entry
--                     mode and the user presses one of the event keys. So
--                      far, the only event keys defined are EnterKey and
--                      CancelKey.  Do determine which key generated the
--                      event, we call EventKey. Depending on the mode, we
--                      will do certain integrity checks on the entry string
--                      before saving the entered data.
-- Side Effects       : none permanent
----------------------------------------------------------------------------
handler UserEntry;

  strEntry : string := "";
  rValue : real;

begin

  -- We need to find out which key caused the this event.
  if EventKey = EnterKey then

    -- Call GetEntry to get the string entered by the user before the
    -- "ENTER" key was pressed.
    strEntry := GetEntry;

    -- This will remove the numeric entry info and the entry prompt from
    -- the display.
    ClosePrompt;

    -- We need to make sure that they entered something, and that the
    -- something they entered is a numeric value (may contain a decimal).
    if (len(strEntry) > 0) and IsNumeric(strEntry, TRUE) then

      -- Convert the user entry into a real value.
      rValue := StringToReal(strEntry);

      -- Depending what the programmer defined entry mode is, we will save
      -- the value in different globals.
      if g_eEntryMode = MINVAL then
        g_rLowTolerance := rValue;
      elsif g_eEntryMode = MAXVAL then
        g_rHighTolerance := rValue;
      elsif g_eEntryMode = THRESHOLD then
        g_rThreshold := rValue;
      else
        DisplayStatus("Invalid Entry Mode!");
      end if;

    else
      DisplayStatus("Invalid Entry!");
    end if;

  else
    --  With all other keys that generate this event, we should just close
    -- the prompt.
    ClosePrompt;
    DisplayStatus("Entry Cancelled!");
  end if;

  g_eEntryMode := NONE;

end;

----------------------------------------------------------------------------
-- Handler Name       : DiginS0B1Activate
-- Created By         : RLWS
-- Last Modified on   : 2/4/2002
-- Purpose            : This digital input is tied to a photoeye that is
--                      activated whenever a piece is blocking its lazer.
--                      The global g_flgPhotoEyeMade is set to TRUE.
-- Side Effects       : The global g_flgPhotoEyeMade is set to TRUE.
----------------------------------------------------------------------------
```

```
handler DiginS0B1Activate;
begin

  if g_ciDebugOn = 1 then
    DisplayStatus("DigI/O 1 Activate");
  end if;

  g_flgPhotoEyeMade := TRUE;
end;

-----------------------------------------------------------------------------
-- Handler Name      : DiginS0B1Deactivate
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : This digital input is tied to a photoeye that is
--                     deactivated when ever its light is reflected back to
--                     it.  i.e. nothing is blocking it.
--                     The global g_flgPhotoEyeMade is set to FALSE.
-- Side Effects      : The global g_flgPhotoEyeMade is set to FALSE.
-----------------------------------------------------------------------------
handler DiginS0B1Deactivate;
begin

  if g_ciDebugOn = 1 then
    DisplayStatus("DigI/O 1 Deactivate");
  end if;

  g_flgPhotoEyeMade := FALSE;
end;


-----------------------------------------------------------------------------
-- Handler Name      : DiginS0B2Activate
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : This digital input is tied to a "Run" switch that is
--                     maintained on either "Run" or "Pause".  When in the
--                     run position, the input is activated.
--                     The global g_flgStart is set to TRUE.  This will allow
--                     the process to start and proceed until g_flgRunning is
--                     no longer true.
-- Side Effects      : The global g_flgStart is set to TRUE.
-----------------------------------------------------------------------------
handler DiginS0B2Activate;
begin

  if g_ciDebugOn = 1 then
    DisplayStatus("DigI/O 2 Activate");
  end if;

  g_flgStart := TRUE;
end;


-----------------------------------------------------------------------------
-- Handler Name      : DiginS0B2Deactivate
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : This digital input is tied to a "Run" switch that is
--                     maintained on either "Run" or "Pause".  When in the
--                     pause position, the input is deactivated.
--                     The global g_flgRun is set to FALSE.  This will stop
--                     the process.
-- Side Effects      : The global g_flgRun is set to FALSE.
-----------------------------------------------------------------------------
handler DiginS0B2Deactivate;
begin
```

```
  if g_ciDebugOn = 1 then
    DisplayStatus("DigI/O 2 Deactivate");
  end if;

  StopConveyor;

  SetDigout(g_ciOnBoardIO, g_ciRejectRam, DIO_OFF);

  g_flgRunning := FALSE;
end;



----------------------------------------------------------------------------
-- Handler Name      : Timer1Trip
-- Created By        : RLWS
-- Last Modified on  : 2/4/2002
-- Purpose           : Deactivate the Digitial I/O named RejectRam.
-- Side Effects      : The Digitial I/O named RejectRam is deactivated.
----------------------------------------------------------------------------
handler Timer1Trip;
begin

  if g_ciDebugOn = 1 then
    DisplayStatus("Timer 1 Trip");
  end if;

  -- De-energize the reject ram.
  if SetDigout(g_ciOnBoardIO, g_ciRejectRam, DIO_OFF) <> SysOK then
    g_flgFatalError := TRUE;
  end if;
end;



----------------------------------------------------------------------------
-- Handler Name      : Timer2Trip
-- Created By        : RLWS
-- Last Modified on  : 2/5/2002
-- Purpose           : Check the status of the process each time.
-- Side Effects      :
----------------------------------------------------------------------------
handler Timer2Trip;
  rStableWeight : real;
begin

  -- If the fatal error flag has been set, then we want to put up a message
  -- and loop forever.  The program will hang and the operator will have to
  -- restart the indicator.
  while g_flgFatalError
  loop
    DisplayStatus("Fatal System Error!");
  end loop;


  -- We always need to check if we are running.  If we are then we need to
  -- check the state of the conveyor and then get the weight if the piece
  -- is in position, and do other stuff.
  if g_flgRunning then

    -- We need to check to see if the conveyor is not running. If it is
    -- still running then we don't have to do anything until it stops.
    if CheckConveyorRunning = FALSE then

      -- The conveyor is not running.  We need to get the weight and decide
      -- to accept or reject the piece.
      GetStableWeight(g_ciScale, GROSS, Primary, rStableWeight);
```

```
        if (rStableWeight >= g_rThreshold) then

          if     (rStableWeight < g_rLowTolerance)
              or (rStableWeight > g_rHighTolerance) then

            -- This means that the stable weight captured is out of tolerance.
            -- We must reject the part by energizing the reject ram.
            RejectPiece;

          end if;

        end if;

        -- Now we can restart the conveyor and index to the next piece.
        StartConveyor;

      end if;

    else

      -- If we are NOT running then we need to check to see if we should
      -- start because the start flag was set to true when the run switch
      -- was first turned to "RUN".
      if g_flgStart then
        g_flgRunning := TRUE;
        g_flgStart := FALSE;

        StartConveyor;
      end if;

    end if;

  end;


  -- End of event handlers.
  ----------------------------------------------------------------------------



------------------------------------------------------------------------------
-- This chunk of code is the system startup event handler.

begin

  -- Initialize all global variables here.
  g_flgFatalError := FALSE;

  g_flgStart := FALSE;
  g_flgConveyorOn := FALSE;
  g_flgPhotoEyeMade := FALSE;
  g_flgWaitingForClear := FALSE;


  g_eEntryMode := NONE;

  -- Increment the build number every time you make a change to a new version.
  g_iBuild := 10;

  -- Set up a timer to use for energizing and de-energizing the reject ram.
  if SetTimerMode(g_ciRejectRamOnTimer, TimerOneShot) <> SysOK then
    g_flgFatalError := TRUE;
  end if;


  -- Setup a timer to use for checking the status of the process
  if SetTimerMode(g_ciProcessTimer, TimerContinuous) = SysOK then
    if SetTimer(g_ciProcessTimer, g_ciCheckProcTime) <> SysOK then
```

```
      g_flgFatalError := TRUE;
    end if;
  else
    g_flgFatalError := TRUE;
  end if;

  -- Start the process timer.
  if StartTimer(g_ciProcessTimer) <> SysOK then
    g_flgFatalError := TRUE;
  end if;


  -- Set the text for the softkeys.
  SetSoftKeyText(1, "MinVal");
  SetSoftKeyText(2, "MaxVal");
  SetSoftKeyText(3, "Threshold");

  -- Display the version number to the display.
  DisplayVersion;


end CheckWeigher;
-- This name must match the name following "program" keyword at
-- the beginning of the program.
```

5.8 WriteLn Example

```
--
-- Module Description:
-- This file is the iRite source code for a 920i program.
--
-- This program will send some string data out a port using the Write and
-- WriteLn commands to give examples of the different ways to use formatting
-- in these procedures.  Everything is done in the impicitly named
-- ProgramStartup event handler, so the output will be sent out port 4 only
-- when the program is reset, or taken out of the configuration mode.
------------------------------------------------------------------------------

program WriteLnExample;

  g_ciPort : integer := 4;
  g_crVersion : real := 0.03;
  g_csProgramName : string := "WriteLn Example";

begin

  -- This is one way to print out the program name and version. All the
  -- following data will appear on the same line.
  Write(g_ciPort, "Program ");
  Write(g_ciPort, g_csProgramName);
  Write(g_ciPort, "  Version ");
  Write(g_ciPort, RealToString(g_crVersion, 4, 2));
  Write(g_ciPort, Chr$(13));  -- 13 is the ASCII value for a Carriage Return.
  Write(g_ciPort, Chr$(10));  -- 10 is the ASCII value for a Line Feed.

  -- Here is a more concise way to get the exact same output.
  WriteLn(g_ciPort, "Program ", g_csProgramName,  "  Version ",
          g_crVersion:4:2);


  -- Output some blank lines:
  WriteLn(g_ciPort, "");
  WriteLn(g_ciPort, "");
  -- If we want to output data and format it in columns, then we could do
  -- it like this.
  WriteLn(g_ciPort, "Part Number        Description      In Stock    Price");
  WriteLn(g_ciPort, "-----------        -----------      --------    -----");
  WriteLn(g_ciPort, 12345:11, "Head Gasket":19, "Yes":14, 23.79:9:2);
  WriteLn(g_ciPort, 21135:11, "Lug Nuts":19, "Yes":14, 5.49:9:2);
  WriteLn(g_ciPort, 67855:11, "Air Filter":19, "Yes":14, 4.99:9:2);

  -- The blanks after the commas in the next two lines are for program
  -- readablility and don't effect the output.
  WriteLn(g_ciPort, 44512:11,         "PCV Valve":19, "No" :14,  2.27:9:2);
  WriteLn(g_ciPort, 23007:11, "Carb. Rebuild Kit":19, "Yes":14, 41.00:9:2);

end WriteLnExample;
```

Program printed output:

```
Program WriteLn Example  Version 0.03
Program WriteLn Example  Version 0.03


Part Number        Description      In Stock    Price
-----------        -----------      --------    -----
      12345        Head Gasket           Yes    23.79
      21135           Lug Nuts           Yes     5.49
      67855         Air Filter           Yes     4.99
      44512          PCV Valve            No     2.27
      23007  Carb. Rebuild Kit           Yes    41.00
```

5.9 Database Example

```
------------------------------------------------------------------------------
-- Program Name: DatabaseExample
--
-- Copyright 2002 RLWS as an unpublished work.
-- All Rights Reserved.
--
-- The information contained herein is confidential property of RLWS.
-- The use, copying, transfer or disclosure of such information is prohibited
-- except by express written agreement with RLWS.
--
-- First written on March 12, 2002 by RLWS.
--
-- Module Description:
-- This file is the iRite source code for a 920i program.
--
-- This program will stuff some made up data into a database when the "Default
-- DB" user key is pressed. The "Delete" user key is used to delete the
-- selected record.  "ID Sort" will sort all the records by ID (this is the
-- default).  The "Name Sort" key will sort the data by the db.Last field. And
-- finally the "Age Sort" will sort the data by the db.Age field.  The ability
-- to change the age and department of an employee is also available through
-- the "Change Age" and "Change Dept" keys.
------------------------------------------------------------------------------

program DatabaseExample;

#include common.iri
#include numtools.iri

  -- Create an enumeration to keep track of how the records are sorted.
  type SortType is (SortID, SortNAME, SortAGE);
  enumSort : SortType := SortID;

  -- Create an enumeration to keep track of what type of data entry we have.
  type EntryType is (NoEntry, AgeEntry, DeptEntry);
  enumEntry : EntryType := NoEntry;


  -- Widget Aliases
  lblFirst : constant integer := 2;
  lblLast  : constant integer := 3;
  lblInit  : constant integer := 4;
  lblID    : constant integer := 5;
  lblDept  : constant integer := 6;
  lblAge   : constant integer := 7;
  lblTitle : constant integer := 8;
  lblNames : constant integer := 9;
  llbIDDpt : constant integer := 10;

  -- Database created in iRev Database Table Editor.
  type EmployeeDatabase is database ("Employee")
     Last     : string;
     First    : string;
     Initial  : string;
     ID       : integer;
     Age      : integer;
     Dept     : string;
  end database;

  --Database Field Aliases.
  Employee_Last     : constant integer := 1;
  Employee_First    : constant integer := 2;
  Employee_Initial  : constant integer := 3;
  Employee_ID       : constant integer := 4;
  Employee_Age      : constant integer := 5;
  Employee_Dept     : constant integer := 6;
```

```
g_crVersion : real := 1.21;
g_csProgramName : string := "Database Example";

g_DBWork : EmployeeDatabase;


procedure CreateSampleData;
begin

  -- This procedure will add some sample data to the database.

  g_DBWork.First := "Samual";
  g_DBWork.Initial := "J";
  g_DBWork.Last := "Johnson";
  g_DBWork.ID := 1;
  g_DBWork.Age := 61;
  g_DBWork.Dept := "Sales";
  g_DBWork.Add;

  g_DBWork.First := "Peter";
  g_DBWork.Initial := "P";
  g_DBWork.Last := "Parker";
  g_DBWork.ID := 2;
  g_DBWork.Age := 30;
  g_DBWork.Dept := "Security";
  g_DBWork.Add;

  g_DBWork.First := "James";
  g_DBWork.Initial := "L";
  g_DBWork.Last := "Hetfield";
  g_DBWork.ID := 3;
  g_DBWork.Age := 48;
  g_DBWork.Dept := "Shipping";
  g_DBWork.Add;

  g_DBWork.First := "Jill";
  g_DBWork.Initial := "K";
  g_DBWork.Last := "Bowe";
  g_DBWork.ID := 4;
  g_DBWork.Age := 23;
  g_DBWork.Dept := "Engineering";
  g_DBWork.Add;

  g_DBWork.First := "Chris";
  g_DBWork.Initial := "M";
  g_DBWork.Last := "Anderson";
  g_DBWork.ID := 5;
  g_DBWork.Age := 19;
  g_DBWork.Dept := "Sales";
  g_DBWork.Add;

  g_DBWork.First := "Chuck";
  g_DBWork.Initial := "X";
  g_DBWork.Last := "Williams";
  g_DBWork.ID := 6;
  g_DBWork.Age := 29;
  g_DBWork.Dept := "Service";
  g_DBWork.Add;

  g_DBWork.First := "Lester";
  g_DBWork.Initial := "T";
  g_DBWork.Last := "Smith";
  g_DBWork.ID := 7;
  g_DBWork.Age := 54;
  g_DBWork.Dept := "Service";
  g_DBWork.Add;
  g_DBWork.First := "Tina";
  g_DBWork.Initial := "B";
  g_DBWork.Last := "Jenkins";
  g_DBWork.ID := 8;
  g_DBWork.Age := 37;
  g_DBWork.Dept := "Shipping";
  g_DBWork.Add;

  g_DBWork.First := "Susan";
  g_DBWork.Initial := "H";
  g_DBWork.Last := "Milliford";
  g_DBWork.ID := 9;
  g_DBWork.Age := 41;
  g_DBWork.Dept := "Security";
  g_DBWork.Add;

  g_DBWork.First := "John";
  g_DBWork.Initial := "A";
  g_DBWork.Last := "Krueger";
  g_DBWork.ID := 10;
  g_DBWork.Age := 59;
  g_DBWork.Dept := "Engineering";
  g_DBWork.Add;

end;


procedure DisplayDBRecord(dbRec : EmployeeDatabase);
begin

  -- This procedure will display all the fields of the database record
  -- passed in the record dbRec.

  SetLabelText(lblFirst, g_DBWork.First);
  SetLabelText(lblInit, g_DBWork.Initial);
  SetLabelText(lblLast, g_DBWork.Last);
  SetLabelText(lblID, IntegerToString(g_DBWork.ID, 2));
  SetLabelText(lblAge, IntegerToString(g_DBWork.Age, 2));
  SetLabelText(lblDept, g_DBWork.Dept);

end;

procedure RedisplayRecord(iID : integer);

  sysReturnVal : SysCode;

begin

  -- Instead of using the FindFirst function, which would require a
  -- another sort afterwords, we will use the GetFirst and GetNext
  -- functions to step through all the SORTED records until we find
  -- the one we want to display.
  sysReturnVal := g_DBWork.GetFirst;

  while (sysReturnVal = sysOK) and (g_DBWork.ID <> iID)
  loop

    sysReturnVal := g_DBWork.GetNext;

  end loop;

  if sysReturnVal = sysOK then

    DisplayDBRecord(g_DBWork);

  end if;

end;
```

```
function SortEmployees(enumSortType : SortType; iDisplay : integer) : SysCode;

  sysReturnStatus : SysCode;

begin

  if enumSortType = SortID then

    -- They want to sort the database by the ID field. So we pass the alias
    -- of the field we want to sort by to the Sort function.
    sysReturnStatus := g_DBWork.Sort(Employee_ID);

  elsif enumSortType = SortName then

    -- They want to sort the database by the Name field. So we pass the
    -- alias of the field we want to sort by to the Sort function.
    sysReturnStatus := g_DBWork.Sort(Employee_Last);

  elsif enumSortType = SortAge then

    -- They want to sort the database by the Age field. So we pass the alias
    -- of the field we want to sort by to the Sort function.
    sysReturnStatus := g_DBWork.Sort(Employee_Age);

  end if;


  if (sysReturnStatus = SysOK) and iDisplay then

    -- The data is resorted so get the first record.
    if g_DBWork.GetFirst = SysOk then

      -- This procedure will display all the fields of the database record passed
      -- in the record.
      DisplayDBRecord(g_DBWork);

    end if;
  end if;

  return sysReturnStatus;

end;



handler NavLeftKeyPressed;
begin
  -- The left arrow navigation key will always look for a previous record in
  -- the DB.
  if g_DBWork.GetPrev = SysOK then
    DisplayDBRecord(g_DBWork);
  else
    -- If we couldn't find the previous, then we should at least display the
    -- first record.  This could mean we were already at the first.
    if g_DBWork.GetFirst = SysOK then
      DisplayDBRecord(g_DBWork);
    end if;
  end if;

end;
```

```
handler NavRightKeyPressed;
begin
  -- The right arrow navigation key will always look for the next record
  -- in the DB.
  if g_DBWork.GetNext = SysOK then
    DisplayDBRecord(g_DBWork);
  else
    -- If we couldn't find the next, then we should at least display the
    -- last record.  This could mean we were already at the last.
    if g_DBWork.GetLast = SysOK then
      DisplayDBRecord(g_DBWork);
    end if;
  end if;

end;


-- Default the database.
handler User1KeyPressed;
begin

  -- This next lines clears all records from the database.
  g_DBWork.Clear;

  -- This procedure will add some sample data to the database.
  CreateSampleData;

  -- The next line will get the very first record in the database and put the
  -- results in g_DBWork.
  if g_DBWork.GetFirst = SysOk then

    -- This procedure will display all the fields of the database record passed
    -- in the record.
    DisplayDBRecord(g_DBWork);

  end if;

end;


-- Delete the current record from the database.
handler User2KeyPressed;

  iThisEmployeeID : integer;
  iNextEmployeeID : integer;
  iDeleteOK : integer := 0;
  iGetNewLast : integer := 0;
  sysReturnVal : SysCode;

begin

  -- We want to save the employee ID of the record to be deleted incase we
  -- move from this record and have to get back to it.
  iThisEmployeeID := g_DBWork.ID;

  -- After a delete, we will have to resort.  After resorting, we will lose
  -- our position.  It would be nice if after deleting a record, the "next"
  -- record is displayed.  This will give the appearance of the data
  -- compressing.  If we delete the last record, then we should display the
  -- new "last" record.  If we delete the only record, then we should clear
  -- out the label widgets on the screen.

  if g_DBWork.GetNext = SysOk then

    -- We got the next logical record so we need to remember the ID of this
    -- record.
    iNextEmployeeID := g_DBWork.ID;
    -- We should be able to call GetPrev to get us back to the original
    -- record and then compare it with the one we started with.
    if (g_DBWork.GetPrev = SysOk) and (g_DBWork.ID = iThisEmployeeID) then

      -- Set DeleteOK flag to true.
      iDeleteOK := 1;

    else

      -- We have major problems.  We shouldn't be here.  But it we are we
      -- can still get the original record by specifically searching for the
      -- matching ID.
      g_DBWork.ID := iThisEmployeeID;

      if g_DBWork.FindFirst(Employee_ID) = SysOK then
        -- We found the original (assume only one with unique ID).  So Set
        -- DeleteOK flag to true.
        iDeleteOK := 1;
      end if;
    end if;

  else

    -- The GetNext call failed.  Maybe we were at the last already.  We
    -- could get the last and make sure it is the record we had
    -- originally before deleting it.

    -- We should be able to call GetLast to get us back to the original
    -- record and then compare it with the one we started with.
    if (g_DBWork.GetLast = SysOk) and (g_DBWork.ID = iThisEmployeeID) then

      -- Set DeleteOK flag to true.
      iDeleteOK := 1;
      iGetNewLast := 1;
    else

      -- We have major problems.  We shouldn't be here.  But it we are we
      -- can still get the original record by specifically searching for the
      -- matching ID.
      g_DBWork.ID := iThisEmployeeID;

      if g_DBWork.FindFirst(Employee_ID) = SysOK then
        -- We found the original (assume only one with unique ID).  So Set
        -- DeleteOK flag to true.
        iDeleteOK := 1;
      end if;
    end if;

  end if;

  if iDeleteOK = 1 then

    -- The next line will try to delete the current record stored in g_DBWork.
    if g_DBWork.Delete = SysOk then

      -- Anytime a delete operation is performed, the database needs to be
      -- resorted.  So call the sort function with the appropriate sort mode.
      SortEmployees(enumSort, 0);

      if iGetNewLast = 1 then

        if g_DBWork.GetLast = SysOk then

          DisplayDBRecord(g_DBWork);

        else
```

```
          -- If we get here, then the most likely reason is we deleted the
          -- last record in the database.  We need to clear out g_DBWork and
          -- then display it.  This will clear out the label widgets on the
          -- screen.
          g_DBWork.First := "";
          g_DBWork.Initial := "";
          g_DBWork.Last := "";
          g_DBWork.ID := 0;
          g_DBWork.Age := 0;
          g_DBWork.Dept := "";

          DisplayDBRecord(g_DBWork);

        end if;

      else

        -- Instead of using the FindFirst function, which would require a
        -- another sort afterwords, we will use the GetFirst and GetNext
        -- functions to step through all the SORTED records until we find
        -- the one we want to display.
        sysReturnVal := g_DBWork.GetFirst;

        while (sysReturnVal = sysOK) and (g_DBWork.ID <> iNextEmployeeID)
        loop

          sysReturnVal := g_DBWork.GetNext;

        end loop;

        if sysReturnVal = sysOK then

          DisplayDBRecord(g_DBWork);

        end if;
      end if;
    end if;
  end if;

end;



-- Sort all database records by the ID field.
handler User3KeyPressed;
begin

  enumSort := SortID;
  SortEmployees(enumSort, 1);

end;



-- Sort all database records by the Last Name field.
handler User4KeyPressed;
begin

  enumSort := SortName;
  SortEmployees(enumSort, 1);

end;



-- Sort all database records by the Age field.
handler User5KeyPressed;
begin
```

```
  enumSort := SortAge;

  SortEmployees(enumSort, 1);

end;



-- Prompt the user to update the db.Age field of the current record.
handler User6KeyPressed;
begin

  -- We have to disable the handlers for the left and right navigation keys
  -- since we need to use the default functionality for the entry dialog.
  DisableHandler(NavLeftKeyPressed);
  DisableHandler(NavRightKeyPressed);

  enumEntry := AgeEntry;
  PromptUser("Age: ");

end;



-- Prompt the user to update the db.Dept field of the current record.
handler User7KeyPressed;
begin

  -- We have to disable the handlers for the left and right navigation keys
  -- since we need to use the default functionality for the entry dialog.
  DisableHandler(NavLeftKeyPressed);
  DisableHandler(NavRightKeyPressed);

  enumEntry := DeptEntry;
  PromptUser("Dept: ");

end;



-- Get the string that was entered before the ENTER key was pressed
handler UserEntry;

  strEntry : string := "";
  flgUpdate : boolean := FALSE;
  iEmployeeID : integer;

begin

  -- We have to enable the handlers for the left and right navigation keys.
  EnableHandler(NavLeftKeyPressed);
  EnableHandler(NavRightKeyPressed);

  -- The function EventKey will return the key that closed the dialog box.
  -- We are only concerned with the case where the EnterKey was pressed.
  if EventKey = EnterKey then

    -- We call the function GetEntry to get the string of numeric or alpha
    -- characters entered.
    strEntry := GetEntry;

    -- We can now close the prompt box since we have the user entry.  We
    -- could validate the input first before deciding to close the dialog
    -- box but we will just not update the record if the entry is invalid
    -- for simplicities sake in this example.
    ClosePrompt;
```

```
      -- Check to see if any characters were entered.
      if len(strEntry) > 0 then

        if enumEntry = AgeEntry then

          -- Must be numeric with no decimal places.
          if IsNumeric(strEntry, FALSE) then

            g_DBWork.Age := StringToInteger(strEntry);

            flgUpdate := TRUE;

          end if;

        elsif enumEntry = DeptEntry then

          -- Must be 20 characters or less.
          if Len(strEntry) <= 20 then

            g_DBWork.Dept := strEntry;

            flgUpdate := TRUE;

          end if;
        end if;

        if flgUpdate then

          iEmployeeID := g_DBWork.ID;

          g_DBWork.Update;

          -- We now need to re-sort the data and re-display the current
          -- record to reflect the new data.
          SortEmployees(enumSort, 0);

          RedisplayRecord(iEmployeeID);

        end if;

      end if;

    else

      ClosePrompt;

    end if;

  end;


begin  -- This is the ProgramStartup handler.

  -- Display program name and version.
  DisplayStatus(g_csProgramName + "  Ver " + RealToString(g_crVersion, 4, 2));


  -- The next line will get the very first record in the database and put the
  -- results in g_DBWork.
  if g_DBWork.GetFirst = SysOk then

    -- This procedure will display all the fields of the database record passed
    -- in the record.
    DisplayDBRecord(g_DBWork);

  end if;
  -- The next five lines set the text for each of the softkeys.
  SetSoftKeyText(1, "Default DB");
  SetSoftKeyText(2, "Delete");
  SetSoftKeyText(3, "ID Sort");
  SetSoftKeyText(4, "Name Sort");
  SetSoftKeyText(5, "Age Sort");
  SetSoftKeyText(6, "Change Age");
  SetSoftKeyText(7, "Change Dept");

end DatabaseExample;
```

6.0 Appendix

6.1 Event Handlers

Handler	Description
ClearKeyPressed	Runs when the CLR key on the numeric keypad is pressed
Cmd*x*Handler	Runs when an F#*x* serial command is received on a serial port, where *x* is the F# command number, 1–32. The comunications port number receiving the command and the text associated with the F#*x* command can be returned from the Cmd*x*Handler using the EventPort and EventString functions (see page 36).
DiginS*x*B*y*Activate	Runs when the digital input assigned to slot *x*, bit *y* is activated. Valid bit assignments for slot 0 are 1–4; valid bit assignments for slots 1 through 14 are 1–24.
DiginS*x*B*y*Deactivate	Runs when the digital input assigned to slot *x*, bit *y* is activated. Valid bit assignments for slot 0 are 1–4; valid bit assignments for slots 1 through 14 are 1–24.
DotKeyPressed	Runs when the decimal point key on the numeric keypad is pressed
EnterKeyPressed	Runs when the ENTER key on the front panel is pressed
GrossNetKeyPressed	Runs when the GROSS/NET key is pressed
KeyPressed	Runs when any front panel key is pressed. Use the EventKey function within this handler to determine which key caused the event.
MajorKeyPressed	Runs when any of the five preceding major keys is pressed. Use the EventKey function within this handler to determine which key caused the event.
NavDownKeyPressed	Runs when the DOWN navigation key is pressed
NavKeyPressed	Runs when any of the navigation cluster keys (including ENTER) is pressed. Use the EventKey function within this handler to determine which key caused the event.
NavLeftKeyPressed	Runs when the LEFT navigation key is pressed
NavRightKeyPressed	Runs when the RIGHT navigation key is pressed
NavUpKeyPressed	Runs when the UP navigation key is pressed
NumericKeyPressed	Runs when any key on the numeric keypad (including CLR or decimal point) is pressed. Use the EventKey function within this handler to determine which key caused the event.
N*x*KeyPressed	Runs when a numeric key is pressed, where *x*=the key number 0–9
Port*x*CharReceived	Runs when a character is received on port *x*, where *x* is the port number, 1–32. Use the EventChar function within these handlers to return a one-character string representing the character that caused the event.
PrintKeyPressed	Runs when the PRINT KEY is pressed
ProgramStartup	Runs when the indicator is powered-up or when exiting setup mode
SoftKeyPressed	Runs when any softkey is pressed. Use the EventKey function within this handler to determine which key caused the event.
Soft*x*KeyPressed	Runs when softkey *x* is pressed, where x=the softkey number, 1–5, left to right
SP*x*Trip	Runs when setpoint *x* is tripped, where *x* is the setpoint number, 1–100)
TareKeyPressed	Runs when the TARE key is pressed
Timer*x*Trip	Runs when timer *x* is tripped, where *x* is the timer number, 1–32
UnitsKeyPressed	Runs when the UNITS key is pressed
User*x*KeyPressed	Runs when a user-defined softkey is pressed, where *x* is the user-defined key number, 1–10
UserEntry	Runs when the ENTER key or Cancel softkey is pressed in response to a user prompt
ZeroKeyPressed	Runs when the ZERO key is pressed

Table 6-1. 920i Event Handlers

6.2 Compiler Error Messages

Error Messages	Cause (Statement Type)
Argument is not a handler name	Enable/disable handler
Arguments must have intrinsic type	Write/Writeln
Array bound must be greater than zero	Type declaration
Array bound must be integer constant	Type declaration
Array is too large	Type declaration
Conditional expression must evaluate to a discrete data type	If/while statement
Constant object cannot be stored	Object declaration
Constant object must have initializer	Object declaration
Exit outside all loops	Exit statement
Expected array reference	Subscript reference
Expected object or function reference	Qualifying expression
Expression must be numeric	For statement
Expression type does not match declaration	Initializer
Function name overloads handler name	Function declaration uses name reserved for handler
Handlers may not be called	Procedure/function call
Identifier already declared in this scope	All declarations
Illegal comparison	Boolean expression
Index must be numeric	Subscript reference
Invalid qualifier	Qualifying expression
Loop index must be integer type	For statement
Name is not a subprogram	Procedure/function call
Name is not a valid handler name	Handler declaration
Not a member of qualified type	Qualifying expression
Only a function can return a value	Procedure/handler declaration
Operand must be integer or enumeration type	Function or procedure call
Operand must be integer type	Logical expression
Operand type mismatch	Expression
Parameter is not a valid l-value	Procedure/function call
Parameter type mismatch	Procedure/function call
Parameters cannot be declared constant	Subprogram declaration
Port parameter must be integer type	Write/Writeln
Procedure name overloads handler name	Procedure declaration uses name reserved for handler
Procedure reference expected	Subprogram invocation
Record fields cannot be declared constant	Type declaration
Record fields cannot be declared stored	Type declaration
Reference is not a valid assignment target	Assignment statement
Return is only allowed in a subprogram	Startup body
Return type mismatch	Return statment
Step value must be constant	For statement
Subprogram invocation is missing parameters	Procedure/function call

Table 6-2. iRite Compiler Error Messages

Error Messages	Cause (Statement Type)
Syntax error	Any statement
Cannot find system files	Internal error
Compiler error — Context stack error	Internal error
Too many names declared in this context	Any declaration
Operand must be numeric	Numeric operators
Subprogram reference expected	Procedure/function call
Type mismatch in assignment	Assignment statement
Type reference expected	User-defined type name
Undefined identifier	Identifier not declared
VAR parameter type must match exactly	Procedure/function call
Wrong number of array subscripts	Subscript reference
Wrong number of parameters	Procedure/function call

Table 6-2. iRite Compiler Error Messages