920i®

*Programmable HMI Indicator/Controller*

*Version 5.00*

irite

# Programming Reference


RICE LAKE®
WEIGHING SYSTEMS
To be the best by every measure®

67888

# Contents





Version 5.00, April 2012

# About This Manual

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

This manual applies to Version 4.01 and later of the *920i* indicator software and should be used in conjunction with the Version 4.01 *920i Installation Manual*, PN 67887. See that manual for detailed descriptions of indicator capability and operation.


***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.ricelake.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* has 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, 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 any 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?

Templates and sample programs are available from RLWS to provide the skeleton of a working program. Once you have the *iRev* Editor open, you are ready to start writing a program. *iRite* source files are named with the *.src* extension.

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 .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.

## 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 whenever 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 whatever 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.

## 1.6 Summary of Changes

This manual has been updated to include APIs and handlers available in Version 4.01 of the *920i* indicator software. Changes to this manual include the following:

- The list of built-in types described in the system.src file has been updated (see Section 4.0 on page 28).
- The API reference in Section 5.0 on page 31 includes several new APIs.
- The examples section of the previous edition has been removed. Please see the *920i* Support page at **www.ricelake.com** for downloadable program examples.
- Several new sections have been added to the appendix (see Section 6.0), including: "iRev Database Operations" on page 88, "Fieldbus User Program Interface" on page 89, "Program to Retrieve 920i Hardware Configuration" on page 90, and "920i User Graphics" on page 92.

# 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:

```
01   program HelloWorld;
02
03     handler KeyPressed;
04     begin
05       DisplayStatus("Hello, world!");
06     end;
07
08   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 than 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 $0 \ldots 2^{31} - 1$. *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, it's 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
```

***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
```

***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.


***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>
  ;
```

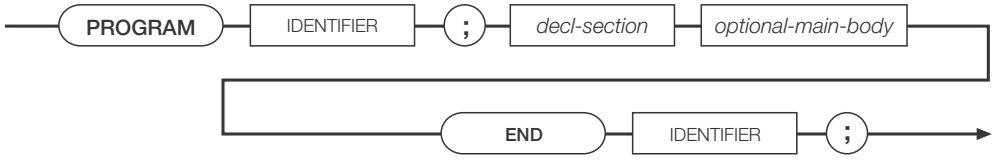


*Figure 3-1. Program Statement Syntax*

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>
   ;
```

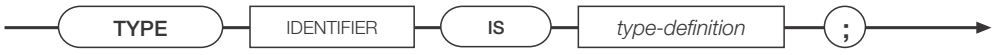


*Figure 3-2. Type Declaration Syntax*

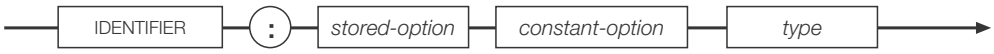


*Figure 3-3. Identifier Syntax*

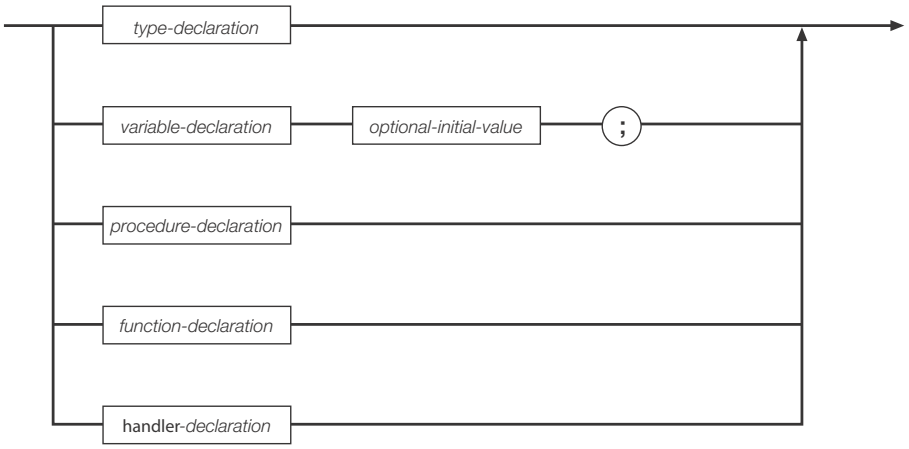


*Figure 3-4. Type Declaration Syntax*

### 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> ';'
  ;
```

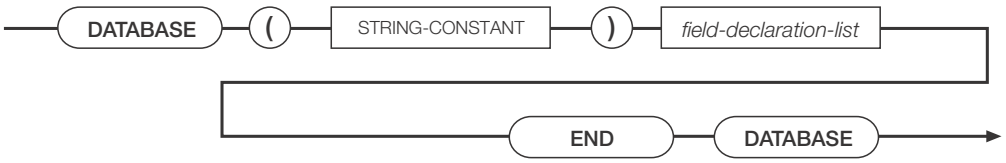


*Figure 3-6. Database Type Definition Syntax*

**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;
```


*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 ';'
   ;
```

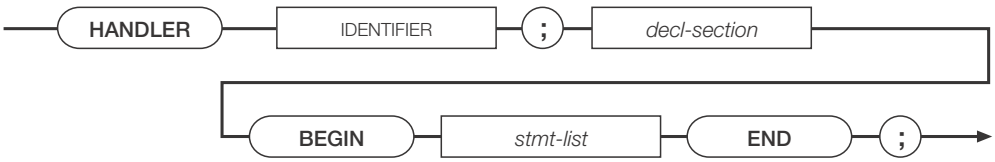


*Figure 3-8. Handler Declaration Syntax*

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
  ;
```

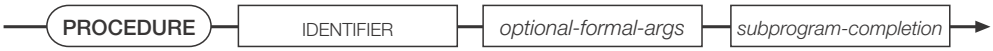


*Figure 3-9. Procedure Declaration Syntax*

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 ';'
  ;
```

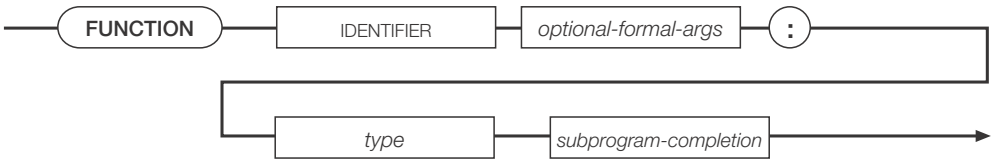


*Figure 3-10. Function Declaration Syntax*

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

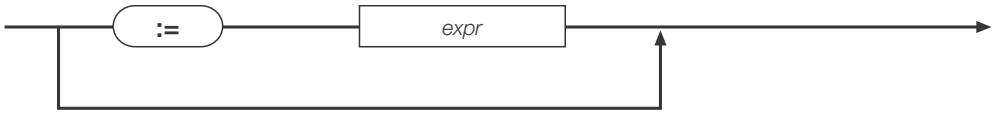


*Figure 3-11. Assignment Statement Syntax*

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

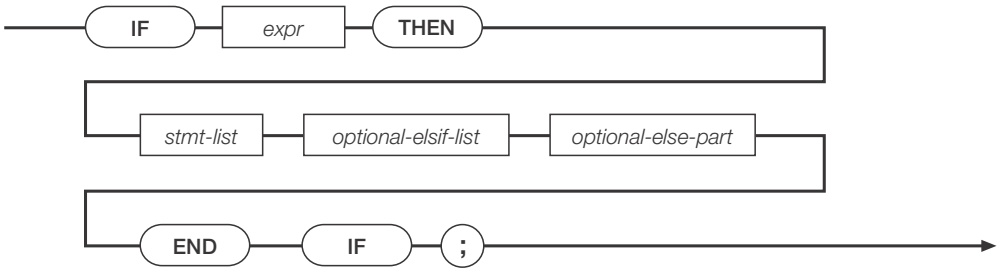


*Figure 3-12. If Statement Syntax*

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;
```

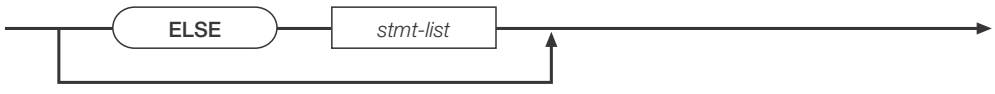


*Figure 3-13. Optional Else Statement Syntax*

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**.

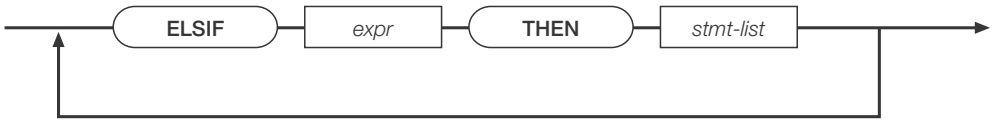


*Figure 3-14. Optional Else-If Statement Syntax*

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

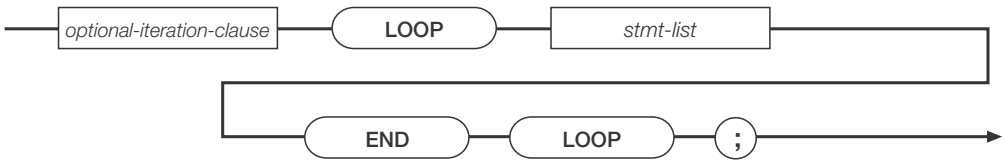


*Figure 3-15. Loop Statement Syntax*

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;
```

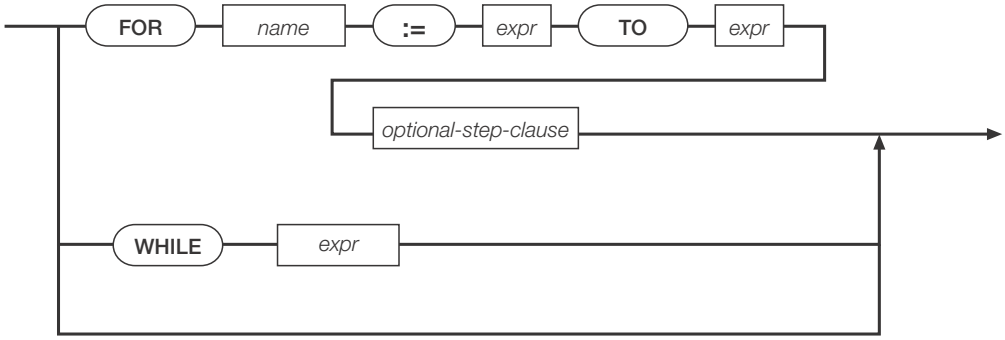


*Figure 3-16. Optional Loop Iteration Clause Syntax*

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;
```

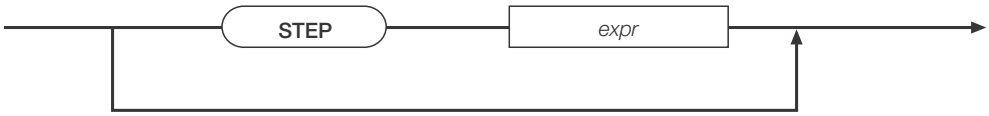


*Figure 3-17. Optional Step Clause Syntax*

**Note** ***Use caution when designing loops to ensure that you don't create an infinite loop. If your program encounters an infinite loop, only the loop will run; subsequent queued events will not be run.***

### 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 than 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 25 for the **exit** statement in action.


*Figure 3-19. Exit Statement Syntax*

# 4.0 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,
                  SysInvalidChecksum,
                  SysOk,
                  SysNoFileSystemFound,
                  SysPortbusy,
                  SysFileNotFound,
                  SysDirectoryNotFound,
                  SysFileExists,
                  SysInvalidFileFormat,
                  SysInvalidMode,
                  SysBadFilename, (over 8 characters)
                  SysMediaChanged,
                  SysNoFileOpen,
                  SysEndOfFile);
type Mode is (GrossMode, NetMode, TareMode);
type Units is (Primary, Secondary, Tertiary);
type TareType is (NoTare, PushButtonTare, KeyedTare);
type BatchingMode is (Off, Manual, Auto);
type BatchStatus is (BatchComplete, BatchStopped, BatchRunning, BatchPaused);
-- PrintFormat must match the definitions in print.h in the core software.
type PrintFormat is (GrossFmt, NetFmt,
 AuxFmt,
 TrWInFmt, TrRegFmt, TrWOutFmt,
 SPFmt,
 AccumFmt, AlertFmt,
```

```
AuxFmt1, AuxFmt2, AuxFmt3, AuxFmt4, AuxFmt5,
AuxFmt6, AuxFmt7, AuxFmt8, AuxFmt9, AuxFmt10,
AuxFmt11, AuxFmt12, AuxFmt13, AuxFmt14, AuxFmt15,
AuxFmt16, AuxFmt17, AuxFmt18, AuxFmt19, AuxFmt20 );
--TimerMode must match the definitions in API_timer.c in the core software.
type TimerMode is (TimerOneShot, TimerContinuous, TimerDigoutON, TimerDigoutOFF);
type OnOffType is (VOff, VOn);
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),
              TimeDateKey,     WeighInKey,      WeighOutKey,     ID_EntryKey,
              DisplayTareKey,  TruckRegsKey,    DisplayAccumKey, ScaleSelectKey,
              DisplayROCKey,   SetpointKey,     BatchStartKey,   BatchStopKey,
              BatchPauseKey,   BatchResetKey,   DiagnosticsKey,  ContactsKey,
              DoneKey,         TestKey);
type DT Component is (DateTimeYear,
                      DateTimeMonth,
                      DateTimeDay,
                      DateTimeHour,
                      DateTimeMinute,
                      DateTimeSecond);
type BusImage is array[32] of integer;
type BusImageReal is array[32] of real;
type DataArray is array[300] of real;
type DisplayImage is array[2402] of integer;
type Color_type is (White, Black);
-- UnitType must match the core definitions in cfg.h
type UnitType is  (pound, kilogram, gram, ounce, short_ton,
                   metric_ton, grain, troy_ounce, troy_pound,
                   long_ton, custom, units_off, none);
type ExtFloatArray is array[5] of integer;
type WgtMsg is array[12] of integer;
-- This enumeration must match the RESP_CODE_* definitions in core code dtable.h.
type HW_type is (  NoCard,
                   DualSerial,
                   DualAtoD,
                   SingleAtoD,
                   AnalogOut,
                   DigitalIO,
                   Pulse,
                   Memory,
                   reservedCard,
                   DeviceNet,
                   Profibus,
                   Ethernet,
                   ABRIO,
                   BCD,
                   DSP2000,
                   AnalogInput,
                   ControlNet
```

```
                    DualAnalogOut );
-- Array size must match MAX_SLOTS in core code common.h.
type HW_array_type is array[14] of HW_type;
-- Graph type must match definitions in graphing.h.
type GraphType is ( Line, Bar, XY );
-- Decimal_Type must match enumeration in cfg.h.
type Decimal_type is (  DP_8_888888,
                        DP_88_88888,
                        DP_888_8888,
                        DP_8888_888,
                        DP_88888_88,
                        DP_888888_8,
                        DP_8888888,
                        DP_8888880,
                        DP_8888800,
                        DP_DEFAULT );
-- IQValType must match the enumeration in iQube.h in the core software.
type IQValType is ( IQSys, IQPlat, IQRawLC, IQCCorrLC, IQZeroLC, IStatLC
IQ2ScaleWt, IQ2StatusLC );
type USBDeviceType is(USBNoDevice, USBHostPC, USBPrinter1, USBPrinter2,
USBKeyboard, USBFileSystem);
type FileAccessMode is(FileCreate, FileAppend, FileRead);
type FileLineTermination is(FileCRLF, FileCR, FileLF);
```

### 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;
```

# 5.0 API Reference

This section lists the application programming interfaces (APIs) used to program the *920i* indicator. Functions are grouped according to the kinds of operations they support.

***If you are unsure whether your version of software supports a given API, check the system.src file to see if the API is present.***

## 5.1 Scale Data Acquisition

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

### 5.1.1 Weight Acquisition

#### CloseDataRecording

Turns off data recording started with InitDataRecording. This procedure removes all connections to the data recording function. To restart data recording, use the InitDataRecording function.

Method Signature:

```
procedure CloseDataRecording (scale_no : Integer);
```

Parameters:

| | | |
|---|---|---|
| [in] | `scale_no` | Scale number |

#### GetDataRecordSize

Returns the number of data points recorded in the user-specified data array.

Method Signature:

```
function GetDataRecordSize (scale_no : Integer) : Integer;
```

Parameters:

| | | |
|---|---|---|
| [in] | `scale_no` | Scale number |

SysCode values returned:

| | |
|---|---|
| *`number`* | The SysCode contains the number of data points recorded. |

#### 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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | `S` | Scale number |
| [in] | `U` | Units (Primary, Secondary, Tertiary) |
| [out] | `W` | Gross weight |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | W | Net weight |

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;
```

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | W | Tare weight |

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);
```

### InitDataRecording

InitDataRecording allows raw weights to be stored to a user program-specified array on each iteration of the scale processor. Recording begins when the start_sp is satisfied and ends when the stop_sp is satisfied. InitDataRecording specifies the data array used for the recording, scale number, and the start and stop setpoint numbers.


***If the setpoint conditions return to the start conditions (start_sp satisfied, stop_sp not satisfied), recording will continue at the array location where it left off. Thus, a continuous batch will need to call CloseDataRecording to stop recording, then call InitDataRecording to restart data recording at the beginning of the array.***

Method Signature:

```
function InitDataRecording (data : DataArray; scale_no : Integer; start_sp :
Integer; stop_sp : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `data` | Data array name |
| [in] | `scale_no` | Scale number |
| [in] | `start_sp` | Start setpoint number |
| [in] | `stop_sp` | Stop setpoint number |

SysCode values returned:

| | |
|---|---|
| `SysRequestFailed` | The function did not complete. |
| `SysOK` | The function completed successfully. |

### 5.1.2 Tare Manipulation

#### AcquireTare

Acquires a pushbutton tare from scale `S`.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | `S` | Scale number |

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);
```

#### ClearTare

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | `S` | Scale number |

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);
```

#### GetTareType

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | `S` | Scale number |
| [out] | `T` | Tare type |

TareType values returned:

| | |
|---|---|
| NoTare | There is no tare value associated with the specified scale. |
| PushbuttonTare | The current tare was acquired by pushbutton. |
| KeyedTare | 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. |

Example:

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

#### SetTare

Sets the tare weight for the specified channel.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [in] | W | Tare weight |

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);
```

### 5.1.3 Rate of Change

#### GetROC

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | R | Rate of change value |

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);
```

### 5.1.4 Accumulator Operations

#### ClearAccum

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |

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);
```

#### GetAccum

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | W | Accumulated weight |

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);
```

#### GetAccumCount

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | N | Accumulator count |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | D | Accumulator date |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | T | Accumulator time |

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);
```

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | W | Average accumulator weight |

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);
```

#### GetUnitsString

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | V | Current display units string |

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);
```

#### SetAccum

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [in] | W | Accumulator value |

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);
```

### 5.1.5 Scale Operations

#### CurrentScale

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

Method Signature:
```
function CurrentScale : Integer;
```

Example:
```
ScaleNumber : Integer;
...
ScaleNumber := CurrentScale;
```

### GetMode

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | U | Current display mode |

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);
```

### GetUnits

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | U | Current display units |

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);
```

### 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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | V | Center-of-zero value |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | V | In-motion value |

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);
```

### 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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | V | In-range value |

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);
```

### SelectScale

Sets scale S as the current scale.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |

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);
```

### SetMode

Sets the current display mode on scale S to M.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | M | Scale mode |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Scale units |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |

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);
```

### 5.1.6 A/D and Calibration Data

#### GetFilteredCount

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | C | Current filtered A/D count |

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);
```

#### GetLCCD

Sets V to the calibrated deadload count for scale S.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | V | Deadload count |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | V | Calibrated span count |

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. |

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | C | Current raw A/D count |

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);
```

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | V | Test weight value |

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.

Method Signature:
```
function GetZeroCount (S : Integer; VAR V : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | V | Zero count |

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. |

## 5.2 System Support

#### Date$
Returns a string representing the system date contained in DT.

Method Signature:
```
function Date$ (DT : DateTime) : String;
```

#### DisableHandler
Disables the specified event handler. See Section 6.1 on page 85 for a list of handlers.

Method Signature:
```
procedure DisableHandler (handler);
```

### 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.

Method Signature:
```
function DisplayIsSuspended : Integer;
```

### EnableHandler
Enables the specified event handler. See Section 6.1 on page 85 for a list of handlers.

Method Signature:
```
procedure EnableHandler (handler);
```

### 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 85 for information about the *PortxCharReceived* event handler.

Method Signature:
```
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 4.0 on page 28 for a definition of the Keys data type.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
function EventString : String;
```

### GetConsecNum
Returns the value of the consecutive number counter.

Method Signature:
```
function GetConsecNum : Integer;
```

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [in] | U | Units (Primary, Secondary, Tertiary) |
| [out] | C | Count-by value |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | DT | DateTime variable name |
| [out] | Year | Year |
| [out] | Month | Month |
| [out] | Day | Day |

### GetGrads
Sets G to the configured grad value of scale S.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Scale number |
| [out] | G | Grads value |

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. |

### GetIqubeData
Returns data from a given iQube. The types that IQValType may be are: IQSys, IQPlat, IQRawLC, IQCorrLC, IQZeroLC, IQStatLC, IQScaleWt, and IQ2StatusLC. IQSys returns the system weight value. IQPlat returns the millivolt value for the indexed platform. IQRawLC returns the indexed raw load cell millivolt value. IQCorrLC returns the indexed corrected load cell millivolt value. IQZeroLC returns the indexed load cell deadload millivolt value. IQStatLC returns the indexed load cell status. IQ2ScaleWt returns the indexed scale weight value. IQSys and IQPlat are revised to also return the scale data. IQ2StatusLC returns the indexed load cell status. The old IQStatLC is not supported and will return SysInvalidRequest.

***When using with Firmware 4.xx/iQube2: The IQSys and IQPlat data types will return SysOk as long as the command is correctly formatted (i.e., scale exists). If you want to know whether the iQube2 is in an error condition, look at the value (not the syscode) of the IQ2StatusLC data type.***

Method Signature:
```
function GetIqubeData(port_no : integer; dataType : IQValType; index : integer;
data : real) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysOutOfRange` | The array index is less than or equal to 0. |
| `SysInvalidRequest` | The requested port is not configured as an iQube; the value cannot be returned due to the device configuration, i.e., trying to address load cell 17; certain requests while the diagnostic screen is open; or an invalid data type is requested. |
| `SysDeviceError` | The scale is reporting an internal error. |
| `SysOK` | The function completed successfully. |

### GetKey

Waits for a key press from the indicator front panel before continuing the program. The optional time-out is specified in 0.01-second intervals (1/100 seconds); if the wait time is set to zero, the procedure will wait indefinitely.

Method Signature:
```
function GetKey (timeout : Integer) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `timeout` | Time-out value |

Example:
```
this_key : Keys;
…
DisplayStatus ("Press [Enter] for Yes");

GetKey(0);
if this_key = EnterKey then
  DisplayStatus ("Yes");
else
  DisplayStatus ("No");

end if;
```

### GetSoftwareVersion

Returns the current software version.

Method Signature:
```
function GetSoftwareVersion : String;
```

### GetTime

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

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

Parameters:

| | | |
|---|---|---|
| [in] | `DT` | DateTime variable name |
| [out] | `Hour` | Hour |
| [out] | `Minute` | Minute |
| [out] | `Second` | Second |

### GetUID

Returns the current unit identifier.

Method Signature:
```
function GetUID : String;
```

### Hardware

Returns an array of HW_type. The elements of the array correspond to option card slots in the 920i. This API is useful for determining the presence of option cards that are required or that could activate different options in the user program.

Method Signature:

```
procedure Hardware(var hw : HW_array_type);
```

SysCode values returned: None

### KeyPress

Provides intrinsic functionality for a key. The following keys will have intrinsic function, in addition to the front panel keys already in the Keys built-in type: TimeDateKey, WeighInKey, WeighOutKey, ID_EntryKey, DisplayTareKey, TruckRegsKey, DisplayAccumKey, ScaleSelectKey, DisplayROCKey, SetpointKey, BatchStartKey, BatchStopKey, BatchPauseKey, BatchResetKey, DiagnosticsKey, ContactsKey, DoneKey, TestKey. The ContactsKey will actually function like the Dignostics softkey, while the DiagnosticsKey will go straight to the Diagnostics screen. The DoneKey will only return from the contacts screen. The TestKey will allow the user program to test for strict weigh mode by not doing anything at all. This API will only function in actual weigh mode.

Method Signature:

```
function KeyPress (K : Keys) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysInvalidMode` | The indicator is not actually in weigh mode. The TestKey will return SysInvalidMode for all sub-modes of weigh mode (ie, the contact screen) as well as any other mode (ie, time & date entry, or open prompt). |
| `SysInvalidKey` | Any Invalid key. Softkeys and Undefined Keys are considered invalid. |
| `SysInvalidRequest` | Processing the key returns invalid or error. |
| `SysOK` | The function completed successfully |

### 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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | `K` | Key name |

SysCode values returned:

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

### ProgramDelay

Pauses the user program for the specified time. Delay time is entered in 0.01-second intervals (1/100 seconds, 100 = 1 second).

Method Signature:

```
procedure ProgramDelay (D : Integer);
```

Parameters:

| | | |
|---|---|---|
| [in] | `D` | `Delay time` |

Example:

```
ProgramDelay(200); -- Pauses the program for 2 seconds.
```

### ResumeDisplay
Resumes a suspended display.

Method Signature:
```
procedure ResumeDisplay
```

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

Method Signature:
```
function SetConsecNum (V : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | V | Consecutive number |

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.

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

Parameters:

| | | |
|---|---|---|
| [out] | DT | DateTime variable name |
| [in] | Year | Year |
| [in] | Month | Month |
| [in] | Day | Day |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | K | Softkey number |
| [in] | S | Softkey text |

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. |

### SetSystemTime
Sets the realtime clock to the value specified in DT.

Method Signature:
```
function SetSystemTime (VAR DT : DateTime) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | DT | System DateTime |

SysCode values returned:

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

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

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

Parameters:

| | | |
|---|---|---|
| [out] | DT | DateTime variable name |
| [in] | Hour | Hour |
| [in] | Minute | Minute |
| [in] | Second | Second |

SysCode values returned:

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

### SetUID
Sets the unit identifier.


***Changes made to the UID using the SetUID function are lost when the indicator power is cycled. When power is restored, the UID is reset to the value at the last SAVE/EXIT from configuration mode.***

Method Signature:
```
function SetUID (newid : String) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | newid | Unit identifier |

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).

Method Signature:
```
function STick : Integer;
```

### SuspendDisplay
Suspends the display.

Method Signature:
```
procedure SuspendDisplay;
```

### SystemTime
Returns the current system date and time.

Method Signature:
```
function SystemTime : DateTime;
```

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

Method Signature:
```
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.

Method Signature:
```
function UnlockKey (K : Keys) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | K | Key name |

SysCode values returned:

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

#### UnlockKeypad

Enables operation of the entire front panel keypad.

Method Signature:
```
function UnlockKeypad : SysCode;
```

SysCode values returned:

| | |
|---|---|
| SysPermissionDenied | |
| SysOK | The function completed successfully. |

#### WaitForEntry()

Similar to GetEntry, WaitForEntry causes the user program to wait for operator input. Wait time is specified in 0.01-second intervals (1/100 seconds); if the wait time is set to zero, the procedure will wait indefinitely or until the **Enter** key is pressed.

**Note** ***The UserEntry handler must be disabled (see DisableHandler on page 42) before using this procedure.***

Method Signature:
```
procedure WaitForEntry (I : Integer);
```

Parameters:

| | | |
|---|---|---|
| [in] | I | Wait time value |

## 5.3 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.

Method Signature:
```
function Print (F : PrintFormat) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | F | Print format |

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*x* | 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);
```

### Send
Writes the integer or real number specified in <number> to the port specified by P.

Method Signature:
```
procedure Send (P : Integer; <number>);
```

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |

Example:
```
Send (Port1, 123.55); -- sends the value "123.55" to Port 1.
```

### SendChr
Writes the single character specified to the port specified by P.

Method Signature:
```
procedure SendChr (P : Integer; character Integer);
```

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |

Example:
```
SendChr (Port1, 65); -- sends upper-case "A" (ASCII 65) to Port 1.
```

### SendNull
Writes a null character (ASCII 00) to the port specified by P.

Method Signature:
```
procedure SendNull (P : Integer);
```

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |

Example:
```
Send (Port1); -- sends a null character (ASCII 00) to Port 1.
```

### SetPrintText
Sets the value of the user-specified format (1-99) to the text specified. The text can be any string of up to 16 characters; if a string of more than 16 characters is specified, nothing is printed.

Method Signature:
```
function SetPrintText (fmt_num : Integer ; text : String) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | fmt_num | User-specified format number |
| [in] | text | Print format text |

Example:
```
SetPrintText(1, "User Pgm. Text");
```

### StartStreaming
Starts data streaming for the port number specified by P. Streaming must be enabled for the port in the indicator configuration.

Method Signature:
```
function StartStreaming (P : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |

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.

Method Signature:
```
function StopStreaming (P : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |

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.

***This procedure cannot be used to send null characters. Use the SendChr or SendNull procedure to send null characters.***

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

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |
| [in] | arg_list | Print text |

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.

***This procedure cannot be used to send null characters. Use the SendChr or SendNull procedure to send null characters.***

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

Parameters:

| | | |
|---|---|---|
| [in] | P | Serial port number |
| [in] | arg_list | Print text |

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

## 5.4 Program Scale

### SubmitData

For *920i* indicators configured for program scale operation, passes data from a user program to the scale processor. Weight, mode, and tare values are provided by the user program; the displayed weight is the weight value minus tare. Gross/net mode is set by the gn parameter regardless of whether a tare value is passed. This allows display of a net value when the net is known but gross and tare values are not available.

Note that because the user program supplies all weight data, weight data acquisition APIs are not valid for program scales. When used with program scales, these APIs (including GetGross, GetNet, GetTare) will typically return a SysCode value of SysInvalidScale. Always check the returned SysCode value of scale-related APIs to ensure valid data.

Syntax:

```
function SubmitData (scale : Integer; weight : Real; gn : Mode; units : UnitType;
tare : Real) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| SysInvalidScale | The scale is not set up as a program scale. |
| SysOK | The function completed successfully. |

### SubmitDSPData

Submit data to a program scale. This function works much like SubmitData() but has fewer parameters. New to this function is the dp : Decimal_Type that allows the program to set the decimal point for the display. The call assumes Gross mode and primary units.

Syntax:

```
function SubmitDSPData( scale : integer; weight : real; units : string; dp :
Decimal_Type ) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| SysInvalidScale | The scale is not set up as a program scale. |
| SysOK | The function completed successfully. |

## 5.5 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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |

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).

Method Signature:

```
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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [out] | S | Batch status |

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 batch setpoint.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [out] | SP | Setpoint number |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Band value |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Captured weight value |

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. |

### GetSPCount
For DINCNT setpoints, sets Count to the value specified for setpoint SP.

Method Signature:
```
function GetSPCount (SP : Integer; VAR Count : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | Count | Count value |

SysCode values returned:

| | |
|---|---|
| SysInvalidSetpoint | The setpoint number specified by SP is less than 1 or greater than 100. |
| SysInvalidRequest | The specified setpoint is not a DINCNT setpoint. |
| SysOK | The function completed successfully. |

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | DT | Setpoint trip duration |

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);
```

### GetSPHyster

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Hysteresis value |

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);
```

### GetSPNSample

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | N | Sample value |

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);
```

### GetSPPreact

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Preact value |

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);
```

### GetSPPreCount

Sets Count to the preact learn interval value (PCOUNT parameter) of setpoint SP.

Method Signature:
```
function GetSPPreCount (SP : Integer; Count : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | Count | Preact learn interval value |

SysCode values returned:

| | |
|---|---|
| SysInvalidSetpoint | The setpoint specified by SP does not exist. |
| SysInvalidRequest | The setpoint specified by SP has no preact learn interval (PCOUNT) parameter. |
| SysOK | The function completed successfully. |

Example:
```
SP3PCount : Integer;
...
GetSPPreCount (3, SP3PCount);
WriteLn (Port1, "Current Preact Learn Value of SP3 is", SP3PCount);
```

### GetSPTime

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | DT | Current setpoint trip time |

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);
```

### GetSPValue

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Setpoint value |

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);
```

### GetSPVover

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

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Overrange value |

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 underrange value (VUNDER parameter) of the setpoint SP.

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [out] | V | Underrange value |

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

Initiates a latched pause of a running batch process.

Method Signature:
```
function PauseBatch : SysCode;
```

SysCode values returned:

| | |
|---|---|
| SysPermissionDenied | The BATCHNG configuration parameter is set to OFF. |
| SysBatchRunning | No batch routine is running. |
| SysOK | The function completed successfully. |

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

Method Signature:
```
function ResetBatch : SysCode;
```

SysCode values returned:

| | |
|---|---|
| SysPermissionDenied | The BATCHNG configuration parameter is set to OFF. |
| SysBatchRunning | No batch routine is running. |
| SysOK | The function completed successfully. |

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

Method Signature:
```
function SetBatchingMode (M : BatchingMode) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | M | Batching mode |

BatchingMode values sent:

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

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Band value |

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);
```

### SetSPCount
For DINCNT setpoints, sets the VALUE parameter of setpoint SP to the value specified by Count.

Method Signature:
```
function SetSPCount (SP : Integer; Count : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | Count | Count value |

SysCode values returned:

| | |
|---|---|
| SysInvalidSetpoint | The setpoint number specified by SP is less than 1 or greater than 100. |
| SysInvalidRequest | The specified setpoint is not a DINCNT setpoint. |
| SysOK | The function completed successfully. |

### SetSPDuration

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | DT | Setpoint trip duration |

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);
```

### SetSPHyster

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Hysteresis value |

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);
```

### SetSPNSample

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | N | Sample value |

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);
```

### SetSPPreact

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Preact value |

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);
```

### SetSPPreCount

Sets the preact learn interval value (PCOUNT parameter) of setpoint SP to the value specified by Count.

Method Signature:

```
function SetSPPreCount (SP : Integer; Count : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | Count | Preact learn interval value |

SysCode values returned:

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

Example:

```
SP3PCount : Integer;
…
SP3Pcount := 4;
SetSPPreCount (3, SP3PCount);
```

### SetSPTime

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | DT | Setpoint trip time |

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);
```

### SetSPValue

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Setpoint value |

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);
```

### SetSPVover

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Overrange value |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | SP | Setpoint number |
| [in] | V | Underrange |

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);
```

#### StartBatch
Starts or resumes a batch run.

Method Signature:
```
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.

Method Signature:
```
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. |

## 5.6 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).

#### 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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [in] | D | Bit number |
| [out] | D | Digital input status |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [in] | D | Bit number |
| [out] | D | Digital output status |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [in] | D | Bit number |
| [in] | D | Digital output status |

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);
```

## 5.7 Fieldbus Data

#### BusImage

BusImage is a data type to allow a user program to pass integer data to and from a fieldbus.

Method Signature:
```
type BusImage is array[32] of integer;
```

### BusImageReal
BusImageReal is a data type to allow a user program to pass real data to and from a fieldbus.

Method Signature:
```
type BusImageReal is array[32] of real;
```

### GetFBStatus
Returns the status word for the specified fieldbus. See the fieldbus *Installation and Programming* manual for a description of the status word format.

Method Signature:
```
function GetFBStatus (fieldbus_no : Integer;  scale_no : Integer; VAR status : Integer) :
SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `fieldbus_no` | Fieldbus number |
| [in] | `scale_no` | Scale number |
| [out] | `status` | Fieldbus status |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | |
| `SysOK` | The function completed successfully. |

### GetImage
For integer data, GetImage returns the content of the BusImage for the specified fieldbus.

Method Signature:
```
function GetImage (fieldbus_no : Integer; VAR data : BusImage) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `fieldbus_no` | Fieldbus number |
| [out] | `BusImage` | Bus image |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | |
| `SysOK` | The function completed successfully. |

### GetImageReal

For real data, GetImage returns the content of the BusImageReal for the specified fieldbus.

Method Signature:
```
function GetImageReal (fieldbus_no : Integer; VAR data : BusImageReal) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `fieldbus_no` | Fieldbus number |
| [out] | `BusImageReal` | Bus image |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | |
| `SysOK` | The function completed successfully. |

### SetImage
For integer data, SetImage sets the content of the BusImage for the specified fieldbus.

Method Signature:
```
function SetImage (fieldbus_no : Integer; data : BusImage) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `fieldbus_no` | Fieldbus number |
| [in] | `BusImage` | Bus image |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | |
| `SysOK` | The function completed successfully. |

#### SetImageReal
For real data, SetImageReal sets the content of the BusImageReal for the specified fieldbus.

Method Signature:
```
function SetImage (fieldbus_no : Integer; data : BusImageReal) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `fieldbus_no` | Fieldbus number |
| [in] | `BusImageReal` | Bus image |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | |
| `SysOK` | The function completed successfully. |

## 5.8 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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [in] | P | Analog output percentage value |

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. |

## 5.9 Pulse Input Operations

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [out] | C | Current pulse count |

SysCode values returned:

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

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

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

Parameters:

| | | |
|---|---|---|
| [in] | S | Slot number |
| [out] | C | Current pulse rate |

SysCode values returned:

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

## 5.10 Display Operations

#### ClosePrompt
Closes a prompt opened by the PromptUser function.

Method Signature:
```
procedure ClosePrompt;
```

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

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

Parameters:

| | | |
|---|---|---|
| [in] | msg | Display text |

#### GetEntry
Retrieves the user entry from a programmed prompt.

Method Signature:
```
function GetEntry : String;
```

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

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

Parameters:

| | | |
|---|---|---|
| [in] | msg | Prompt text |

SysCode values returned:

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

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

Method Signature:
```
function SelectScreen (N : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | N | Screen number |

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.

Method Signature:
```
procedure SetEntry (S : String);
```

## 5.11 Display Programming

### ClearGraph
Clears a graph by setting all elements of a DisplayImage array to zero.

Method Signature:
```
procedure ClearGraph (VAR graph_array : DisplayImage);
```

Parameters:

| | | |
|---|---|---|
| [out] | `graph_array` | Graph identifier |

### DrawGraphic
Displays or erases a graphic defined in the bitmap.iri file incorporated into the user program source (.src) file. See Section 6.6 on page 92 for more information about display programming.

Method Signature:
```
function DrawGraphic (gr_num : Integer; x_start : Integer; y_start : Integer;
bitmap : DisplayImage; color : Color_type; height : Integer; width : Integer) :
SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `gr_num` | Graphic number |
| [in] | `x_start` | X-axis starting pixel location |
| [in] | `y_start` | Y-axis starting pixel location |
| [in] | `bitmap` | Graphic bitmap |
| [in] | `color` | Color type |
| [in] | `height` | Graphic height |
| [in] | `width` | Graphic width |

SysCode values returned:

| | |
|---|---|
| `SysDeviceError` | The value specified for `gr_num` is greater than 100. |
| `SysOK` | The function completed successfully. |

Setting up a graph requires several functions that must be performed in the following order:

- GraphCreate assigns storage and defines the type of graph
- GraphInit sets the location of the graph on the display
- GraphScale sets the value bounds for the graph
- GraphPlot is used to actually plot the graph on the display

### GraphCreate
GraphCreate assigns storage and defines the graph display type for use by other graphing functions.

Method Signature:
```
function GraphCreate (graphic_no : Integer; bitmap : DisplayImage; color :
Color_type; kind : GraphType) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `graphic_no` | Graphic number |
| [in] | `bitmap` | Bitmap |
| [in] | `color` | Graphic color |
| [in] | `kind` | Graphic kind |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | The DisplayImage specified by `bitmap` does not exist. |
| `SysOK` | The function completed successfully. |

Example:

```
G_Graph1 : DisplayImage;
  result : Syscode;
begin
  result := GraphCreate(1, G_Graph1, Black, Bar);
  if result = SysOK then
    result :=GraphInit(71,30,60,110,240);
  end if;
end;
```

### GraphInit

GraphInit sets the location of the graph on the display. `x_start` and `y_start` values specify the distance, in pixels, from top left corner of the display at which the top left corner of the graph is shown. `height` and `width` specify the graph size, in pixels. (Full display size is 240 pixels high by 320 pixels wide.)

Method Signature:

```
function GraphInit (graphic_no : Integer; x_start : Integer; y_start : Integer;
height : Integer; width : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `graphic_no` | Graphic number |
| [in] | `x_start` | X-axis starting pixel location |
| [in] | `y_start` | Y-axis starting pixel location |
| [in] | `bitmap` | Graphic bitmap |
| [in] | `color` | Color type |
| [in] | `height` | Graphic height |
| [in] | `width` | Graphic width |

SysCode values returned:

| | |
|---|---|
| `SysInvalidRequest` | The DisplayImage specified by `bitmap` does not exist. |
| SysOutOfRange | Specified parameters exceed display height or width, or are too small to accommodate the graphic. |
| SysDeviceError | Internal error |
| `SysOK` | The function completed successfully. |

Example:

```
G_Graph1 : DisplayImage;
  result : Syscode;
begin
  result := GraphCreate(1, G_Graph1, Black, Bar);
  if result = SysOK then
    result :=GraphInit(71,30,60,110,240);
  end if;
end;
```

### GraphPlot

GraphPlot plots the graph previously set up using the GraphCreate, GraphInit, and GraphScale functions. The graph appears as a histogram: each GraphPlot call places a bar or line at the right edge of the graph, moving values from previous calls to the left. The width of the bar, in pixels, is specified by `width` parameter. The maximum width value is 8; larger values are reduced to 8. If the `y_value` is beyond the bounds set by GraphScale, the bar is plotted to the maximum or minimum value.

Method Signature:

```
function GraphPlot (graphic_no : Integer; y_value : Real; width : Integer; color :
Color_type) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `graphic_no` | Graphic number |
| [in] | `y_value` | Pixel height of histogram |
| [in] | `color` | Color type |
| [in] | `width` | Pixel width of moving bar |

SysCode values returned:

| | |
|---|---|
| SysInvalidRequest | Graph not initialized. |
| SysOK | The function completed successfully. |

Example:

```
  result : Syscode;
  weight : real;

begin
  GetGross(1,Primary,weight);
  result := GraphPlot(1, weight, 1, Black);
end;
```

### GraphScale

GraphScale sets the minimum and maximum x and y values for a graph. Currently, only the y values are used for the histogram displays; x values are reserved for future use, but must be present in the call.

Method Signature:

```
function GraphScale (graphic_no : Integer; x_min : Real; x_max : Real; y_min :
Real; y_max : Real) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | graphic_no | Graphic number |
| [in] | x_min | Minimum x-axis value |
| [in] | x_max | Maximum x-axis value) |
| [in] | y_min | Minimum y-axis value |
| [in] | y-max | Maximum y-axis value |

SysCode values returned:

| | |
|---|---|
| SysInvalidRequest | Graph not initialized. |
| SysOutOfRange | A minimum value (x_min or y_min) is greater than its specified max value. |
| SysOK | The function completed successfully. |

Example:

```
GraphScale(1, 10.0, 50000.0, 0.0, 10000.0);
```

### SetBargraphLevel

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

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | W | Bargraph widget number |
| [in] | Level | Bargraph widget level |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | W | Label widget number |
| [in] | S | Label widget text |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | W | Numeric widget number |
| [in] | V | Numeric widget value |

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.

Method Signature:
```
function SetSymbolState (W : Integer; S : Integer) : SysCode;
```

Parameters:

| | | |
|---|---|---|
| [in] | W | Symbol widget number |
| [in] | S | Symbol widget state |

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.

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

Parameters:

| | | |
|---|---|---|
| [in] | W | Widget number |
| [in] | V | Widget visibility |

SysCode values returned:

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

## 5.12 Event Handlers

#### BusCommandHandler
When enabled, this event handler is activated when new data arrives on a field bus option card. SetImage() must be called before BusCommandHandler() will be activated again. A new activation of the handler can occur when new data is present on the bus.

Method Signature:
```
BusCommandHandler()
```

#### xKeyReleased
This class of event handlers is activated when a key is released. The "x" is replaced with the name of the key. Key names are the same as for the xKeyPressed handlers. Note that the xKeyReleased handlers are subject to the same timing considerations as all other user handlers. The events are queued in the order they are detected. Any handler that involves lengthy operations may delay the start of other handlers.

Method Signature:
```
handler xKeyReleased;
```

## 5.13 Database Operations

#### <DB>.Add

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

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:

```
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.

Method Signature:

```
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. |

## 5.14 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.

#### 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.

Parameters:

| | | |
|---|---|---|
| [in] | T | Timer number |

Method Signature:

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

SysCode values returned:

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

#### ResumeTimer

Restarts a stopped timer T (1–32) from its stopped value.

Method Signature:

```
function ResumeTimer (T : Integer) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | T | Timer number |

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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | T | Timer number |
| [in] | V | Timer value |

SysCode values returned:

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

### SetTimerDigout

SetTimer Digout is used to provide precise control of state changes for timers using TimerDigoutOff or TimerDigoutOn modes. The state of the specified digital output (slot S, bit D)is changed when timer T (1–32) expires.

Method Signature:

```
function SetTimer (T : Integer ; S : Integer ; D: Integer) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | T | Timer number |
| [in] | S | Digital I/O slot number |
| [in] | D | Digital I/O bit number |

SysCode values returned:

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

Example:

```
SetTimer(1,100); -- Set value of Timer1 to 100 (1 second)
SetTimerMode(1,TimerDigoutOn); -- Set timer mode to turn on the digital output
SetTimerDigout(1,0,1); -- Set which digital output to control (slot 0, bit 1)
StartTimer(1); -- Start timer
```

### 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.

Method Signature:

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

Parameters:

| | | |
|---|---|---|
| [in] | T | Timer number |
| [in] | M | Timer mode |

TimerMode values sent:

| | |
|---|---|
| TimerOneShot | Timer mode is set to one-shot. |
| TimerContinuous | Timer mode is set to continuous. |
| TimerDigOutOff | One-shot timer sets a digital output off when the timer expires. |
| TimerDigOutOn | One-shot timer sets a digital output on when the timer expires. |

SysCode values returned:

| | |
|---|---|
| SysInvalidTimer | The timer specified by T is a not valid timer. |
| SysInvalidState | The timer specified by M is a not valid timer mode. |
| SysInvalidRequest | The slot or bit number configured is not a valid digital output . |
| 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.

Method Signature:
```
function StartTimer (T : Integer) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `T` | Timer number |

SysCode values returned:

| | |
|---|---|
| `SysInvalidTimer` | The timer specified by `T` a not valid timer. |
| `SysOK` | The function completed successfully. |

#### StopTimer
Stops timer `T` (1–32).

Method Signature:
```
function StopTimer (T : Integer) : Syscode;
```

Parameters:

| | | |
|---|---|---|
| [in] | `T` | Timer number |

SysCode values returned:

| | |
|---|---|
| `SysInvalidTimer` | The timer specified by `T` a not valid timer. |
| `SysOK` | The function completed successfully. |

## 5.15 Mathematical Operations

#### Abs
Returns the absolute value of `x`.

Method Signature:
```
function Abs (x : Real) : Real;
```

#### ATan
Returns a value between $-\pi/2$ and $\pi/2$, representing the arctangent of `x` in radians.

Method Signature:
```
function Atan (x : Real) : Real;
```

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

Method Signature:
```
function Ceil (x : Real) : Integer;
```

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

Method Signature:
```
function Cos (x : Real) : Real;
```

#### Exp
Returns the value of $e^x$.

Method Signature:
```
function Exp (x : Real) : Real;
```

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

Method Signature:
```
function Log (x : Real) : Real;
```

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

Method Signature:
```
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.

Method Signature:
```
function Sign (x : Real) : Integer;
```

#### Sin
Returns the sine of X. X must be specified in radians.

Method Signature:
```
function Sin (x : Real) : Real;
```

#### Sqrt
Returns the square root of X.

Method Signature:
```
function Sqrt (x : Real) : Real;
```

#### Tan
Returns the tangent of X. X must be specified in radians.

Method Signature:
```
function Tan (x : Real) : Real;
```

## 5.16 Bit-wise Operations

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

Method Signature:
```
function BitAnd (X : Integer; Y : Integer) : Integer;
```

#### BitNot

Returns the bit-wise NOT result of X.

Method Signature:
```
function BitNOT (X : Integer) : Integer;
```

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

Method Signature:
```
function BitOr (X : Integer; Y : Integer) : Integer;
```

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

Method Signature:
```
function BitXor (X : Integer; Y : Integer) : Integer;
```

## 5.17 Built-in Types

#### BatchingMode
type BatchingMode is (Off, Auto, Manual);

#### BatchStatus
type BatchStatus is (BatchComplete, BatchStopped, BatchRunning, BatchPaused);

#### BusImage
type BusImage is array[32] of integer;

#### Color_type
type Color_type is (White, Black);

#### DataArray
type DataArray is array[300] of real;

#### Decimal_type
type Decimal_type is ( DP_8_888888, DP_88_88888, DP_888_8888, DP_8888_888, DP_88888_88, DP_888888_8, DP_8888888, DP_8888880, DP_8888800, DP_DEFAULT );

#### DisplayImage
type DisplayImage is array[2402] of integer; Type DisplayImage is for user graphics and will hold the largest displayable user graphic.

#### DTComponent
type DTComponent is (DateTimeYear, DateTimeMonth, DateTimeDay, DateTimeHour, DateTimeMinute, DateTimeSecond);

#### ExtFloatArray
type ExtFloatArray is array[5] of integer;

#### GraphType
type GraphType is ( Line, Bar, XY );

#### HW_array_type
type HW_array_type is array[14] of HW_type; Used with the Hardware() API, each element of the array represents a 920i expansion slot.

#### HW_type
type HW_type is ( NoCard, DualSerial, DualAtoD, SingleAtoD, AnalogOut, DigitalIO, Pulse, Memory, reservedcard, DeviceNet, Profibus, reserved2card, ABRIO, reserved3card, DSP2000, AnalogInput, Ethernet ); Each of the enumerations represent a kind of option card for the 920i.

#### Keys
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);

#### Mode
type Mode is (GrossMode, NetMode);

#### OnOffType
type OnOffType is (VOff, VOn);

#### PrintFormat
type PrintFormat is (GrossFmt, NetFmt, AuxFmt, TrWInFmt, TrRegFmt, TrWOutFmt, SPFmt, AccumFmt, AlertFmt);

#### SysCode
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, SysInvalidChecksum, SysDatabaseAccessTimeout);

#### TareType
type TareType is (NoTare, PushbuttonTare, KeyedTare);

#### TimerMode
type TimerMode is (TimerOneShot, TimerContinuous, TimerDigoutON, TimerDigoutOFF);

#### Units
type Units is (Primary, Secondary, Tertiary);

#### UnitType
type UnitType is (kilogram, gram, ounce, short_ton, metric_ton, grain, troy_ounce, troy_pound, long_ton, custom, none, pound);

#### WgtMsg
type WgtMsg is array[12] of integer;

## 5.18 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.

Method Signature:
```
function Asc (S : String) : Integer;
```

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

Method Signature:
```
function Chr$ (I : Integer) : String;
```

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

Method Signature:
```
function Hex$ (I : Integer) : String;
```

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

Method Signature:
```
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.

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

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

Method Signature:
```
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.

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

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

Method Signature:
```
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.

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

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

Method Signature:
```
function Space$ (N : Integer) : String;
```

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

Method Signature:
```
function UCase$ (S : String) : String;
```

## 5.19 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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
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.

Method Signature:
```
function StringToReal (S : String) : Real;
```

## 5.20 High Precision

#### DecodeExtFloat
A five-byte IEEE-1594 extended floating point number, expressed as an array or bytes, is converted to a standard 4-byte floating point real. NaN and infinity are processed. If a number is too small to convert to 4-byte precision, zero is returned. If a number is too large to convert to 4-byte precision, infinity is returned.

Method Signature:
```
function DecodeExtFloat( weight : ExtFloatArray ) : real;
```

#### EncodeExtFloat
Converts a 4-byte floating point real to a 5-byte IEEE-1394 extended floating point number in the form of an array of five bytes.

Method Signature:
```
function EncodeExtFloat( weight : real ) : ExtFloatArray;
```

### DecodeMessage

An entire measured value response message is partially decoded. Use with DecodeWeight() to decode all parameters. This routine takes an entire measured value response message (from byte count to checksum) and decodes the quarter-D bit, the weighing range, stability, and the verified bit. Enumerations are returned as integers specified in the Sartorius xBPI protocol documentation.

Method Signature:
```
function DecodeMessage( msg : WgtMsg; var qd : integer; var range : integer; var
stability : integer; var verified : integer ) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysInvalidChecksum` | The message checksum is incorrect. |
| `SysOK` | The function completed successfully. |

### DecodeWeight

An entire measured value response message is partially decoded. Use with DecodeMessage() to decode all parameters. This routine takes an entire measured value response message (from byte count to checksum) and decodes the weight, decimal point, units, and status. Enumerations are returned as integers specified in the Sartorius xBPI protocol documentation.

Method Signature:
```
function DecodeWeight( msg : WgtMsg; var weight : real; var dp : integer; var units
: integer; var status : integer ) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysInvalidChecksum` | The message checksum is incorrect. |
| `SysOK` | The function completed successfully. |

### InitHiPrec

Establishes communications with an RS-485 Sartorius platform. Sends a BREAK signal to the platform, then sets communications parameters for xBPI protocol, RS-485, 9600 baud, 8-bit odd parity, two stop bits. 920i indicator must have port set up accordingly. The platform parameters set by InitHiPrec() are: * Set baud rate to 9600 * Delete tare and application tares * The following are parameter table settings * Allow changes in the Parameter Table (setting 40) * Standard Weighing Mode (setting 2) * Stability Range (setting 3) * Stability Symbol Delay (setting 4) * Auto Zero (setting 6) * Zero Range (setting 11) * Power-On Zero Range (setting 12) * Power-On Tare or Zeroing (setting 13) * Normal Output of Measured Values (setting 14) * Calibration Prompt Off (setting 15) * Only one Weighing Range (setting 25) * Weight Units = Kilograms (setting 7) * Basic Accuracy (setting 8) * Communication Type = xBPI (setting 35) * Data Output at Defined Intervals = Auto (setting 38) * Allow Tare and Zero without standstill (setting 5) Setting numbers are the Sartorius parameter table setting numbers.

Method Signature:
```
function InitHiPrec( port_no : integer ) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysRequestFailed` | The function did not complete. |
| `SysOK` | The function completed successfully. |

### SubmitMessage

An entire Sartorius weight response message (from byte count to checksum) is decoded and submitted to a scale for display. The scale must be setup as a Program Scale.

Method Signature:
```
function SubmitMessage( scale : integer; msg : WgtMsg ) : SysCode;
```

SysCode values returned:

| | |
|---|---|
| `SysInvalidChecksum` | The message checksum is incorrect. |
| `SysOK` | The function completed successfully. |
| `SysInvalidScale` | The scale is not a Program Scale. |

## 5.21 USB

User program access to the USB file system requires new APIs for the user program to manipulate and use these files. A user program may have only one file open at a time. Once opened, any further file accesses will be to that file.

### USBFileOpen( filename : string; mode : FileAccessMode ) : Syscode

This API is used to read a file from the flash drive. Opening a file as Read positions the internal pointer at the start of the file. Opening a file as Create or Append positions the internal pointer at the end of the file. Any attempt to read a file opened as Create or Append will return SysEndOfFile.

Parameters:

`Filename` - The 920i will look in a folder named whatever the 920i's UID is set for (defaulted to 1) for the filename sent as the parameter. Use the entire path (without the drive). For example, if your file is stored on C:/ Examples/USB/Testing.txt the parameter would be: Examples/USB/Testing.txt

`FileAccessMode` - A new enumeration (see Section 4.0) with the choices of FileCreate, FileAppend, or FileRead.

SysCode values returned:

```
SysOk                                   SysFileExists
SysNoFileSystemFound                    SysInvalidFileFormat
SysPortBusy                             SysBadFilename (over 8 characters)
SysFileNotFound                         SysEndOfFile
SysDirectoryNotFound
```

Example:

`USBFileOpen(Testing.txt, FileCreate);` --Creates a new empty file called Testing.txt.

`USBFileOpen(test,FileAppend);` --Adds to a currently stored file called Testing.txt

`USBFileOpen(test,FileRead);` --Reads from a currently stored file

### USBFileClose()

This API is used to close a currently opened file (see USBFileOpen). A file must be closed before device removal or the file contents may be corrupted.

No parameters.

SysCode values returned:

```
SysOk                                   SysMediaChanged
SysNoFileSystemFound                    SysNoFileOpen
```

### USBFileDelete( filename : string )

This API deletes a file saved to the USB drive. To overwrite an existing file, the user program should first delete the file, then reopen it with Create access.

Parameters:

`Filename` - The 920i will look in a folder named whatever the 920i's UID is set for (defaulted to 1) for the filename sent as the parameter.

SysCode values returned:

```
SysOk                                   SysFileNotFound
SysNoFileSystemFound                    SysDirectoryNotFound
SysPortBusy                             SysBadfilename
```

Example:

```
USBFileDelete(Testing.txt);
```

#### USBFileExists( filename : string )
This API checks to see if a file exists on the USB drive.

Parameters:

`Filename` - The 920i will look in a folder named whatever the 920i's UID is set for (defaulted to 1) for the filename sent as the parameter.

SysCode values returned:

```
SysOk                          SysInvalidMode
SysNoFileSystemFound           SysBadfilename
SysPortBusy
```

Example:

```
USBFileExists(Testing.txt);
```

#### ReadLn( var data : string )
This API will read a string from whatever file is currently open. The string will be placed in a string-type-variable that must be defined.

Parameters:

`Data:` This is the string type variable that they data will be placed in to display or print or otherwise be used by the program. It reads one line at a time and the entire line is in this string.

SysCode values returned:

```
SysOk                          SysNoFileSystemFound
SysNoFileOpen                  SysEndOfFile
SysMediaChanged
```

Example:

`Result := ReadLn(sTempString);` --Reads a line of data from whatever file is open

while `Result <> SysEndOfFile` --Loops, looking at the return code until the end

```
loop
   Result := ReadLn(sTempString);
   WriteLn(3, sTempString);   --Prints each line read out Port 3
end loop;
```

#### WriteLn( port : integer; data : string )

#### Write( port : integer; data : string )
These APIs both writ out a port (and are not new to USB but can be used by the USB). If writing to the USB drive it will append the string to the end of the currently open file. The only difference between the two is the WriteLn sends a carriage return/line feed at the end, and Write does not.

Parameters:

`Port` - Whichever port on the 920i the data will be sent out of. Port 2 is used for USB.

Example - see ReadLn.

#### GetUSBStatus() : Syscode
This API returns the most recent status report for the USB port. This is useful for validating a Write or WriteLn.

Example:

```
Result := GetUSBStatus;
```

#### GetUSBAssignment() : deviceType
Returns the DeviceType currently in use.

Example:

`dDevice := GetUSBAssignment;` -- verify the assignment

```
if dDevice = USBFileSystem then
   WriteLn(3,"USBFlashDrive");
elsif dDevice = USBHostPC then
   WriteLn(OutPort,"USBHostPC");
elsif dDevice = USBPrinter2 then
   WriteLn(OutPort,"USBPrinter2");
elsif dDevice = USBPrinter1 then
   WriteLn(OutPort,"USBPrinter1");
elsif dDevice = USBKeyboard then
   WriteLn(OutPort,"USBKeyboard");
else
   WriteLn(OutPort,"Device Unknown");
end if;
```

#### SetUSBAssignment( device : deviceType )

Selects a secondary device for current use, capturing the current device as primary.

Parameters: `device` (see Section 4.0).

SysCode values returned:

| | |
|---|---|
| `SysOk` | `SysPortBusy` |
| `SysDeviceNotFound` | |

Example:

```
SetUSBAssignment(USBHostPC);
```

#### ReleaseUSBAssignment()

Returns the current USB device to the captured primary device.

SysCode values returned:

| | |
|---|---|
| `SysOk` | `SysPortBusy` |
| `SysDeviceNotFound` | |

Example:

```
ReleaseUSBAssignment;
```

#### IsUSBDevicePresent( device : deviceType )

Checks to see if the device passed is there or not.

Parameters: `device` (see Section 4.0).

SysCode values returned:

| | |
|---|---|
| `SysOk` | `SysDeviceNotFound` |

Example:

```
Result := IsUSBDevicePresent(USBFileSystem);
if Result <> SysOk then
   WriteLn(OutPort,"Flash Drive Not Found");
else
   WriteLn(OutPort,"SysOK");
end if;
```

#### SetFileTermin( termin : LineTermination )
This determines what is appended at the end of each line.

`Termin` - See Section 4.0 for LineTermination type options.

Example:

```
SetFileTermin(FileCRLF);
```

#### DBLoad(database name)
Opens a file in Read mode using the name of the database and the Unit ID and calls the core to process it as a database file. The file is closed when done.

SysCode values returned:

```
   SysOk                          SysFileNotFound
   SysNoSuchDatabase              SysDirectoryNotFound
   SysNoFileSystemFound           SysInvalidFileFormat
   SysFileAlreadyOpen             SysPortBusy
```

Example:

```
if DBLoad("Product") = Sysok then
   DisplayStatus("Product Database Loaded into 920i")
end if;
```

#### DBSave(database name)
Opens a file in Create mode using the name of the database and the Unit ID and calls the core to process it as a database file. File is closed when done. For example if the Unit ID in the 920i was 5, it would store a file to E:/5/Product.txt. (If your computer recognized the thumb drive as drive E).

SysCode values returned:

```
   SysOk                          SysFileNotFound
   SysNoSuchDatabase              SysDirectoryNotFound
   SysNoFileSystemFound           SysFileExists
   SysFileAlreadyOpen             SysPortBusy
```

Example:

```
if DBSave("Product") = Sysok then
   DisplayStatus("Product Database Saved to thumb drive")
end if;
```

#### SysCodeToString( code : SysCode )
Returns the name of the SysCode as a string (so it can be printed or displayed).

Example:

```
Result := SetFileTermin(FileCRLF);
if Result <> SysOk then
  WriteLn(3,(SysCodeToString(Result)));--Makes the syscode able to be printed
else
  WriteLn(3,"SysOK");
end if;
```

# 6.0 Appendix

## 6.1 Event Handlers

| Handler | Description |
|---|---|
| AlertHandler | Runs when an error is generated from an attached *iQUBE*. Use the EventString function to retrieve the error message displayed by the *920i*. |
| BusCommandHandler | Runs when data is received on the fieldbus. |
| ClearKeyPressed | Runs when the CLR key on the numeric keypad is pressed |
| ClearKeyReleased | Runs when the CLR key on the numeric keypad is released |
| 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 communications 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 43). |
| DiginS*x*By*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*By*y*Deactivate | Runs when the digital input assigned to slot *x*, bit *y* is deactivated. 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 |
| DotKeyReleased | Runs when the decimal point key on the numeric keypad is released |
| EnterKeyPressed | Runs when the ENTER key on the front panel is pressed |
| EnterKeyReleased | Runs when the ENTER key on the front panel is released |
| GrossNetKeyPressed | Runs when the GROSS/NET key is pressed |
| GrossNetKeyReleased | Runs when the GROSS/NET key is released |
| KeyPressed | Runs when any front panel key is pressed. Use the EventKey function within this handler to determine which key caused the event. |
| KeyReleased | Runs when any front panel key is released. 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. |
| MajorKeyReleased | Runs when any of the five preceding major keys is released. Use the EventKey function within this handler to determine which key caused the event. |
| NavDownKeyPressed | Runs when the DOWN navigation key is pressed |
| NavDownKeyReleased | Runs when the DOWN navigation key is released |
| 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. |
| NavKeyReleased | Runs when any of the navigation cluster keys (including ENTER) is released. Use the EventKey function within this handler to determine which key caused the event. |
| NavLeftKeyPressed | Runs when the LEFT navigation key is pressed |
| NavLeftKeyReleased | Runs when the LEFT navigation key is released |
| NavRightKeyPressed | Runs when the RIGHT navigation key is pressed |
| NavRightKeyReleased | Runs when the RIGHT navigation key is released |
| NavUpKeyPressed | Runs when the UP navigation key is pressed |
| NavUpKeyReleased | Runs when the UP navigation key is released |
| 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. |

*Table 6-1. 920i Event Handlers*

| Handler | Description |
|---|---|
| NumericKeyReleased | Runs when any key on the numeric keypad (including CLR or decimal point) is released. 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 |
| N*x*KeyReleased | Runs when a numeric key is released, 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. |
| PrintFmt*x* | Runs when a print format *x* (1–10) that includes the event raised (<EV>) token is printed. |
| PrintKeyPressed | Runs when the PRINT key is pressed |
| PrintKeyReleased | Runs when the PRINT key is released |
| 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. |
| SoftKeyReleased | Runs when any softkey is released. 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 |
| Soft*x*KeyReleased | Runs when softkey *x* is released, 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 |
| TareKeyReleased | Runs when the TARE key is released |
| 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 |
| UnitsKeyReleased | Runs when the UNITS key is released |
| User*x*KeyPressed | Runs when a user-defined softkey is pressed, where *x* is the user-defined key number, 1–10 |
| User*x*KeyReleased | Runs when a user-defined softkey is released, 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 |
| ZeroKeyReleased | Runs when the ZERO key is released |

*Table 6-1. 920i Event Handlers (Continued)*

## 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 |

*Table 6-2. iRite Compiler Error Messages*

| Error Messages | Cause (Statement Type) |
|---|---|
| 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 statement |
| Step value must be constant | For statement |
| Subprogram invocation is missing parameters | Procedure/function call |
| 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*

## 6.3 iRev Database Operations

You can use *iRev* and the *920i Interchange* ® database utility software (PN 72809) to edit, save, and restore databases for the *920i*. This section describes procedures for maintaining *920i* databases using *iRev*, including:

- Upload: Copies the database from the *920i* to *iRev*
- Download: Copies the database from *iRev* to the *920i*
- Import: Copies the contents of a database file stored on the PC into *iRev*
- Export: Copies the contents of a database file opened in *iRev* to a file on the PC
- Clear All: Clears the contents of a database on the *920i*.


***An existing 920i database must be cleared before downloading a database of the same name.***

- Editing: Databases can be displayed and edited using the *iRev* Data Editor

### 6.3.1 Uploading

To upload a database from the indicator (for viewing, editing, or backup), do the following:

1. Make a serial connection between the PC and the *920i*
2. Start *iRev*
3. Connect to the indicator by clicking on the **Connect** button on the right side of the top toolbar
4. Click the *Database* bar on the left side of the *iRev* window
5. Click the *Data Editor* icon.
6. Select the database you want to upload, then click the **Upload** button on top right of the toolbar.
7. A status message box will confirm that *iRev* is *Uploading Data*. When the upload is done, the message will change to *Upload Complete. Please export your data to a delimited file for backup*. Click **OK**.

You can now view, edit, or export the contents of the *920i* database. Note that changing the database in *iRev* alone does not change the database stored in the *920i*; you must then clear the existing *920i* database and replace it by downloading the edited database (see Section 6.3.5 on page 89).

### 6.3.2 Exporting

For display, printing, or backup, you can save a database opened in *iRev* to a text file by using the Export function.

1. With an open database uploaded to or created in *iRev*, click **Export** on the top toolbar.
2. A dialog box is shown to select which separator (delimiter) should be used to separate the database fields. For example, if you pick tab-delimiting for a customer database, it might look like the following:

   ```
   ElliotRobert1234555-8686
   ```

   If you select semi-colon delimiting instead, the same entry will appear as shown below:

   ```
   Elliot;Robert;1234;555-8686.
   ```

3. After you select the delimiter, click **Begin**. You are prompted to choose where to store the text file. Save it in the same folder as your other program files.
4. When complete, a message box confirms *Export Successful*. You can now use the exported file for viewing or printing the database, or for later import to *iRev* for download to the *920i*.

### 6.3.3 Importing

Import works the same way as export but brings a previously exported text file into *iRev*. The imported database can then be downloaded to the *920i*.

1. Start the *iRev* Data Editor and select the table you into which you want to import data.
2. Click **Import** on the top toolbar.
3. A dialog box is shown to select the file to import. Double click on the file you want to import.
4. The *Data Import Wizard* box is shown that displays the first couple of rows of data in your file. **Notice that the field names are shown as the first row**. This is *not* something you want to import into your database

since the field names are not part of the data. Click the up arrow next to *Start import at row:* prompt to start at row 2 (the real data).

5. Click *Next* and select the separator character you used when the file was exported (the default is tab-delimited).
6. Click *Next* again, then click *Finish* to import the file. All of your data should now be displayed in *iRev*. If you wish to downloaded the imported database to the *920i*, follow the procedure described in Section 6.3.5.

### 6.3.4 Clearing

The **Clear All** button on the top of the toolbar in the *iRev* Data Editor clears both the *iRev* screen and the entire *920i* database. You must clear an existing *920i* database before downloading edited data, but this function must be used with care to avoid losing data.

To clear a database:

1. Upload the database from the *920i* (see Section 6.3.1).
2. Edit the database and fields, if necessary.
3. Use the Export function described in Section 6.3.2 to save a copy of the database.
4. Highlight all of the fields at once and copy them using either Ctrl-C or by choosing *Edit-Copy* from the toolbar.
5. Click the **Clear All** button to clear both the *920i* database and the *iRev* fields.
6. Upload the blank database from the *920i* to ensure data integrity. The lock symbol on the *iRev* screen will open, allowing a new database to be downloaded.
7. To replace the cleared database with edited data, move the cursor to the upper left-hand box and paste the copied data back into the *iRev* database. (Press Ctrl-V or choose *Edit-Paste* from the toolbar.)
8. Click the **Download** button to send fresh, edited data back down to the indicator (see below).

### 6.3.5 Downloading

**IMPORTANT:** When you download data to the *920i*, it does **not** overwrite data that is there. Downloaded data is added to the database regardless of whether it is the same data. If you edit uploaded data in *iRev* and want to replace the indicator database, you must first Clear All, upload the cleared (blank) database, and then download the edited data. (See Section 6.3.4 above.)

1. Create or edit the data in the rows and columns you want entered in the database.
2. With the indicator connected, click the **Download** button at the top on the toolbar.
3. A status box shows the download progress (*Downloading Row [number] of [total rows]*). When complete, a *Download completed successfully* message is shown. The database is now stored in the *920i*.

## 6.4 Fieldbus User Program Interface

The fieldbus data APIs (see “Fieldbus Data” on page 63), two type definitions (BusImage, BusImageReal), and the EventPort function are used to manage fieldbus data.

The function of BusCommandHandler is similar to other user-written event handlers. When present and enabled with the EnableHandler(BusCommandHandler) call, the BusCommandHandler is activated every time a message is received on a fieldbus. Keeping the BusCommandHandler execution short is important in order to not miss data transfers on the fieldbus.

The normal operation of BusCommandHandler is expected to include the following system calls in the following order:

- EventPort
- GetImage, or GetImageReal
- SetImage, or SetImageReal

with intervening code to perform the required user functions. The SetImage or SetImageReal call should be as close to the end of the BusCommandHandler as possible.

The BusImage type is the data type passed in GetImage and SetImage (or, for real data, GetImageReal and SetImageReal).

```
GetImage(fieldbus_no : integer; var data : BusImage) : SysCode
```

This call returns an array of data as received from the fieldbus. As only the data elements received on the fieldbus are changed in a GetImage call, the array should be initialized prior to the GetImage call. The `fieldbus_no` is the number returned by an EventPort call from within the BusCommandHandler.

```
SetImage(fieldbus_no : integer; var data : BusImage) : SysCode
```

This call writes data to the fieldbus chip for access on the next cycle of the PLC. All data elements of the data array should be properly set before calling SetImage. The `fieldbus_no` is the number returned by an EventPort call from within the BusCommandHandler.

#### Example BusCommandHandler Code

```
------------------------------------------------------------
-- Handler Name : BusCommandHandler
-- Created By : Rice Lake Weighing Systems
-- Last Modified on : 1/16/2003
--
-- Purpose : Example handler skeleton.
--
-- Side Effects :
------------------------------------------------------------
handler BusCommandHandler;
--Declaration Section
busPort : integer;
data : BusImage;
i : integer;
result : SysCode;

begin
    -- Clear out the data array.
    for i := 1 to 32 loop
        data[i] := 0;
    end loop;

    -- Find out which port (which bus card) started this event.
    busPort := EventPort;

    -- Then read the received data.
    result := GetImage(busPort, data);

-- Test result as desired

-- Data interpretation and manipulation goes here.

    -- Finally, put the changed data back.
    result := SetImage(busPort, data);

-- Test result as desired

end;
```

## 6.5 Program to Retrieve 920i Hardware Configuration

The HARDWARE serial command (see Section 10 of the *920i Installation Manual*, PN 67887) returns a list of coded identifiers to describe which option cards are installed in a *920i* system. The following program provides a similar function by deciphering the coded values returned by the HARDWARE command and printing a list of installed option cards.

The largest *920i* system configuration (CPU board plus two six-card expansion boards) can support up to 14 option cards; the following program builds a 1 x 14 array by searching each slot for an installed option card then printing a list of slots and installed cards.

```
program Hardware;

    my_array : HW_array_type;


handler User1KeyPressed;

  i : integer;
  next_slot : HW_type;
  begin
    Hardware(my_array);
    for i := 1 to 14
    loop
      if my_array[i] = NoCard then
       WriteLn(2,"Slot ",i," No Card");
      elsif my_array[i] = DualAtoD then
       WriteLn(2,"Slot ",i," DualAtoD");
      elsif my_array[i] = SingleAtoD then
       WriteLn(2,"Slot ",i," SinglAtoD");
      elsif my_array[i] = DualSerial then
       WriteLn(2,"Slot ",i," DualSerial");
      elsif my_array[i] = AnalogOut then
       WriteLn(2,"Slot ",i," AnalogOut");
      elsif my_array[i] = DigitalIO then
       WriteLn(2,"Slot ",i," DigitalIO");
      elsif my_array[i] = Pulse then
       WriteLn(2,"Slot ",i," Pulse");
      elsif my_array[i] = Memory then
       WriteLn(2,"Slot ",i," Memory");
      elsif my_array[i] = DeviceNet then
       WriteLn(2,"Slot ",i," DeviceNet");
      elsif my_array[i] = Profibus then
       WriteLn(2,"Slot ",i," Profibus");
      elsif my_array[i] = Ethernet then
       WriteLn(2,"Slot ",i," Ethernet");
      elsif my_array[i] = ABRIO then
       WriteLn(2,"Slot ",i," ABRIO");
      elsif my_array[i] = BCD then
       WriteLn(2,"Slot ",i," BCD");
      elsif my_array[i] = DSP2000 then
       WriteLn(2,"Slot ",i," DSP2000");
      elsif my_array[i] = AnalogInput then
       WriteLn(2,"Slot ",i," AnalogInput");
      elsif my_array[i] = ControlNet then
       WriteLn(2,"Slot ",i," ControlNet");
      elsif my_array[i] = DualAnalogOut then
       WriteLn(2,"Slot ",i," DualAnalogOut");
      end if;

    end loop;
    WriteLn(2,"");
  end;

end Hardware;
```

## 6.6 920i User Graphics

*iRite* user programs can be used to display graphics. The entire *920i* display is writeable; graphics can be of any size, up to the full size of the *920i* display, and up to 100 graphic images can be displayed. The actual number of graphics that can be loaded depends on the size of the graphics and of the user program, both of which reside in the user program space.

Graphics used in *iRite* programs can be from any source but must be saved as monochrome bitmap (.bmp) files with write access (file cannot be read-only). To enable the file for use in an *iRite* program, it is converted to a user program #include (.iri) file using the bmp2iri.exe program (see Figure 6-1).

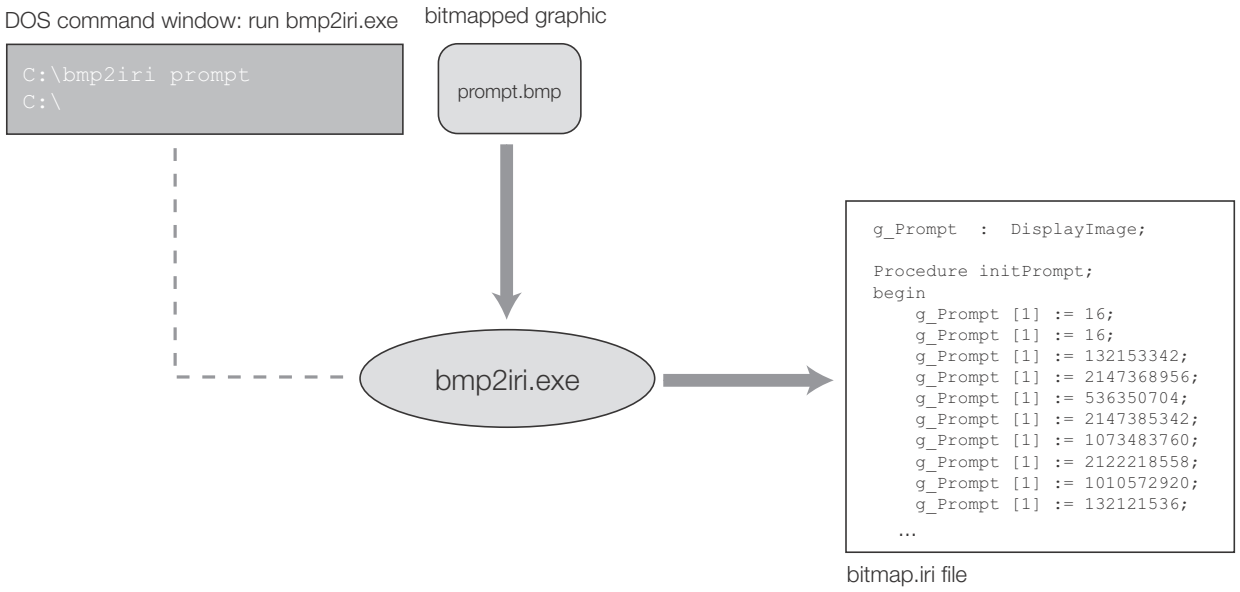


*Figure 6-1. Example of Converting Bitmapped Graphic (prompt.bmp) to an .iri File*

Figure 6-1 shows the conversion process for a graphic file, prompt.bmp, to a user program #include, bitmap.iri. The conversion is done by running the bmp2iri.exe program in a DOS command window: note that the bmp2iri program assumes the .bmp extension for the input graphic file (prompt.bmp). If additional files are converted using bmp2iri.exe, the output of the program is appended to the bitmap.iri file.

To display the graphic, the bitmap.iri file must be incorporated into the user program by doing the following:

- In the *iRite* source (.src) file, immediately following the program declaration, add: `#include bitmap.iri`
- In the startup handler, call the array initialization routine for each graphic.
- To display or erase a graphic, or to clear all graphics, call the DrawGraphic API with the appropriate parameters (see page 67).

# API Index

## P



## R



## S

## T



## U



## W

## Z

RICE LAKE®
WEIGHING SYSTEMS

230 W. Coleman St. • Rice Lake, WI 54868 • USA
U.S. 800-472-6703 • Canada/Mexico 800-321-6703
International 715-234-9171

**www.ricelake.com**
**mobile: m.ricelake.com**



PN 67888 04/12