VCL Programmer's Guide

MANUAL

VEHICLE CONTROL LANGUAGE

Programmer's Guide

VCL Programmer's Guide, p/n 37313
Rev. A: September 2005

CURTIS INSTRUMENTS, INC.
200 Kisco Avenue
Mt. Kisco, New York 10509 USA
Tel. 914.666.2971
Fax 914.666.2188
www.curtisinstruments.com

CONTENTS

OVERVIEW

This manual describes how to use VCL, Curtis's **V**ehicle **C**ontrol **L**anguage. VCL is a robust real-time programming language targeted at real-time vehicle control applications. VCL:

- allows fast testing of alternative design strategies
- builds real-time control systems that are simple, safe, and efficient
- is easy to learn and easy to maintain
- allows quick and easy customization for specific customer requests.

After reading this manual, you will be able to design programs that make efficient use of your system's resources and that are straightforward to read and to maintain.

You will find this manual useful if you are directly responsible for writing programs in VCL, or if you are simply evaluating the suitability of VCL for your project, or if you are responsible for supervising a project that uses VCL. You should have some programming experience. To use VCL effectively, you should also have some appreciation of the problems involved in making a program respond in real-time.

Organization of the Manual

This manual begins with a general introduction to VCL (Section 1) followed by a description of those aspects of VCL that are common to most programming languages (Sections 2 and 3). Because VCL is a procedural language—like Basic and C—this information should seem familiar. Next comes a description of VCL's unique automatic commands, which allow you to guarantee the sequencing of signal processing (Section 4). We then describe how to create and use programmable parameters, another unique aspect of VCL (Section 5). There are also discussions on how best to structure VCL programs and how to efficiently find any errors in them (Sections 6 and 7).

TEXT EDITOR

The creation of source code requires the use of a text editor. Although any text editor will create code suitable for compilation, we strongly suggest that you take advantage of an editor that has the ability to color-code your source file. This makes it much easier to detect syntax errors. Also, whatever editor you use, you *must* save the source code in plain text format (i.e., without any embedded formatting such as is inserted by Microsoft Word using its native format).

1 AN INTRODUCTION TO VCL

It is standard practice to introduce a new programming language with its version of the "Hello" program. This approach is popular because it is simple; it introduces the print statement, which is often your primary method of debugging; and (in this case) it serves as an excellent means of demonstrating how to use the WinVCL interface to compile and download programs.

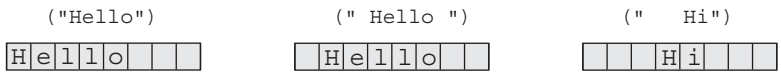
1.1 "HELLO"

The simplest version of "Hello" in VCL is:

```
put_spy_text(" Hello ")
while (1=1) {}
```

The first line of this program will send the word *Hello* through the system's serial port, suitably encoded for a Curtis 840 Spyglass display — which is the default display for VCL-enabled products.

The Spyglass has a single line, eight-character display. The space after the opening quotation mark in `(" Hello ")` moves the text into a more centered position in the eight-character display by making the first character be a blank space. (The blank spaces after the word are moot, but are often included anyway.)



The second line of this program is an infinite loop. Without this instruction, VCL would find the end of the program block, which is considered an error. VCL has been designed to be as robust and safe as possible. Executing instructions beyond the end of a program block is definitely not safe!

So far, our program prints only *Hello*—not *Hello World*. The problem is that we want to show an 11-character message on an 8-character display. There are two obvious solutions. We could scroll the display, or we could show the message sequentially with a time delay between the two words. Here is a version of the program using a time delay.

```
while (1=1)
    {
    put_spy_text(" Hello ")
    setup_delay(DLY1,500)
    while (DLY1_Output <> 0(x{}
    put_spy_text(" World ")
    setup_delay(DLY1,500)
    while (DLY1_Output <> 0) {}
    }
```

We now have an outer controlling loop (the first "while (1=1)"). There is a second print statement to issue the *World* part of the message, and there are two Delay statements. The standard VCL library contains a variety of common functions; see the *VCL Common Functions* manual. In this case, we use the

function setup_delay() to allocate a timer (DLY1) and assign it a delay of 500 milliseconds. We then monitor the output of that timer, waiting until it counts down to zero. Our display now displays *Hello*, waits half a second, displays *World*, waits half a second, and then restarts.

1.2 COMPILING AND DOWNLOADING "HELLO"

In order to make this program run in the target system, you will have to compile the program and then download it into the target system. To do either of these tasks, you must first create a new "project" to identify all the components that will be used to create this program.

We assume that you have already installed WinVCL on your system and have read the *WinVCL User's Guide*. The Hello source code and basic project definition were automatically setup for you. Make sure that you have also installed the appropriate operating system for the controller you are using.

Following the instructions in Sections 3.1 and 3.2 of the *WinVCL User's Guide*, build and download the program. Once the program has been downloaded, you can use the Spyglass to view the program's output. (You can also send serial data to any terminal device, such as a "dumb terminal" or a PC running HyperTerminal.)

2 THE VCL CORE LANGUAGE

The core language is the first of several levels of functionality within VCL. You'll notice that the core language looks like every other procedural language you have used (e.g., Basic, Fortran). We have kept the core operations simple, so you'll soon be able to move on to the extended aspects of VCL.

2.1 THE FORMAT OF A VCL PROGRAM

VCL reads numerical and alphabetical characters. It does not see spaces, blank lines, or comments; with alphabetic characters, it does not see case. You can take advantage of VCL's "blind spots" for your own purposes. Use tabs and spaces; use upper and lower and mixed case; use comments. This formatting, which VCL cannot see, will make your program much more readable to you and your fellow humans.

VCL's comment character is a semicolon (;). The remainder of the line following the semicolon is invisible to VCL. We encourage you to use comments freely to make your programs more readable and your intent clear.

2.2 CONSTANTS AND STRINGS IN VCL

VCL has two types of numeric constants: numeric constants and symbolic constants. Both types are 16-bit, signed integers (-32768 to 32767).

Numeric constants are expressed in decimal by default. Numeric constants can also be expressed as hexadecimal values using the *0x* prefix as in C, or with an *h* suffix as in many assembly languages. If you use the trailing *h* notation and your value starts with an alpha character (A-F), you must prefix the value with a zero (i.e., *0Ah* is valid, but *Ah* is not). Numeric constants can also be expressed as binary values by using a trailing *b*. So, to express the decimal value 12, you have these options:

```
12 or 0xC or 0Ch or 1100b
```

Symbolic constants allow you to substitute a symbolic name for a numeric constant. The symbol may be any combination of letters, digits, and the underscore (_). The form of a symbolic constant declaration is:

```
constant-name        constant value
```

where *constant-name* is the symbolic name you wish to declare, *constant* is a VCL keyword, and *value* is either a numeric constant as previously described or another symbolic constant. For example, to declare "twelve" as a numeric constant, you would write:

```
TWELVE    constant 1100b
```

VCL also has limited support for strings. There are no string operators in VCL; however, some functions require them. In order to use a string, you must declare a symbolic reference to it. Then, you will use the symbolic name for the string. The symbol may be any combination of letters, digits, and underscores. The form of a symbolic string reference is:

string-name string “string in double-quotes”

Where *create* is a VCL keyword, *string-name* is the symbolic name you wish to declare, and *string* is a VCL keyword, which is followed by the string delineated by double quotes. Strings must fit on a single line and be less than 65,000 characters long. For example, to declare a string named HELLO, which has the value of Hello World, you would write:

```
HELLO    string "Hello World"
```

VCL accepts everything within quotation marks, with no interpretation; this means that case, punctuation, and blank spaces will be kept exactly as typed.

2.3 VARIABLES IN VCL

VCL provides two kinds of variables: normal variables and bit-variables. Normal variables are 16-bit, signed integers (-32768 to 32767). Bit-variables are also 16 bits, but with the expectation that only certain bits will be available; see Section 2.4.

All variables in VCL (with the exception of local variables) are pre-allocated. Most are pre-allocated as predefined variables within VCL function groups; see *VCL Common Functions* manual. Some, however, are pre-allocated as user variables.

2.3.1 User# Variables

The pre-allocated user# variables (user1 to user120) are for you—the user—to define. Typically you will want to use variable names that have more descriptive power than simply user#. To do this you create a symbolic variable name using the following format:

variable-name equals pre-allocated user variable

where *variable-name* is the symbolic name you wish to declare. This will cause one of the pre-allocated variables to be assigned the variable-name, which you can then use in your program. For example, to create the variable-name “temp” you could write:

```
temp    equals user1
```

You can now assign values to *temp* and use it in your code.

Note: Keep careful track of your variable assignments. If you accidentally define the same variable twice, the new will override the old—and you will not be alerted. In other words, if you write:

```
mouse    equals user43
```

and then later write

```
cat    equals user43
```

user43 will be *cat*, and the earlier code you wrote for *mouse* will not work.

2.3.2 P_user# Variables

These variables can be used in defining programmable parameters in the 1311-accessible parameter block; see Section 5.

2.3.3 Local Variables

Another set of variables is available for you to use. These variables, called local variables, are used only within VCL modules; see Section 2.7. Because they are "hidden" within modules and not in the public part of your VCL code, local variables do not come out of your limited supply of user# variables. And because VCL modules are self-contained, you can use the same local variable name in different modules to mean different things.

2.4 BIT-VARIABLES IN VCL

VCL is primarily concerned with embedded system control, and hence with the isolation and manipulation of bits.

Suppose you have defined user3 as switches:

```
switches     equals user3
```

You can then explicitly state a bit within this variable:

```
temp = switches.2    ; Isolate 2nd least significant bit
```

VCL interprets a period followed by a number as a request to isolate that bit. The number after the period is commonly referred to as a bit "mask" as it masks off all but the bits specified.

Alternatively, you can make an implicit instruction, defining a bit in much the same way you define a variable. To create a symbol "Interlock_Sw" that references bit 2 of user3, you would write:

```
Interlock_Sw       bit switches.2
```

You can now use the symbol in the same way as you would have with the explicit bit notation given previously. For example,

```
temp = Interlock_Sw      ; Isolate 2nd L.S.B.
```

Use whichever method (explicit or implicit) that makes your program most readable.

2.5 SCOPE OF SYMBOLIC NAMES

"Scope" is the visibility of a given name. In some languages, all variable names have global scope. For example, in the first versions of Basic (and in most assembly languages), all names were visible from all parts of the program. In more modern languages, names are limited in visibility—depending on where they are declared. In C, for example, a name declared *outside* a function is globally visible (visible everywhere), whereas a name declared *within* a function is visible only within that function.

VCL has a construct called a module, which is described in Section 2.7. Any variable name declared outside a module is globally visible, whereas any name declared within a module is visible only within that module. You can declare the variable *temp* within three different modules and each of these local variables will be used only within the module in which it was declared.

2.6 PROGRAM FLOW CONTROL

VCL has a modest set of program flow control statements, including an *if-then-else* structure, a *while* loop, a *go-to* instruction, and a simple *call-return* subroutine.

VCL also has the full set of conditional operators you have come to expect, including:

=	equal to	<>	not equal to
<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to

2.6.1 If-Then-Else

VCL's *if-then-else* structure should come as no surprise. The conditional part must be enclosed in parentheses, and the code that is controlled by the conditional must be enclosed in curly braces({}). Conditional phrases are made up of one or more conditional clauses combined with the keywords *and* and *or*. For example, the simplest *if* conditional looks like this:

```
if (user3 = 14)
    {
    controlled code
    }
```

A more complex example:

```
if (((user4 = 0x12) and (user2 = 0x13)) or (user5 = 0x17))
    {
    some code
    }
else if (user5 <> 32)
    {
    some other code
    }
else
    {
    if all else failed -- more code
    }
```

2.6.2 While

The primary means for expressing a loop construct in VCL is the *while* statement. The conditional phrases and clauses are identical in format to those in the *if* statement. As long as the initial conditional statement evaluates to True, the loop will be executed. For example:

```
while ((my_variable <> 0) and (user2 = 0x13))
      {
      do this code
      }
```

It is perfectly permissible for the body of the conditional to be empty. For example, to make the system wait until user3 is non-zero, you would write:

```
while (user3 = 0) {}
```

Although you can format this statement like the previous example, with the braces on new lines and indented, the single-line format is preferred for void operations, i.e., {no operation}.

2.6.3 Go-to

Although much maligned, the *goto* instruction is nevertheless essential in certain circumstances. The *goto* always references a label, which is a combination of letters, digits, and underscores, terminated with a colon. For example, you can setup a basic infinite loop by writing:

```
Main_Loop:
    your code goes here
    goto Main_Loop
```

Use the *goto* with restraint. The example of the *goto* would be much better if it were coded instead as an infinite loop (i.e., while (1=1) { … }) because there would be no label to maintain.

2.6.4 Call-Return

Another type of VCL program flow construct is the subroutine. The format is identical to most assembly language implementations. The *call* instruction references a label, which is a combination of letters, digits, and underscores, terminated with a colon. When the call is executed, control transfers to the next instruction after the label and continues until a *return* instruction is executed. For example, to call a subroutine named “Do_Something” you would write:

```
call Do_Something
    code here
    more code here
    still more code here
Do_Something:
    the code controlled by Do_Something
return
```

This should be very familiar from your previous programming experience.

2.6.5 Include “filename”

You can separate your program into a number of files and then combine them using the *include* statement. The contents of the included files are inserted in place of the *include* statement.

It is likely you will have a set of definitions that are common to some or all of your applications. You can put these definitions into a file and then include the file with each application. For example, let us assume you put the definitions into a file called “common_defs.inc”, which is located in a directory called “\projects\common”:

```
include "\projects\common\common_defs.inc"
```

VCL uses three strategies to find an *include* file. First, it simply uses the name supplied. In this case, we’ve given the full file name and path explicitly. If VCL fails to find the file using the name specified, it will look for the file in the project directory; so, to include a file in the project directory, simply give its name without path information. Finally, VCL will look for the file in the operating system directory.

2.7 MODULES

A VCL module is a section of code contained between the keywords *begin_module* and *end_module*. The code in the module is accessed with a single common name using the special keyword *enter*. This is somewhat like a subroutine statement; however, a module may contain multiple subroutines. Finally, you can return from any point in the module using the keyword *exit*.

2.7.1 Begin_Module, End_Module

The start of a module is indicated by the keyword *begin_module* followed by the module’s name. The end of the module is indicated by the keyword *end_module*. The general format is:

```
begin_module entry-point-name
    module code (any valid VCL statement)
end_module
```

Here is an example:

```
begin_module Show_Plus_Half
    call Add_Half
    put_spy_dec('a',user17,'z')
    exit
Add_Half:
    user17 = user17 + (user17/2)
    return
end_module
```

Single quotation marks are used around single characters or spaces.

Note: You may have multiple modules within your program, but you may not create a module within a module (i.e., modules may not be nested).

2.7.2 Enter-Exit

You call a module just as you would a subroutine, with an *enter* statement. In the example code, the module would be called by writing:

```
enter Show_Plus_Half
```

You return from a module just as you would from a subroutine, with an *exit* statement. (You can use a *return* statement, but the use of the *exit* statement is strongly encouraged.) You can exit from anywhere within the module by using the *exit* statement. For example, if the Add_Half routine had an *exit* statement, rather than a *return* statement, control would transfer out of the module and back to the next instruction following the preceding *enter* command.

It is important to understand that the only access to any of the code in a module is by way of the module's entry-point name. All the subroutines that are contained within the module—between the *begin_module* and *end_module* statements—are invisible to the code outside the module. (In fact, trying to call a routine defined within a module from code located outside that module will result in an error.) This generally makes your code easier to understand and to maintain.

3 VCL FUNCTIONS

VCL has a number of predefined functions, which are the equivalents of library routines in other languages. However, there are some significant differences between VCL functions and the library routines you have encountered previously.

3.1 OVERVIEW OF VCL FUNCTIONS

VCL functions allow you to access parts of the system in a product-independent manner (e.g., access to switch inputs). They reduce the amount of code you need to write by taking the place of certain instruction sequences (e.g., a random number generator). Finally, as you will see in Section 4, functions always execute in a single cycle.

Functions always return a single value. They may or may not have parameters. For example, recall the put_spy_text(1) function from the introduction:

```
while (put_spy_text(" Hello ") = 0) {}
```

The function put_spy_text() takes a single parameter (in this case, the word Hello). It returns a value of 0 when it does not accept the string for display. We can then "hang" here, using a while-statement, until the string is accepted.

3.2 PECULIARITIES OF FUNCTIONS WITHIN VCL

There are some significant differences between VCL functions and those you have encountered before: (a) VCL functions are predefined, (b) there are often several instances of a function, and (c) parameters cannot be the results of other functions.

All VCL functions are predefined. The fact that VCL functions are predefined has some other interesting consequences. Again, using our initial example, recall that we defined a time delay by writing:

```
setup_delay(DLY1,500)
while (DLY1_output <> 0) {}
```

In particular, notice the parameter DLY1. In many other languages, you would declare a time delay by starting a new copy of a delay function. This is very flexible, but it leaves you open to encountering run-time errors (as a result of insufficient memory) that may not be found without exhaustive testing.

Functions often have several instances. VCL takes a more object-oriented approach by pre-declaring the function and then allowing multiple explicitly named copies of it to be run. In the previous example, the delay function allows you to setup and monitor a count-down timer that has a resolution of 1ms. And, in that example, we used the first (DLY1) of several possible delay timers.

Also, notice that the output of the function is available in the variable DLY1_ output. This is typical of VCL functions. In fact, this is a very important aspect of VCL functions.

Parameters cannot be results of other functions. VCL functions also differ from those you've encountered before in that you cannot directly use the result of a function as an input to another function. Consider the VCL display function called put_spy_bin() that accepts a single parameter, a value to be displayed in binary. If you wished to display a random number in binary (yes, there is a VCL function called "random()"!), you might expect to be able to write:

```
put_spy_bin(random())          ; <- This will not work
```

But this is not the case. VCL functions cannot have a function as a parameter. The correct way to perform this operation is to move the result of the random function into a variable and then use that variable as a parameter. For example:

```
random_output equals user18

random_output = random()
put_spy_bin(random_output)     ; <- This is correct
```

3.3 THE SYSTEM INFORMATION FILE (SYSINFO) AND FUNCTIONS

Each operating system has a system information file, generically referred to as SysInfo. This is an HTML file that is distributed with the operating system. The name of the system information file is a combination of the operating system's name and the current version number. For example, the 1310's system information file is currently at revision level "aL"; its SysInfo file is called os1310aL.htm. You can access this file using WinVCL Info button on the main display.

You will use the SysInfo to find out what functions are available, what parameters they take, and what error codes they generate. The structure and contents of SysInfo files are more fully explained in Appendix E. Here, we will give only a brief description of how to read a function description.

Related functions are grouped together. For example, all the time delay functions are grouped in a single section. Sections always start with a line running across the page, followed by a title line that identifies the function group, starting with its 3-letter designation (in this case, DLY). Typically there will be a description immediately following the title line, followed by the group's constants, variables, and functions.. Here's a summary of the Time Delay function group:

DLY - Time Delay Functions

-The following definitions are related to the time delay functions.

Delay Access IDs

- These IDs are used with the setup_delay() function to specify a particular delay timer.

```
Constant    0 (0x0000) DLY1              Delay 1
Constant    1 (0x0001) DLY2                "   2
Constant    2 (0x0002) DLY3                "   3
Constant    3 (0x0003) DLY4                "   4
...
```

(etc.; the DLY group has 16 instances, all of which are listed in the SysInfo)

This will be followed by descriptions of other constants and variables connected to this function group. The DLY group has only one additional data item: the variable DLY#_output.

Finally, you will find one or more function definitions. Here is the definition of the setup_time_delay() function:

SETUP_DELAY(2) – Setup a Time Delay

- This function installs a new time delay.

Parameters

```
DLY#       Holds the ID of the desired delay.
Ticks      Number of ticks to delay (ticks are 1ms, by default)
```

Returns

```
0          Function did not execute.
1          Function successfully executed.
```

Errors

```
BAD_ID     DLY# out of range.
```

The first line holds the function name, the number of parameters it takes (in parentheses), and brief description of what the function does. This is usually followed by a slightly longer description of the function, which is followed by three subsections: Parameters, Returns, and Errors.

The Parameters section describes each of the parameters required by the function. In this case, there are two parameters: *DLY#* and *Ticks*. We see that *DLY#* holds the ID of the delay (DLY1, DLY2, …) and that *Ticks* holds a count, in milliseconds.

The Returns section describes values that the function can return. If, for some reason, this function does not execute (say, because of an error) it will return a value of zero.

The Errors section describes all the errors that can be generated by the function. In this case, there is only one possible error: if the ID is not valid, an error will be reported.

See Appendix E for a more complete description of SysInfo files.

3.4 EXAMPLE OF HOW TO USE A FUNCTION

It's common in the early stages of system development to write short test programs simply to verify that the hardware is correctly wired and working. Let's assume your system has a three-wire pot connected to the second pot input and that you wish to display its value on the Spyglass. This program will do the test:

```
pot_input equals user10             ; Holds the present pot value

setup_pot(POT2,THREE_WIRE)   ; This is a 3-wire pot

while (1=1)                         ; Loop: read-pot then display
    {
    pot_input = get_pot(POT2)
    while (put_spy_dec(' ',pot_input,'b') = 0) {}
    }
```

First, we define a variable to hold the value of the pot. Then we tell the system which pot input to look at and what kind of pot it is. The constants POT2 and THREE_WIRE are predefined in the controller's SysInfo and can be found in association with the setup_pot() function's definition. The system won't read a value unless the pot input has been configured with the setup_pot() function. Then we fall into an infinite loop. In the loop, we read the value of the pot into our temporary variable (remember, you can't use a function as a parameter), and then we output the value to the Spyglass. Notice that we use a while-statement to make sure each new value is really transmitted. The put_spy_dec() routine takes three parameters; the first and last are single characters which will be displayed on the far left and right of the six-digit integer display.

4 CHAINING VCL FUNCTIONS

Perhaps the most powerful aspect of VCL is its ability to chain commands together. Chaining allows you to setup signal paths such that processing and data transfers are done automatically (i.e., run periodically).

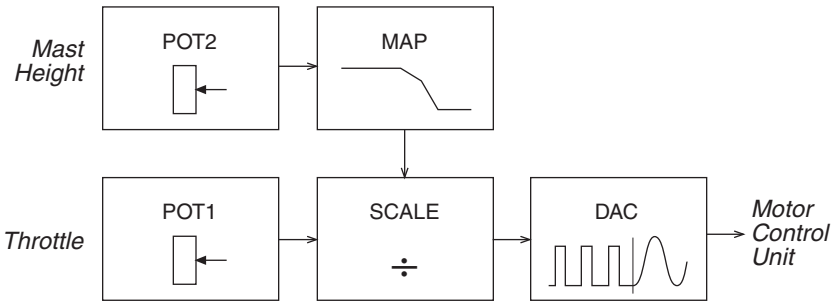
4.1 OVERVIEW OF CHAINING

Programming languages can be divided into those that have no sense of time other than before and after (i.e., this instruction came after that last instruction but before the next instruction) and those that respond in real-time, according to some clock. In general, programs that respond in real-time are several orders of magnitude more complex than their clockless counterparts. VCL, however, has been designed to hide much of this complexity without sacrificing performance. VCL's ability to chain commands is integral to this simplification.

4.2 CHAINING — AN EXAMPLE

It is often the case that you need to acquire a signal and then perform several modifications to it before sending it to its destination. For example, consider the problem of controlling the throttle in a forklift truck. The basic throttle signal comes from the accelerator pedal. However, it is typical to limit that signal depending on the height of the forks (i.e., the load). So, you might allow full speed if the load is very close to the pavement, but limit the vehicle's speed as the load is raised (and the center of gravity with it).

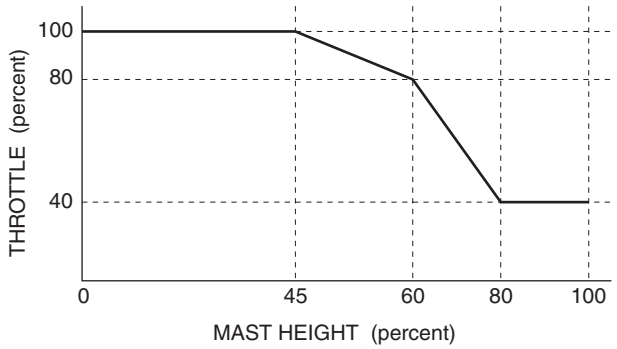
Assume that we have a pot (POT1) for our throttle control, a pot (POT2) to tell us how high the forks are, and an analog output (DAC1) to send to the motor control unit. Our signal chain might look something like this.



Most functional blocks shown in this signal chain diagram should be self-evident; however, the Map block deserves some explanation.

In VCL a map has an X input, a Y output, and up to seven point pairs (X-Y) that specify the relationship between X and Y by using linear interpolation between points. For example, suppose we want the throttle to be 100% up to 45% mast height. Then we want to reduce the throttle to 80% of its requested value up to

60% mast height. Then we want to reduce the throttle to 40% of its requested input when the mast height is at 80% of its upper limit (see graph).



When you first setup the map, you define the relationship of inputs to output by specifying up to seven pairs of points. You start by defining how many point pairs you will actually use, then you specify the point pairs, giving X first, followed by its corresponding Y, and setting unused points to zero.

Here is the setup_map() function definition for this particular mapping (assuming that the largest integer=32767 is 100%).

```
setup_map(MAP1, 5, ; Number of points used
          0, 32767, ; X = 0%  Y = +100%
      14746, 32767, ; X = 45% Y = +100%
      19660, 26214, ; X = 60% Y = +80%
      26214, 13107, ; X = 80% Y = +40%
      32767, 13107, ; X =100% Y = +40%
          0,     0, ;(point pair not used)
          0,     0) ;(point pair not used)
```

If we were going to write this in conventional programming style, we would read the encoder (mast height=POT2), and then re-map (MAP1) the value. Then we would read the pot (POT1) and scale it (SCL3) using the re-mapped mast-height value. Finally, we would send the scaled value to the analog output (DAC1). Here is one version of this program.

```
temp1 equals user2

setup_pot(POT1, THREE_WIRE)
setup_pot(POT2, THREE_WIRE)

setup_map(MAP1, 5, ; Number of points used
          0, 32767, ; X = 0%  Y = +100%
      14746, 32767, ; X = 45% Y = +100%
      19660, 26214, ; X = 60% Y = +80%
      26214, 13107, ; X = 80% Y = +40%
      32767, 13107, ; X =100% Y = +40%
          0,     0, ;(point pair not used)
          0,     0) ;(point pair not used)

while (1=1)
       {
       temp1 = get_pot(POT2) ; Map mast height into scale factor
       temp1 = get_map_output(MAP1,temp1)
       setup_scale_factor(SCL3,temp1)

       temp1 = get_pot(POT1) ; Get the throttle and scale it
       temp1 = scale_value(SCL3,temp1)

       put_dac(DAC1,temp1)   ; Output the rescaled throttle to the DAC
       }
```

This isn't so bad as it stands; however, it is rarely the case that the only thing the system is doing is scaling a pot. What happens to your throttle response

when you have hundreds or even thousands of instructions to process? Here is the same signal processing using *automate* instructions to chain the functional blocks together:

```
setup_pot(POT1, THREE_WIRE)
setup_pot(POT2, THREE_WIRE)

setup_map(MAP1, 5,    ; Number of points used
          0, 32767,   ; X = 0%   Y = +100%
      14746, 32767,   ; X = 45%  Y = +100%
      19660, 26214,   ; X = 60%  Y = +80%
      26214, 13107,   ; X = 80%  Y = +40%
      32767, 13107,   ; X =100%  Y = +40%
          0,     0,   ;(point pair not used)
          0,     0)   ;(point pair not used)

automate_map(MAP1,POT2_output)
automate_scale(SCL3,POT1_output,MAP1_output)
automate_dac(DAC1,SCL3_output,1,0)

while (1=1) {}
```

The automate_map() function causes the indicated value to be automatically read and placed in the selected map's output variable.

The automate_scale() function uses the map output to scale the pot input and leaves the value in its output variable (SCL3_output).

Finally, the automate_dac() function picks up the scaled value and sends it out the analog pot.

Notice that the main line is now empty. All the functions, once they have been chained, will operate automatically—without any further program intervention.

5 SUPPORT FOR THE CURTIS 1311 HANDHELD PROGRAMMER

The 1311 handheld programmer can be connected to a Curtis motor controller to allow users to change selected values, to monitor values, and to see error reports. Among other benefits, this allows generic code to be written that can subsequently be customized in the field to a particular application. This section describes how to add these capabilities to your VCL program.

5.1 PARAMETER DEFINITION OVERVIEW

When you plug the handheld programmer into the controller, it powers up and sends a message asking for a parameter block. The parameter block fully describes the information that is to be exchanged between the 1311 and the device to which it's connected. (Note: The parameters in the parameter block are "programmable parameters" — accessible via the 1311 programmer.)

The parameter block is built using definitions that appear within the comment fields of your VCL program. All definitions begin with the parameter keyword *parameter_entry* followed by a name. Entries are always terminated by the keyword *end*. Between these two keywords are a number of elements that describe what kind of entry you are making—e.g., some kind of menu entry or data entry. For example, here is an entry (put into the comment field, as you can see) that puts a new submenu under the Program menu. Parameter entries are always made in the form of comments; the VCL compiler turns them into parameter block files for you. See Appendix C for a complete list of parameter keywords.

```
;
; parameter_entry    "Trim"
;    type            Program
;    address         user12
; end
;
```

Notice that the entry starts with *parameter_entry* and terminates with *end*. Other than that, the format is very free. Keywords are case insensitive; however, it is customary to put keywords and variables in lower case, constants in uppercase, and parameter names in mixed case. In all situations where you are given the freedom to format, you should determine the style you find most readable and then use that style consistently.

The 1311 has a number of predefined top-level menu labels, among which are Program, Monitor, and Fault. You can create your own submenus and entries under each of these predefined menus. We'll explain how to create submenus, then how to create individual Program, Monitor, and Fault entries. But first, we'll explain how to create parameter names.

5.2 CREATING PARAMETER NAMES

Parameter names can use any of the letters of the alphabet, the digits 0-9, and these fifteen symbols: `! % ^ & ( ) - + | , . / < > ?`

However, to conserve memory in the 1311, you should try to build your parameter names using the following words, which are in the programmer's stored dictionary and therefore use only 1 byte apiece. For example, a parameter named *rhinoceros* would use 10 bytes, whereas *resistance*—which is in the stored dictionary—uses only 1 byte.

Accel	Duty	Level	Pulses	State
Adjust	Emergency	Lift	Pump	Stator
Amps	Empty	Limit	P^W^M	Steer
Anti	Enable	Load	Quick	Stop
Armature	Encoder	Lock	Ramp	Suppression
Auto	External	Low	Range	Switch
Aux	Factory	Lower	Rate	System
Battery	Failsafe	Main	Ratio	Taper
Boost	Fault	Map	Regen	Temp
Brake	Field	Max	Reset	Test
Braking	Filter	Meter	Resistance	Thermal
Calibration	Forward	Min	Restraint	Throttle
Check	Frequency	Missing	Return	Tiedown
Clamp	Full	Mode	Reverse	Time
Close	Gain	Modulation	Right	Timeout
Code	Gear	Motor	Rollback	Traction
Coeff	Hardware	Neutral	Rotor	Tremor
Coil	Heatsink	Node	Seat	Turn
Comp	High	Nominal	Select	Type
Constant	Hill	Open	Sensitivity	Under
Contactor	Hold	Option	Sequencing	Value
Control	Hour	Output	Series	Valve
Creep	H^P^D	Over	Service	Variable
Crnt	Hysteresis	Parameter	Setting	Vehicle
Current	Impedance	Pedal	Shape	Voltage
Cutback	Inductance	Plug	Short	Warning
Cycle	Inhibit	Poles	Shunt	Weaken
Deadband	Init	Position	Sleep	Weld
Decel	Input	Pot	Slip	Wheel
Delay	Interlock	Power	Soft	Winding
Diagnostic	Joystick	Precharge	Spare	Wiper
Direction	Key	Procedure	Speed	Wiring
Disable	Latch	Program	S^R^O	
Drive	Left	Pull-in	Start	

An up-arrow (^) in the name-string forces the following letter to upper case:

```
; parameter_entry    "Throttle^Enable"
```

In this example, we will see ThrottleEnable on the display. Without the up-arrow in the name-string, we would see Throttleenable on the display. If the first letter of the name-string is capitalized, it will display as a capital without the use of the up-arrow; letters following an up-arrow will be capitalized whether or not they are typed as capitals. Thus, "P^w^m" will display as PWM.

5.3 SUBMENU ENTRIES

The 1311 handheld programmer displays information under a series of menus. It is under the Program, Faults, and Monitor menu levels that you can create your own submenus—and, ultimately, your own data entries.

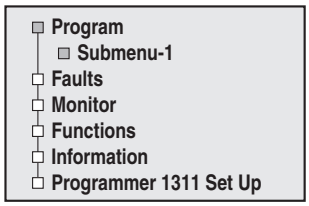
To create a submenu entry, you must specify the name of the entry (using the keyword *parameter_name*), the menu under which it is to be placed (using the keyword *type*) and the indentation level (using the keyword *level*).

Submenu Entry Keywords

KEYWORD	VALUES	OPTIONAL
parameter_entry	"name-string"	no
type	Program, Monitor, Faults	no
level	1-6	no
end	—	no

For example, suppose you want to create a submenu under the Program heading with the title "Submenu-1". You would simply enter:

```
; parameter_entry    "Submenu-1"
;     type           Program
;     level          1
; end
```



The resulting screen will look like this, after the Program menu has been expanded once.

You can create submenus of your own submenus if you like. For example, you could have added another submenu parameter entry immediately following the one above, identical except for a different label ("Submenu-2") and the value of Level, which would be set to 2. This would have caused a submenu to appear under "Submenu-1", with one more level of indentation. However, if you create two submenus that have the same level number but have no intervening parameter definitions, only the last submenu will appear. Also, be aware that the keyword *Level* only appears within a submenu definition (i.e., it does not appear within the definitions of Program, Monitor, or Faults entries).

Submenus can greatly add to the usability of your definitions both by limiting the number of keystrokes it will take a user to get to a particular element and by controlling screen clutter.

5.4 COMMON ASPECTS OF PROGRAM, MONITOR, AND FAULTS DEFINITIONS

Program, Monitor, and Faults definitions use many of the same keyword definitions, and they have many of the same options.

5.4.1 Basic Keywords (Program_Entry, Type, and End)

Just like the submenu entry, the Program, Monitor, and Faults definitions all require the keywords *parameter_entry*, *type*, and *end*. Every parameter needs to be named using the keyword *parameter_entry* (although names need not be unique). The keyword *type* is the primary means used to distinguish between Program, Monitor, and Faults definitions. Every entry must be terminated with the keyword *end*.

Unlike the submenu entry, you cannot use the keyword *Level*. The only place where the keyword *Level* appears is within a submenu definition.

5.4.2 Other Basic Keywords (Signed, DecimalPos, Units)

Values can be signed or unsigned. By default, values are unsigned (*signed* = No). A normal VCL variable, which is 16 bits wide, will have a range from 0 to 65535. If you were to set *signed* to Yes, the same variable would have a range of displayed values from -32768 to +32767.

The keyword *DecimalPos* is used to place a decimal point. *DecimalPos* takes a numeric argument from 0 to 3. By default, *DecimalPos* has a value of 0, which suppresses display of the decimal point. Any non-zero value for *DecimalPos* specifies that a decimal point is to be displayed. The actual value of *DecimalPos* tells the 1311 how many places from the left the decimal point should be placed. For example, if the normal display value is 1234 but you set *DecimalPos* to a value of 2, the value displayed on the 1311 would be 12.34.

The keyword *units* is used to apply common labels to a value. See Appendix D for a list of the available standard parameter unit labels.

5.4.3 Variable Characterization Keywords (Address and Width)

Program, Monitor, and Faults definitions all require the keyword *address*; the Faults definition requires an *alt_address* as well. The *address* refers to the symbolic name of the variable you wish to reference. Although you can reference any variable in the system, most of the time you will be referencing one of the general-purpose user variables: user1 through user120. You will notice that there are an additional one hundred user variables with the prefix “p_” (p_user1 through p_user100). The *p* stands for “persistent”. Writing to a persistent variable from within your VCL program is the same as writing to a non-persistent variable; however, when the 1311 writes to a persistent variable the value is saved and will be restored when the system powers up. Persistent variables are useful for holding configuration values and some status values, like error history.

Program, Monitor, and Faults definitions all require the keyword *width*. The *width* refers to the size (in bits) of the variable being referenced. Usually, the width will be set to 16bit, the default size of VCL variables. The exception to this is when you define a bit variable. In this case, width must be set to 8bit; and you can only reference the predefined 8-bit variables (user_bit1 through user_bit10 and p_user_bit1 through p_user_bit10).

5.4.4 Value-Limiting Keywords (MaxRaw, MinRaw)

You can use the 1311 to alter the value of a variable. The keywords *MaxRaw* and *MinRaw* limit the range by which the variable can be altered by the 1311. By default, you will be able to use the 1311 to alter a given variable over its intrinsic range. For example, a normal VCL variable is 16 bits wide; it can have values from 0 to 65535 (or -32768 to +32767 if it is signed). However, you may want to limit the value of your variable to a range of 1024 to 4096. To do this, you would set MinRaw=1024 and MaxRaw=4096. The user would then be unable to use the 1311 to set the variable to a value outside this range.

5.4.5 Value-Display Keywords (MaxDsp, MinDsp)

You can use the 1311 to alter the displayed value of a variable. By default, the 1311 displays the raw value. For example, let us suppose that you have used *MinRaw* and *MaxRaw* to limit the value of your variable to a range of 32 to 1456. The displayed value on the 1311 will also range between 32 and 1456. But, let us suppose that you really want the 1311 to display values of 0 to 125. You can do this by setting *MinDsp* to 0 and *MaxDs*p to 125. Now, when the actual (raw) value is 1456, the 1311 re-scales it and displays the result as 125.

5.5 PROGRAM AND MONITOR DEFINITIONS

These two types of definitions are identical in format. In use, the Program menu contains elements that you want the user to be able to modify. For example, you might have some key value that, when set to a particular value, will enable a product feature. On the other hand, Monitor menu entries are read-only and are used, literally, to monitor variables (or bits) of interest. There are two basic forms for these entries: variable (16 bits) and bit (1 bit).

5.5.1 Program and Monitor Definition Keywords for Variables

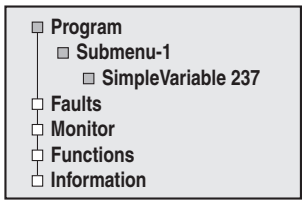
These are the keywords for defining 16-bit variables:

Program and Monitor Definition Keywords for 16-bit Variables

KEYWORD	VALUES	DEFAULT	OPTIONAL
parameter_entry	"name-string"	—	—
type	Program or Monitor	—	—
width	16bit	—	—
address	"address-string"	—	—
MaxDsp	*Integer*	*65535*	*yes*
MaxRaw	*Integer*	*65535*	*yes*
MinDsp	*Integer*	*0*	*yes*
MinRaw	*Integer*	*0*	*yes*
signed	*YES, NO*	*NO*	*yes*
DecimalPos	*0, 1, 2, 3*	*0*	*yes*
units	*"units-string"*	—	*yes*
end	—	—	—

As an example, let us assume that you wish to create a simple definition for the variable "p_user3" and that you want to allow the user to modify this value. You could do this with the following definition:

```
; parameter_entry    "Simple^Variable"
;     type           Program
;     width          16bit
;     address        p_user3
; end
```



If this were entered after the Submenu-1 definition, the 1311 display would look like this (assuming the present value of SimpleVariable is 237).

Because p_user3 has been used as the address, changes to the value of p_user3 will be remembered by the system and automatically restored on power-up. If we had specified user3 as the address, the value could be changed by the 1311, but it would revert to zero (or whatever value you set in your VCL initialization code) every time the power is cycled.

5.5.2 Program and Monitor Definition Keywords for Bits

Defining bit-variables is very similar, with some obvious exceptions. You must specify the bit of interest using the keyword *BitSelect*. You have the option of having the 1311 display an On message when the bit is high (default) or when it's low, using the keyword *BitActiveLow*. Of course, the options that apply specifically to an integer (*MinRaw*, *MinDsp*, *MaxRaw*, *MaxDsp*, *signed*, *DecimalPos*, and *units*) don't apply when defining bit-variables.

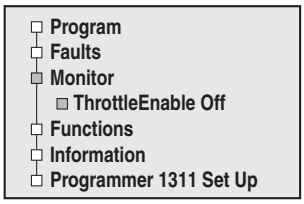
Here are all the options that apply when defining bit-variables:

Program and Monitor Definition Keywords for Bits

KEYWORD	VALUES	DEFAULT	OPTIONAL
parameter_entry	"name-string"	—	—
type	Program or Monitor	—	—
width	8bit	—	—
address	"address-string"	—	—
BitSelect	0-7	—	—
BitActiveLow	*YES, NO*	*NO*	*yes*
end	—	—	—

For example, suppose you wanted to monitor the predefined bit-variable user_bit5 using the label ThrottleEnable. Also, suppose that this bit is true when its value is low. The following definition would do this.

```
; parameter_entry    "Throttle^Enable"
;     type           Monitor
;     width          8bit
;     address        user_bit5
;     BitSelect      2
;     BitActiveLow   YES
; end
```



Notice that user_bit5 was specified as the address. You must use a bit-variable (user_bit1 to user_bit# or p_user_bit1 to p_user_bit#) with a bit definition—otherwise the 1311 will issue an error when it tries to access the variable. In this example, the value of ThrottleEnable must be high because BitActiveLow is set to Yes.

5.6 FAULTS DEFINITIONS

Faults definitions, like Monitor definitions, are read-only. Fault definitions always refer to bits. In fact, Faults definitions refer to two bits: the present fault bit, and the history fault bit. With the exception of the keyword *alt_address*, the Faults definition looks identical to a Monitor definition for a bit.

Faults Definition Keywords for Bits

KEYWORD	VALUES	DEFAULT	OPTIONAL
parameter_entry	"name-string"	—	—
type	Faults	—	—
width	8bit	—	—
address	"present-address-string"	—	—
alt_address	"history-address-string"	—	—
BitSelect	0-7	—	—
BitActiveLow	*YES, NO*	*NO*	*yes*
end	—	—	—

For example, suppose you wanted to declare a bit as MyFault and that you are keeping that bit's present value in user_bit5 and the history in p_user_bit5. The definition would look like this:

```
; parameter_entry    "My^Fault"
;     type           Faults
;     width          8bit
;     address        user_bit5
;     alt_address    p_user_bit5
;     BitSelect      4
;     BitActiveLow   YES
; end
```

Notice that you must use bit variables. Also, notice that there is an implicit assumption that the bits in the History variable will correspond to those in the Present faults variable (in this case, user_bit5 and p_user_bit5). Finally, notice that the *Alt_Address* is persistent while the present faults address is not. This is the proper way to specify these variables. The fault history will be restored when the power is again turned on, while the present faults will be automatically cleared.

5.7 UPDATING YOUR PARAMETER DEFINITIONS WITH WINVCL

To update your parameter definitions, use WinVCL; see the *WinVCL User's Guide*. The path to this function is Management Display → Project button → Edit this project → Add or Delete Parameter Files.

6 SUGGESTIONS ON HOW TO STRUCTURE PROGRAMS

In this section, we describe our prejudices about how to write programs that are readable and maintainable. These prejudices are born out of the knowledge that a large portion of software effort is spent in maintaining programs.

6.1 GENERAL PROGRAM STRUCTURE

As you initially write a program, it is very easy to forget that six months from now someone else may have to find the program, understand it, and then modify it to suit changing requirements. Ideally, you want to minimize the time spent looking for information, in order to maximize productivity. Consistency in formatting program code is the key.

We have found that the following organization consistently reduces the amount of time it takes to find the piece of information you are looking for in any given program. Programs are formatted into the following sections:

- An introductory block that describes the program
- A declaration section where all constants and variables are declared
- A one-time initialization section where the startup code is placed
- The main body of the code
- Supporting subroutines.

6.1.1 Introductory Block

The introductory block should clearly identify the program and include a brief description of what the program is supposed to do. It is also a good idea to describe the system resources required by this program. For example, what I/O is used? We like to see: a one-line program title within a banner, the copyright, the current version, a description of what the program does, perhaps the VCL compiler version, and, if appropriate (and it almost always is), a description of the I/O used.

6.1.2 Declaration Section

The declaration section should start with a banner to make it easy to find. In general, it is better to group constants and variables separately (we like to put constants first). Also, although VCL is case insensitive, following the C convention of putting constants in upper case and variables in lower case will save you time by making your programs more readable.

6.1.3 One-Time Initialization Section

The initialization section should appear immediately after the declaration section and be set off with a banner. If there is a lot of code in this section, it may be a good idea to use subroutines to make your intentions more clear.

6.1.4 Main Code and Subroutines

The main body of the code should come immediately after the initialization section and also be set off by a banner and followed by any subroutines.

6.2 AN EXAMPLE PROGRAM – "HELLO" REVISITED

```
;=========================================================================
; Hello.exe - Print Hello World on the Spyglass
;=========================================
;
; Copyright © 2004, all rights reserved
; XYZ Corporation
; Anytown, USA
;
; Version 1.1
;
;
; Description
;------------
; This program demonstrates how to display text on the Curtis
; Spyglass display. It also demonstrates the use of the time
; delay function to display multiple lines of text.
;
;
; I/O Requirements
;-----------------
; This program uses only the serial port.
;
;
;=========================================================================
; Declaration Section
;====================
; none
;
;
;=========================================================================
; One Time Initialization
;========================
; none
;
;
;=========================================================================
; Main :: Main Program Loop
;==========================
;
;
while (1=1)
    {
    put_spy_text(" Hello ")

    setup_delay(DLY1,500)
    while (DLY1_output <> 0) {}

    put_spy_text(" World ")

    setup_delay(DLY1,500)
    while (DLY1_output <> 0) {}
    }
```

Notice that the major sections are set off by banners. This visual aid helps you find your way around quickly. This can save a lot of time in larger programs.

In this example, we do not have any declarations, nor do we have any one-time initialization; however, these sections are so common that we like to see them declared, even if they are blank. Notice that we have put the word *none* to make

sure that the reader realizes that if there were any information to go into this section, it would be there, but that in this case the sections really are blank.

On the other hand, we don't have any subroutines and we don't create any blank sections for them. The idea is to provide a consistent look and feel to your programs by always filling in the most important parts, but not to "make work" by demanding that rarely used sections are always populated.

Although we haven't shown it here, it is often convenient to keep a table of contents at the beginning of the file. This is especially true if there are a large number of subroutines. One strategy consists of putting a distinctive string in the routine's banner and then using a search program to retrieve all such strings. For example, say we have a subroutine called my_routine. The banner for this routine would look like this:

```
;------------------------------------------------------------------------
; my_routine :: This is a sample routine
;---------------------------------------
; -Some comments about what my routine does
;
my_routine:
```

Using a distinctive string, preceded by a pair of colons (::), allows us to search for these strings (we use grep) so that we can easily build a table of contents.

Finally, notice that the code is formatted consistently. The part of the code that is controlled by a conditional statement is surrounded by braces, which are at the same level of indentation as the code. There are many ways to format your code. You should pick one style and then apply it consistently—it will make maintenance easier.

6.3 USE OF SUBROUTINES AND MODULES

It is usually the case that a large program will be more easily maintainable if you judiciously use subroutines. The amount of time it takes to execute the subroutine (the call and return) is usually trivial, compared to the benefits.

The argument for using modules is the same as for using subroutines. The time it takes to execute a section of code in a module is trivial, while the benefit of identifying a block of common code can be significant when maintaining the code. In fact, a module is even more effective in helping to simplify your code in that it allows you to group a number of submodules under a common, unifying concept (the module). One effective way to implement large modules is to put the code into a separate file. This file is then included with the main program, either by using an explicit include-statement or by adding the modules to the source list in WinVCL (see the *WinVCL User's Guide*). This makes the boundaries of the module more apparent.

Another reason to use subroutines and modules is that they enhance your ability to schedule low-priority tasks. For example, say you need to monitor an emergency stop button and to run a display. The display only needs to be

updated every 0.1 seconds. On the other hand, you may need to respond to the emergency stop button within 0.01 second. Rather than update the display each time through your main loop, consider putting a counter in so the display is serviced every tenth time through the loop. Always conserve system bandwidth; see Section 7 of the *WinVCL User's Guide*, "Program Timing".

6.4 DESIGNING PROGRAMS TO MINIMIZE ERRORS

With very small programs (the kind you wrote as you were learning how to program), you can sometimes get away without a formal design; however, as the size of the program increases, so does the importance of a proper design. The time spent doing a proper design will be more than recovered in the time saved over the lifetime of the project.

From your previous programming experience, you probably have experience with the standard design strategies. The suggestions that follow should be viewed as reminders to do the right thing: understand the requirements, create a solution, and partition the solution appropriately to the target language.

A proper design begins with understanding the requirements of the finished program. If you don't know what the finished product is supposed to do, it is very unlikely that you will be able to write a program to do it. There are two major problems in gathering the requirements for a program, both of which you have probably experienced.

First, it is often difficult to elicit the requirements from the customer. Most of the time, the people who will use the program have little or no idea of how to program, and you will have little or no idea of the particular needs of their application.

The solution to this problem is to establish a common vocabulary. There are a number of techniques for establishing a common way to talk about requirements. The Universal Modeling Language's (UML) "use cases" are very popular. Whatever technique you choose should allow you to verify your understanding of the program requirements with the customer and also allow you to create tests to make sure that the final program does indeed meet its requirements.

The other difficulty that we often face is that the customer may not be sure of what the ultimate program should do. This isn't so much a reflection of the user's intelligence as it is a predictable reaction to the introduction of new technologies. It will often be the case that one or more subsystems will be described mostly by hand waving.

The solution to this problem is to (a) acknowledge and document that you don't know enough, (b) separate the questionable subsystems from the rest of the code as best you can and then (c) put the questionable subsystem code into 'stub' routines. (A stub is a subroutine with no code in its body.) One of the benefits of VCL is that it is usually easy to put a rough demonstration together. For example,

suppose that your customer is building an electric vehicle and can't decide how the steering should respond. What do you do? Put the steering control into its own subroutine. As a first approximation, provide a simple linear response. Then, as the customer gets a feel for the vehicle, you could then change to a hyperbolic, exponential, or progressive response, to name but a few of your options.

Once you understand the requirements, you can create a solution. You may use control flow diagrams, data flow diagrams, the UML, or any number of design techniques to help yourself to understand how to solve the problems posed by the requirements. Once again, we assume that you have at least a working familiarity with one or more of these techniques. In general, what we are calling the solution will record all of the information necessary to create a program without prejudice about the implementation language.

The final phase of the programming process is recasting the solution in terms of the implementation language. Each language has its strengths and weaknesses. Understanding these is essential to writing programs that perform to their requirements and that are maintainable.

The process of translating a design into a given language is generally a process of divide and conquer, making the divisions appropriate to the target language. One of VCL's strengths is its ability to build signal chains (see Section 4); you will want to identify these sections of code first. Next, as in any language, you will want to create subroutines by looking for common sections of code and sections of code that should be grouped to make your code easier to understand. As you create these subroutines, you will do well to keep in mind how you will test each of the pieces.

The remaining code falls into two general categories: initialization and glue. The initialization code is run once when the system starts up (or when it is restarted) to set the initial value of variables and setup signal chains. The glue code gets its name from the fact that it is used to bind the subroutines (and any external controls for the signal chains) together. Generally, the glue code is run in a main processing loop that runs continuously.

Finally, always keep in mind that you are writing programs that will run in real-time. You may find that you need to schedule certain low priority tasks so that there is enough time to do the critical tasks. Keep in mind that VCL has a number of timers you can use to do this, and also keep in mind that the compiler can help by providing instruction timing information; see Section 7 of the *WinVCL User's Guide*, "Program Timing".

7 DEBUGGING PROGRAMS

This section describes techniques for finding defects in programs, because almost inevitably there will be some. It is important to remember that the most effective technique for minimizing errors is a good up-front design (see Section 6.4).

7.1 FINDING COMPILER ERRORS

The compiler checks to make sure that your program is syntactically correct. When the compiler finds an error, it emits a message in the form of:

```
Error around line nnn: error-message
```

Where *nnn* is the line number and *error-message* is a short description of the problem. The method employed in the translation leaves the actual line number somewhat ambiguous, hence the "around line". This ambiguity will be resolved by looking at the message. For example, assume that, on line 354 of your program, you had the following statement:

```
setup_dlay(DLY1, 150)
```

It is sometimes happens that our mind moves faster than our fingers. In this case, we meant to type *setup_delay*, but the *e* in delay was skipped. The compiler will detect this and issue the following message:

```
Error around line 354: setup_dlay is not a function
```

Once again, keep in mind that just because the compiler runs to completion with no errors only means that the syntax of your program is correct. It does not mean that the program runs correctly.

7.2 FINDING RUN-TIME ERRORS

To verify the functionality of your program, you will have to test it. It is at this point in the development process that you will reap the benefit of a good design. If you have decomposed your program properly, you will find it easy to verify your subroutines. If you have captured the requirements adequately, you will be able to easily generate tests for them (in effect, testing the logic of your glue).

There are actually two different classes of run-time errors: range errors and semantic errors.

Range errors occur as a result of passing a parameter to a predefined function that is either too large or too small. These errors are automatically detected by the system at run-time (see SysInfo for a description of the error codes generated by the product you are using). Because of this, it is important to have good test coverage.

Semantic errors occur as a result of your program not performing according to the requirements. It is good to keep in mind that there are actually two sets of requirements: externally visible and internal.

The externally visible requirements you developed in collaboration with the customer. This set of requirements should have been recorded without any information about how they were to be implemented. For example, you might say that the maximum speed of a vehicle is to be no more than eight miles an hour. This is stated without respect as to how this will be accomplished. You could limit the speed using a speedometer in the feedback loop, or you could use a mechanical governor. The external requirements don't say how, only what.

The internal requirements will have been developed by you as you re-cast the program design into a program. Using the previous example, you might have specified a solid-state speed sensor. In this case, your tests may include checking if the sensor's output is within range, if the output of the sensor is getting to the code that uses it, and, perhaps, if the sensor is being read in a timely manner. Once again, it is best to consider all such issues as you are designing the code.

All VCL enabled systems have a serial I/O connection. The traditional technique for testing systems is to insert print-statements in the code so that you can then verify the values at run-time. Spyglass statements are the VCL equivalent of print-statements. Using Spyglass statements you could, for example, output the value read from your speed sensor to a Spyglass display and compare it to a known good speedometer.

Within the WinVCL environment, there is a much more efficient debugging technique: you can monitor variables directly, in real-time. See Section 4.1, "Accessing parameter values", in the *WinVCL User's Guide*.

APPENDIX A
VCL KEYWORDS

Like most programming languages, VCL has a set of reserved words and symbols. These keywords have intrinsic definitions that cannot be changed.

Because of the way VCL was developed, some of the keywords have synonyms, which appear in parentheses in the following list. The initial keyword is preferred, although the synonyms will also work.

```
{   (begin, then)
}   (end, endif)
=   (==, eq)
<>   (!=, ne)
<   (lt)
<=   (le)
>   (gt)
>=   (ge)
&   (&&, and)
|   (||, or)
^   (xor)
~   (not)
%   (mod)
>>   (sr)
<<   (sl)
@
```

```
begin_module
bit
call
constant  (const)
else
end_module
enter
equals  (alias)
exit
false
function
goto
if
include
off
on
return  (ret)
string
true
uses
variable  (var)
while
```

APPENDIX B
PRECEDENCE RELATIONSHIPS

The order in which expressions are evaluated depends on their operator's precedence and the order in which the operators appear.

Precedence Table

Level 1 (highest)	`()`	parentheses
Level 2	`-`	unary minus
	`~`	bit-wise negation
Level 3	`*`	multiplication
	`/`	division
Level 4	`+`	addition
	`-`	subtraction
Level 5	`<<`	shift left *n* bits
	`>>`	shift right *n* bits
Level 6	`&`	bit-wise AND
	`|`	bit-wise OR
	`^`	bit-wise exclusive OR
Level 7	`=`	equal to
	`<>`	not equal to
	`>`	greater than
	`>=`	greater than or equal to
	`<`	less than
	`<=`	less than or equal to
Level 8	`&`	logical AND
	`|`	logical OR
Level 9 (lowest)	`=`	assignment

So, the expression:

```
a + b * c
```

will group as (a + (b * c)). The precedence of operators at the same level is left associative. This means that the expression a+b+c will group as ((a+b)+c). Of course, you can explicitly set the order of precedence by using parentheses.

APPENDIX C
PARAMETER DEFINITION KEYWORDS

These keywords can can be used in the comment fields of the source files defining your programmable parameters.

NAME OF TAG	MANDATORY	ARGUMENT	DEFAULT
address	—	“Address-String”	—
alt_address	—	“Address-String”	—
BitActiveLow	—	YES, NO	NO
BitSelect	—	0–7	—
default	—	Integer or On/Off (for bit values)	—
end	Yes	—	—
DecimalPos	—	0, 1, 2, 3	0
level	—	1-6	1
LAL_read	—	1-7	—
LAL_write	—	1-7	—
MaxDsp	—	Integer	—
MaxRaw	—	Integer	—
MinDsp	—	Integer	—
MinRaw	—	Integer	—
parameter_entry	Yes	“Name-String”	—
signed	—	YES, NO	NO
step_size	—	0–15	—
type	Yes	Program, Monitor, Faults	—
units	—	“Units-String”	—
width	—	8bit, 16bit, 32bit	—

Mandatory items must appear in every definition. The remaining entries may or may not be necessary, depending on the *type* field. For example, a Faults entry must have both an *address* and an *alt_address* entry. A Program or Monitor entry need only have an *address* entry.

EXTENDED DESCRIPTION OF EMBEDDED KEYWORDS

A brief description of each of the keywords that can appear within a source file is provided. In the descriptions, the keywords are followed by an argument type. You will see the following types:

Label: A label that comes from the map file (see the tools description).

String: A sequence of characters (not including a new line) enclosed in double quotes.

Integer: An integer in one of the following formats: 123 (decimal), 00101b (binary), 0x1AD (hex) or 1Adh (hex).

{word, word, … ,word}: Choose one (and only one) word from the list enclosed by the braces.

Address *Address-String*

The Address keyword is followed by the symbolic name of the address of the parameter. This will ultimately be translated into the absolute address within the device. In the case of Faults entries, the symbolic name should be the address of the present fault flag.

Alt_Address *Address-String*

The Alt_Address keyword is followed by the symbolic name of the address of the parameter. This will ultimately be translated into the absolute address within the device. In the case of Faults entries, the symbolic name should be the address of the fault history flag.

BitActiveLow *{yes/no}*

The assumption is that bits are always active-high. This keyword allows you to specify that the bit is actually active-low. This affects the bit-on and bit-off messages. Take, for example, a bit that produces an On/Off message. Normally, the On message will be displayed when the bit is high. If the BitActiveLow flag is set to *yes*, the Off message will be displayed when the bit is high.

BitSelect *{0…7}*

This keyword allows you to isolate a bit. You may only specify one bit in a given definition. Parameters with the BitSelect keyword must have a width of 8 bits.

Default *Integer* or *{on/off}*

This keyword allows you to specify the default value of a parameter. The value can be any integer within the range of *width*. If you are specifying bit values, you may use On (1) and Off (0) to specify the value of the bit parameter. Be aware that bit values reflect the value of BitActiveLow; if BitActiveLow is set and the default value is Off (0), the bit will be set.

DecimalPos *{0…3}*

You can adjust the position of the decimal point using this keyword. It effectively shifts the decimal point to the left. If your parameter's value is 1234 and you have a DecimalPos of 2, the display will be 12.34.

End

This must be the last keyword for an entry.

Level *{1...6}*

This keyword is used to set the indentation level for menu entries. Valid indentation levels range from 1 to 6. This keyword must be entered when defining a menu title line. It must not be defined for any other type of entry.

LAL_Read *{1…7}*

Lowest **A**ccess **L**evel with Read privileges. This is the lowest level that still has Read privileges (7=designer only, 1=user).

LAL_Write *{1…7}*

Lowest **A**ccess **L**evel with Write privileges. This is the lowest level that still has Write privileges (7=designer only, 1=user).

MaxDsp *Integer*

This keyword allows you to set the maximum value that is displayed. When the actual value of the variable is equal to MaxRaw, the value displayed by the 1311 will be equal to the value set by MaxDsp.

MaxRaw *Integer*

This keyword allows you to establish the maximum value for a given variable. The 1311 may display a value that is greater than MaxRaw, but you cannot use the 1311 to enter a value that is greater than MaxRaw.

MinDsp *Integer*

This keyword allows you to set the minimum value that is displayed. When the parameter's actual value is equal to MinRaw, the displayed value will be equal to the value specified by MinDsp.

MinRaw *Integer*

This keyword allows you to establish the minimum value for a given variable. The 1311 may display a smaller value than MinRaw, but you cannot use the 1311 to enter a value that is smaller than MinRaw.

Parameter_Entry *Name-String*

This keyword is used to start a parameter entry. It is followed by the name of the entry. The following rules apply:

An up-arrow (^) is a special character that causes the 1311 to force the next character (and only the next character) to uppercase (e.g., ^p^w^m results in PWM).

Enter the parameter name within double quotes (e.g., Parameter_ Entry "^My ^Parameter").

A backslash is the escape character. Whatever character follows it will be entered (e.g., "\"" is used to enter a quotation mark within a string). To enter a backslash into a string, add a second backslash after the escape backslash (i.e., "\\" results in a single backslash in the string).

A vertical bar (|) is a special character that causes the 1311 to backspace once for each vertical bar (e.g., to make *weaken* into *weak*, you could enter "weaken||"). This technique can be used to save bytes by modifying words that are in the stored dictionary rather than creating new ones.

Signed *{yes/no}*

This keyword determines whether the parameter is signed. The default is unsigned (0–65535). Setting Signed to Yes will make the parameter signed (-32768 to +32767).

Step_Size *{0…15}*

This field determines the step size when the display value is incremented or decremented. A value of 0 implies no increment or decrement. A value of 1 means that we will increment or decrement by one least significant digit of the display value as determined by the Exponent field. If the exponent field is 1 (i.e., the decimal place moves one place to the left), then our increment/decrement step size will be 0.1. This strategy holds up through step sizes of 10. Step sizes 11 through 15 are reserved for special step sizes as follows:

11 = 25

12 = 50

13 = 100

14 reserved

15 reserved

Note: It is the responsibility of the developer to ensure that the step size is great enough to produce a change of the raw value. Otherwise the user may see a change of the display value although the actual parameter value in the controller has not changed.

Type *{Program/Monitor/Faults}*

This keyword determines the major menu heading under which this entry will be created.

Program entries are values that can be altered using the handheld programmer

Monitor entries are values that can be monitored in real-time

Faults entries display either the current fault(s) or a log of the last *n* faults

Units *Units-String*

This keyword allows you to define the units (e.g., volts) for this entry.

Width *{8bit/16bit/32bit}*

This keyword is used to establish the number of bits occupied by the variable being defined. If the field is *other*, this is not a normal parameter and a display method will have to be specified (e.g., date and time entries will use a width of *other*).

APPENDIX D

STANDARD PARAMETER UNITS

The 1311 has a standard set of units. By selecting from the predefined set of units, you can decrease the amount of space taken up by your parameter block. Some of the units have shorter forms, which are shown here in parentheses. When space is limited the shorter version will be used.

- amp (A)
- close
- ccw
- cm
- cw
- oC
- disable
- down
- enable
- false
- forward
- g
- hour (Hr)
- Hz
- in
- joule (J)
- kg
- kHz
- kJ
- km
- kOhm (ko)
- kph
- kW
- l/s
- lift
- liter (l)
- lower
- mA
- meter (m)
- mH
- minute
- mJ
- ml
- mm
- mOhm (mo)
- ms
- mV
- us
- negative
- Nm
- no
- off
- ohm (o)
- on
- open
- out
- %
- positive
- reverse
- rpm
- second (s)
- true
- up
- volt (V)
- watt (W)
- yes

APPENDIX E
SYSINFO STRUCTURE

This section shows you how to find your way around a SysInfo (System Information) file.

E.1 TITLE BLOCK

The SysInfo file begins with a title block, listing Curtis's copyright and contact information.

E.2 SECTION HEADERS

The information after the title block is organized by section. A line running across the page precedes the start of each section. Directly below the line is a title describing the section's contents. Finally, there may be one or more comment lines underneath the title line, each beginning with a hyphen (-). Here is an example:

ADC - Analog to Digital Conversion Services

-These constants, variables, and routines allow access to the analog inputs

This announces that we are starting the ADC Function Group section and tells you what you should expect to find in this section.

E.3 SECTION LAYOUT

Typically sections are divided into subsections. For example, each Function Group section is divided into three subsections: constants, variables, and functions. You may find a few exceptions to these rules depending on when the SysInfo file was built, but in general these rules are followed.

Each subsection starts with a header describing the nature of the elements that follow. This may be followed by one or more comments (each preceded with a bullet •), and then the definitions.

Analog Channel Access IDs

• These IDs are used with the get_adc() function to specify a particular input channel.

```
Constant  0 (0x0000) ADC1     ADC Channel 1  Common Pot High
Constant  1 (0x0001) ADC2      "     "    2  Pot 1 wiper input
Constant  2 (0x0002) ADC3      "     "    3  Pot 2 wiper input
```

The subsection header tells you these are ID constants for analog functions. The comment below tells you these constants are used with the get_adc() function. In this example, we have shown only the first three definitions.

E.4 FORMATS FOR DATA DEFINITIONS

Constant, variable, and bit definitions each fit on one line. Each begins with the type of definition (constant, variable, or bit). Next comes the name; there is often a comment at the end of the line.

Constant definitions always have a value between the type and the name:

```
Constant 32768 (0x8000) USEHB     Access a variable's high byte
```

This defines a constant whose value is 32,768 (or hex 0x8000), is named USEHB, and which is used to access the high byte of a value in the variable table.

Variable definitions follow the standard format:

```
Variable CAN_bus_on     CAN Bus Status (0=off, 1=on)
```

This is a variable whose name is CAN_bus_on, which holds the information about the CAN bus status.

Bit definitions reference a specific bit or group of bits with a given variable. Consequently, bits are always defined in conjunction with a variable. To make this relationship clear, bit definitions always appear under the variable that they reference and are indented slightly to show their dependence on the variable. Here is an example of bit definitions, with their controlling variable:

```
Variable CAN_response_error   Message response timeout flags
    Bit CAN1_timeout          Timeout bit def. for CAN1
    Bit CAN2_timeout             "     "   "    "  CAN2
    Bit CAN3_timeout             "     "   "    "  CAN3
    Bit CAN4_timeout             "     "   "    "  CAN4
      etc.
```

Because the bit variables are immediately below and slightly indented from the variable CAN_response_error, we know they are used to reference bits within CAN_response_error.

E.5 FORMAT OF A FUNCTION DEFINITION

The first line tells you what the function name is and how many parameters it expects, along with a short description of what the function does. Directly below this you may find an extended description of the function. In the following example the function setup_can(), which requires 5 parameters, is defined.

SETUP_CAN(5) – Setup the CAN system and leave it in an idle state

- This function is used to reset or reconfigure the CAN system.

Parameters

```
Baud      Baud rate constant (CAN_125KBAUD, CAN_250KBAUD, CAN_500KBAUD)
Sync      Sync period (0=off, n=number of 4ms ticks between syncs)
Reserved  (set to 0)
Reserved  (set to 0)
Restart   CAN bus auto-restart after bus Off (0=yes, !0=no)
```

Returns

```
0           CAN system not setup.
1           CAN system successfully setup.
```

Errors

```
BAD_BAUD    Invalid baud rate constant.
```

The Parameters, Returns, and Errors subsections appear in every function definition.

The *Parameters* section gives a name for each parameter and a description of how the parameter is to be used. In this example, we find that the first parameter is used to set the baud rate.

All VCL functions return a value. The *Returns* section tells you how to interpret the return code. In this case, if the value is 1, the function completed successfully; otherwise, it returns 0.

Finally, the *Errors* section tells you about any system errors that may be generated when you call this function. In this case, if you give the function an undefined baud rate you will generate a system error: BAD_BAUD; see Section E-6.

E.6 ERROR AND STATUS INFORMATION

This section lists the generic error and status information common to all VCL systems (Error Identification, Error Module IDs, and Error Codes), and then the product-specific error and status information. Here are examples of the three types of information.

Error Identification:

```
Variable last_general_error          Last reported (system) error
```

Error Module ID:

```
04 Ramp processing
```

Error Codes:

```
03 (AUTO_RUN)          Attempt to access element running automatically
```

Error codes are grouped as follows:

01–09	global errors
10–29	VCL-related errors
30–39	CAN-system-related errors
40–49	other common errors (limit block, EEPROM, and parameter block errors)
50+	product-specific errors

E.7 CAN OBJECT DICTIONARY

This section lists all the CAN Object IDs and their associated variables. The subindex is separated from the object index by a period, as shown in this example:

```
0x1A00.05      can_pdo_miso_1_map_5
```

For convenience, this list is in alphabetical order by variable name.