
MANUAL

VEHICLE CONTROL LANGUAGE

Common Functions

© 2005 CURTIS INSTRUMENTS, INC.

VCL Common Functions, p/n 37314
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

1.0 OVERVIEW	1
1.1 Application (how the functions are used)	1
1.2 Data: Constants and Variables	1
1.3 Functions	2
A • MATH FUNCTIONS	5
2.0 ABS — ABSOLUTE VALUE	6
2.1 Using the Absolute Value Functions	6
2.2 ABS Data	6
2.3 ABS Functions	6
automate_ABS()	
get_ABS()	
3.0 MTD — MULTIPLY THEN DIVIDE	7
3.1 Using MTD	7
3.1 MTD Data	7
3.2 MTD Functions	7
automate_muldiv()	
get_muldiv()	
4.0 RND — RANDOM NUMBER GENERATOR	9
4.1 Using the Random Number Generator	9
4.2 RND Data	9
4.3 RND Functions	9
random()	
random_seed()	
5.0 SCL — SCALING	10
5.1 Using the SCL Functions	10
5.2 SCL Data	10
5.3 SCL Functions	10
automate_scale()	
scale_value()	
setup_scale_factor()	
B • SIGNAL PROCESSING FUNCTIONS	12
6.0 CPY — COPY ONE VARIABLE TO ANOTHER	12
6.1 CPY Data	12
6.2 CPY Data	12
6.3 CPY Functions	12
automate_copy()	
automate_block_copy()	
automate_limited_block_copy()	

7.0	FLT — FILTERS	15
7.1	Using Filters	15
7.2	FLT Data	15
7.3	FLT Function.....	15
	<code>automate_filter()</code>	
8.0	LIM — LIMITS	16
8.1	Using Limits	16
8.2	LIM Data.....	16
8.3	LIM Functions.....	16
	<code>automate_limit()</code>	
	<code>get_limit()</code>	
	<code>set_limit()</code>	
	<code>setup_limit()</code>	
9.0	MAP — TWO-DIMENSIONAL MAPS	18
9.1	Using Maps.....	18
9.2	MAP Data.....	20
9.3	MAP Functions.....	20
	<code>automate_map()</code>	
	<code>get_map_output()</code>	
	<code>get_map_segment()</code>	
	<code>setup_map()</code>	
10.0	RMP — RAMPING	22
10.1	Using Ramps.....	22
10.2	RMP Data.....	24
10.3	RMP Functions.....	25
	<code>automate_ramp()</code>	
	<code>set_ramp_hold()</code>	
	<code>set_ramp_rate()</code>	
	<code>set_ramp_target()</code>	
	<code>setup_ramp()</code>	
11.0	SEL — SELECTOR SWITCH	27
11.1	Using Selector Switches	27
11.2	SEL Data	27
11.3	SEL Functions.....	28
	<code>automate_select()</code>	
	<code>set_select()</code>	
	<code>setup_select()</code>	
	<code>automate_4p_select()</code>	
	<code>set_4p_select()</code>	
	<code>setup_4p_select()</code>	

C • TIMING FUNCTIONS	31
12.0 DLY — TIME DELAYS	32
12.1 Using Delays	32
12.2 DLY Data	32
12.3 DLY Functions	32
setup_delay()	
setup_delay_prescale()	
13.0 TMR — TIMERS (HOURMETERS)	34
13.1 Using Timers	34
13.2 TMR Data	34
13.3 TMR Functions	35
disable_timer()	
enable_timer()	
reset_timer()	
setup_timer()	
D • CONTROL FUNCTIONS	36
14.0 PID — PROPORTIONAL, INTEGRAL, DERIVATIVE	37
14.1 Using PID Functions	37
14.2 PID Data	37
14.3 PID Functions	38
automate_PID()	
reset_PID()	
E • MEMORY ORGANIZATION FUNCTIONS	39
15.0 NVM — NONVOLATILE MEMORY	40
15.1 Using Nonvolatile Memory	40
15.2 NVM Data	44
15.3 NVM Functions	45
NVM_block_format()	
NVM_block_read()	
NVM_block_write()	
NVM_nvuser_restore()	
set_NVM_save_rate()	
NVM_write_parameter()	
16.0 SYS — SYSTEM-LEVEL GROUP	48
16.1 Using the System-Level Functions	48
16.2 SYS Data	48
16.3 SYS Functions	49
clear_diaghist()	
get_fault_code()	
reset_controller()	
set_sys_save_rate()	

17.0 VCL INTERPRETER	51
17.1 Using the VCL Functions.....	51
17.2 VCL Data	51
17.3 VCL Functions	52
VCL_get_size()	
VCL_get_byte()	
VCL_put_byte()	
F • COMMUNICATIONS FUNCTIONS	53
18.0 SER — SERIAL PORT CONTROL	54
18.1 Using the Serial Port Function	54
18.2 SER Data	54
18.3 SER Function.....	54
setup_serial()	
19.0 SPY — SPYGLASS SUPPORT	55
19.1 Using the Spyglass Functions.....	55
19.2 SPY Data	55
19.3 SPY Functions.....	56
put_spy_LED()	
put_spy_message()	
put_spy_text_offset()	
put_spy_timer()	
setup_spy_protocol()	
19.4 Superseded SPY Functions	58
put_spy_bin()	
put_spy_dec()	
put_spy_hex()	
put_spy_mixed()	
put_spy_text()	
G • CONTROLLER AREA NETWORK (CAN) FUNCTIONS	61
20.0 CANopen	62
20.1 Using CANopen	62
20.2 CANopen Data	62
20.3 VCL Functions Applicable to CANopen	66
enable/disable_CANopen()	
enable/disable_CANopen_emergency()	
enable/disable_CANopen_heartbeat()	
enable/disable_CANopen_nmt()	
enable/disable_CANopen_pdo()	
enable/disable_CANopen_sdo()	

21.0 VCL CAN	70	=====
21.1 Using VCL CAN	70	=====
21.2 VCL CAN Data	79	=====
21.3 VCL CAN Functions	83	=====
CAN_last_error()		=====
CAN_set_cyclic_rate()		=====
enable/disable_mailbox()		=====
send_mailbox()		=====
setup_CAN()		=====
setup_CAN_resync_width()		=====
setup_CAN_sample_delay()		=====
setup_CANRO_mask()		=====
setup_mailbox()		=====
setup_mailbox_byte_select()		=====
setup_mailbox_data()		=====
shutdown_CAN()		=====
shutdown_CAN_cyclic()		=====
startup_CAN()		=====
startup_CAN_cyclic()		=====
21.0 COMBINING CANOPEN & VCL CAN	90	=====

1

OVERVIEW

VCL (Curtis’s Vehicle Control Language) functions are the VCL equivalent of a program library. The functions described in this manual are the common (shared) functions, available on all VCL-enabled systems. We assume you are already familiar with the VCL programming language from the *WinVCL User’s Guide* and the *VCL Programmer’s Guide*.

The VCL functions are described by group. A function group description includes the definition of each of its constants, variables, and functions. Each group has a three-letter designation—for example, ABS—that is used to identify it and (with a few exceptions) to provide a base name for its constants and variables.

Each of the nineteen function groups has its own section in the manual. Each of these sections contains three subsections: application, data, and functions.

1.1 APPLICATION (HOW THE FUNCTIONS ARE USED)

Each function group description starts with an explanation of how the functions would be used. In some cases this is an extended “how-to” scenario; in others it is a brief summary.


1.2 DATA: CONSTANTS AND VARIABLES

The data subsection lists and defines the function group’s constants and then its variables.

Constants

Most function groups include a set of block ID constants (e.g., ABS#), one for each instance of the data block. The description specifies the number of data blocks that are typically available (for the ABS group, there are two). However, the number of instances may vary, depending on controller model. You can confirm the number available for your controller by looking in its SysInfo file.

Although VCL is not case sensitive, we use the convention of writing constants in all caps.

-  Use a constant’s written-out form, and not the corresponding number that appears beside it in the SysInfo file; these numbers are subject to change.

Variables

Although VCL is not case sensitive, we use the convention of writing variables in lowercase. Note, however, that function group identifiers (e.g., ABS) and several common acronyms are often capitalized in variable names.

1.3 FUNCTIONS

The functions subsection describes each of the functions in the group. The functions are listed alphabetically, for easier reference.

Setup and automate functions appear within several function groups. The important thing to note about setup functions is that they should be the first functions to be executed within a function group. Because functions are listed alphabetically, however, the setup function will not appear first. Automate functions cause the indicated instance of the block to begin executing automatically. You should place any automate functions in your code's one-time initialization section.

Format of a function

The number in parentheses at the end of a function's name indicates how many parameters must be entered between the parentheses. For example, the function

automate_ABS(2)

requires two parameters to be entered: an ABS data block ID and the address of a source variable.

Parameters are separated by commas within the parentheses, and must be entered in the order they are listed.

In your code, the automate_ABS() function might look like this:

```
automate_ABS(ABS1, @ABS1_input).
```

Format of a function description

For each function, a brief description is presented first, followed by a listing of its parameters, returns, and reported errors.

Parameters

Function parameters take either static values or dynamic values. To specify a static value, you type in the desired value (which might be a numerical value, a named constant, or a variable). If the parameter takes a dynamic value, it will be preceded by an "at" sign (@), signifying the *address* of a variable; in other words, what you type in is a *data pointer*. Each time the function runs, VCL fetches the present value of these dynamic variables.

Returns

All the returns that can be generated by the function are listed. Almost all functions return simply a confirmation:

1 = function executed successfully
0 = function did not execute.

Any other returns are listed and described.

To get a return, simply set a user variable (typically user1 is used as the “scratch” variable) equal to the function. Thus

```
user1 = automate_ABS (ABS1, @ABS1_input)
```

will return either a “1” or a “0”, depending on whether the function executed successfully.

Reported Errors

The errors listed here are reported in the variables last_VCL_error_module and last_VCL_error (see SysInfo file for error module IDs and error codes). VCL stops running when any of these errors occurs. These are the most common types of reported errors:

AUTO_RUN	Attempt to access a parameter that’s running automatically.
BAD_ID	Block ID out of range.
PARAM_RANGE	Parameter out of range.
PARAM_RO	Attempt to write to a read-only parameter.
PT_RANGE	Unknown parameter.

These additional reported errors are function-group specific:

BAD_BAUD	Invalid baud rate constant (<i>CAN group</i>).
BAD_MO_ID	Block ID not unique (<i>CAN group</i>).
MO_INACTIVE	Mailbox is inactive (<i>CAN group</i>).
MO_SENDING	Attempt to send mailbox before it was sent the previous time (<i>CAN group</i>).
TYPE_SETUP	Unknown parameter (<i>LIM group</i>).
UNIT_UNDEF	Attempt to access a map before it is setup (<i>MAP group</i>).

ORGANIZATION

Function groups are presented in this manual within seven categories:

A. Math	5
ABS Get the absolute value of a variable.	
MTD Multiply then divide.	
RND Random number generator.	
SCL Scale numbers per a specified scaling factor.	
B. Signal Processing	12
CPY Copy a variable from one place to another.	
FLT Filter incoming signals.	
LIM Limit a signal.	
MAP Map one variable to another.	
RMP Setup variables that vary linearly with time.	
SEL Select the output of a software switch.	
C. Timing	31
DLY Setup delays relative to the system clock.	
TMR Setup timers to record elapsed time.	
D. Control	36
PID Time-dependent system control (proportional, integral, derivative).	
E. Memory Organization	39
NVM Nonvolatile memory.	
SYS System-level data management at startup and shutdown.	
VCL Access individual bytes within variables.	
F. Communications	53
SER Configure the serial port.	
SPY Setup and display messages on the Spyglass.	
G. Controller Area Network (CAN)	61
CANopen Ready-made CAN interface.	
VCL CAN VCL's do-it-yourself CAN interface.	

• A •

MATH FUNCTIONS

All the algebraic operations performed in VCL use integer math and thus have truncation and rounding errors. In order to enhance the basic math operations (+, -, *, /), VCL includes functions to aid computation and increase accuracy.

2. The Absolute Value (**ABS**) functions allow you to obtain the absolute value of a variable, either under program control or automatically.6
3. The Multiply then Divide (**MTD**) functions perform a full 16-bit multiplication followed by a division, either under program control or automatically.7
4. The Random Number Generator (**RND**) function returns a 15-bit signed pseudo-random number; you can set a customized seed value if desired.9
5. The Scaling (**SCL**) functions allow you to scale numbers — often necessary in automatically processed signal chains.10

A•2

ABS
ABSOLUTE VALUE

2.1 USING THE ABSOLUTE VALUE FUNCTIONS

The ABS function group allows you to take the absolute value of a variable.

2.2 ABS DATA

CONSTANT GROUP ⇒

ABS# *Block ID*

These constants identify the two ABS data blocks: ABS1 and ABS2.

VARIABLE GROUP ⇒

ABS#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of ABS1 is placed in the variable ABS1_output.

2.3 ABS FUNCTIONS

FUNCTION ⇒

automate_ABS(2) *Run the absolute value function automatically*

This function causes the indicated instance of the ABS block to begin executing automatically, using the value indicated by *Source* as its input and placing the result in the instance's output variable (ABS#_output). Setting *Source* to 0 will turn off automation.

Parameters

ABS# Absolute value block ID.

@Source Source variable.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒

get_ABS(2) *Get the absolute value of the input variable*

This function returns the absolute value of the input variable *Value*. The result is returned as the value of the function and is also placed in the designated instance's output variable (ABS#_output).

Parameters

ABS# Absolute value block ID (e.g., ABS2).

Value Source variable.

Returns

0 Absolute value = 0, or error.

n Absolute value.

Reported Errors: BAD_ID, AUTO_RUN.

A•3

MTD

MULTIPLY THEN DIVIDE

3.1 USING MTD

The functions in this group perform a full 16-bit multiplication, followed by a division. The results are guaranteed not to truncate, no matter how large the input values are.

3.2 MTD DATA

CONSTANT GROUP ⇒ **MTD#** *Block ID*

These constants define the four MTD data blocks: MTD1, MTD2, MTD3, MTD4.

VARIABLE GROUP ⇒ **MTD#_output** *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of MTD3 is placed in the variable MTD3_output.

3.3 MTD FUNCTIONS

FUNCTION ⇒ **automate_muldiv(4)** *Run the Multiply/Divide function automatically*

This function automates the multiplication of *Source* by *Multiplier* followed by division by *Divisor*. Like all automate functions, it is intended for use in a signal chain. Its return lets you know whether it could indeed be automated.

Setting *Source* to zero will disable automation.

Execution rate is 4ms.

Parameters

<i>MTD#</i>	Multiply/Divide function block ID (e.g., MTD3).
<i>@Source</i>	Source (input) variable.
<i>Multiplier</i>	The input will be multiplied by this value first.
<i>Divisor</i>	The input will be divided by this value, after multiplication.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE, PARAM_RANGE.

FUNCTION \Rightarrow **get_muldiv(4)** *Run a user-initiated Multiply/Divide function*

This function returns the end value of a user-specified initial input *Value* multiplied by the *Multiplier* parameter and then divided by the *Divisor* parameter.

Parameters

<i>MTD#</i>	Multiply/Divide function block ID (e.g., MTD3).
<i>Value</i>	Input value that will be multiplied, then divided.
<i>Multiplier</i>	The input will be multiplied by this value first.
<i>Divisor</i>	The input will be divided by this value, after multiplication.

Returns

n *Value* multiplied by *Multiplier*, then divided by *Divisor*.

Reported Errors: BAD_ID, AUTO_RUN.

A•4

RND

RANDOM NUMBER GENERATOR

4.1 USING THE RANDOM NUMBER GENERATOR

The random number generator returns a 15-bit signed pseudo-random number. If you want, you can set a seed value different from the default seed value.

4.2 RND DATA

none

4.3 RND FUNCTIONS

FUNCTION ⇒ **random(0)** *Return a pseudo-random number*

This routine returns the next pseudo-random number.

Parameters: none.

Returns

pnumber A pseudo-random number.

Reported Errors: none.

FUNCTION ⇒ **random_seed(1)** *Seed the random number generator*

When the system starts up, the random number generator is automatically seeded with the value of KSI * 2. If you want, you can use some other pseudo-random number as the seed or you can write a particular number as the seed so that you always get the same pseudo-random sequence.

Parameters

Seed Value to use as the seed.

Returns: 1.

Reported Errors: none.

A•5

SCL
SCALING

5.1 USING THE SCL FUNCTIONS

Numbers must sometimes be scaled in automatically processed signal chains.

5.2 SCL DATA

CONSTANT GROUP ⇒

SCL# *Block ID*

These constants define the four SCL data blocks: SCL1, SCL2, SCL3, SCL4.

VARIABLE GROUP ⇒

SCL#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of SCL3 is placed in the variable SCL3_output.

VARIABLE GROUP ⇒

SCL#_sf *Scaling factor*

These variables hold the scaling factor for the corresponding data blocks; for example, the scaling factor for SCL3 is placed in the variable SCL3_sf.

5.3 SCL FUNCTIONS

FUNCTION ⇒

automate_scale(3) *Setup the scaling to be done automatically*

This function automates the scaling process.

The *Input* and *SF* parameters are both 16-bit signed integers. *SF* is interpreted as a percentage, with 32676 being 100 percent (positive). If *SF* is set to zero, its value will not be altered by this routine; in that case, you can set the scaling factor by using a call to `setup_scale_factor()`.

Execution rate is 4ms.

Parameters

<i>SCL#</i>	Scaling block ID (e.g., SCL4).
<i>@Input</i>	Input variable (the number to be scaled).
<i>@SF</i>	Scaling factor variable.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **scale_value(2)** *Return value of the input scaled by the scaling factor*

This function returns the value scaled by the block's scaling factor. The scaling factor is interpreted as a 16-bit signed integer. For example, if the scaling factor is 16,383 (=50%) and the input value is 400, the output will be 200.

Parameters

SCL# Scaling block ID (e.g., SCL4).
Input Input variable.

Returns

0 Scaling value =0, or error.
 n Scaling value.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **setup_scale_factor(2)** *Setup a scaling factor*

This function installs a scaling factor in the designated scaling block. The scaling factor is a signed 16-bit integer that will be interpreted as a percentage (e.g., 32767=100% and 16383=50%, etc.).

Alias: setup_scaling()

Parameters

SCL# Scaling block ID (e.g., SCL4).
SF Scaling factor.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

• B •

SIGNAL PROCESSING FUNCTIONS

One of VCL's unique characteristics is its built-in signal processing. Control systems require proper signal conditioning and routing. VCL includes powerful functions that can be used to simplify the system control software.

6. The Copy (**CPY**) functions allow you
to copy a variable from one location
to another automatically.13
7. The Filter (**FLT**) function allows you
to filter incoming signals automatically.15
8. The Limit (**LIM**) functions allow you
to limit a signal in several ways.16
9. The Map (**MAP**) functions allow you
to setup 2-D maps.18
10. The Ramp (**RMP**) functions provide values
that vary linearly with time.22
11. The Selector Switch (**SEL**) functions are the
software equivalent of mechanical switches;
they are useful in building adaptive single paths
that run automatically.27

B•6

CPY

COPY ONE VARIABLE TO ANOTHER

6.1 USING COPY FUNCTIONS

The CPY functions allow you to copy a variable from one place to another, automatically. This is occasionally necessary when setting up signal chains. You can copy a single variable, a group of sequential variables, or up to four individual variables.

6.2 CPY DATA

CONSTANT GROUP ⇒

CPY# *Block ID*

These constants define the four CPY data blocks: CPY1, CPY2, CPY3, CPY4.

6.3 CPY FUNCTIONS

FUNCTION ⇒

automate_copy(3) *Automatically transfer variables*

This function copies a variable from *Source* to *Destination*.

To disable automation, set *Source* and *Destination* to zero.

Parameters

CPY# Copy block ID (e.g., CPY3).

@Source Source variable.

@Destination Destination variable.

Returns: 1, 0.

Reported Errors: BAD_ID, PARAM_RO, PT_RANGE.

FUNCTION ⇒ **automate_block_copy(4)** *Automatically transfer a sequential block of variables*
 This function copies a sequential block of variables from one location to another location.

The source and destination blocks cannot overlap.

To disable automatic copying, set *Source*, *Destination*, and *Count* to zero (e.g., `automate_block_copy(CPY3,0,0,0)`).

Parameters

<i>CPY#</i>	Copy block ID (e.g., CPY3).
<i>@Source</i>	Initial source variable.
<i>@Destination</i>	Initial destination variable.
<i>Count</i>	Number of variables to transfer.

Returns: 1, 0.

Reported Errors: BAD_ID, PARAM_RO, PT_RANGE.

FUNCTION ⇒ **automate_limited_block_copy(9)** *Automatically transfer up to four variables*
 This function copies up to four variables from one location to another location.

The *Source* and *Destination* address for each variable transfer must be specified. To turn off automatic copying for any source/destination pair, set either its *Source* or *Destination* to zero.

Parameters

<i>CPY#</i>	Copy block ID (e.g., CPY3).
<i>@Source1</i>	Source variable 1.
<i>@Source2</i>	" " 2.
<i>@Source3</i>	" " 3.
<i>@Source4</i>	" " 4.
<i>@Destination1</i>	Destination variable 1.
<i>@Destination2</i>	" " 2.
<i>@Destination3</i>	" " 3.
<i>@Destination4</i>	" " 4.

Returns: 1, 0.

Reported Errors: BAD_ID, PARAM_RO, PT_RANGE.

B•7

FLT
FILTERS

7.1 USING FILTERS

The filter function gives you the ability to filter incoming signals both automatically and independent of instruction timing. Such filtering is sometimes necessary in constructing signal chains.

7.2 FLT DATA

CONSTANT GROUP ⇒

FLT# *Block ID*

These constants define the four FLT data blocks: FLT1, FLT2, FLT3, FLT4.

VARIABLE GROUP ⇒

FLT#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of FLT3 is placed in the variable FLT3_output.

7.3 FLT FUNCTION

FUNCTION ⇒

automate_filter(5) *Filter a value*

This function initiates filtering on the selected variable.

Place the automate function in your code's one-time initialization section.

Parameters

<i>FLT#</i>	Filter block ID (e.g., FLT3).
<i>@Source</i>	Address of the source variable.
<i>Gain</i>	Filter gain (0 to 32767).
<i>PreLoad</i>	Initial value.
<i>Rate</i>	Number of ticks between filter updates.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

B•8

LIM
LIMITS

8.1 USING LIMITS

This group of functions allows you to limit a signal in several ways.

8.2 LIM DATA

CONSTANT GROUP ⇒

LIM# *Block ID*

These constants define the four LIM data blocks: LIM1, LIM2, LIM3, LIM4.

CONSTANT GROUP ⇒

See List *Limit types*

These constants are used to specify the type of limit that will be applied.

HARD_UPPER_LIMIT *Hard upper limit (0 to ±32765)*

HARD_LOWER_LIMIT *Hard lower limit (0 to ±32765)*

HARD_UPPER_UNSIGNED *Hard upper limit (0 to 65536)*

HARD_LOWER_UNSIGNED *Hard lower limit (0 to 65536)*

VARIABLE GROUP ⇒

LIM#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of LIM3 is placed in the variable LIM3_output.

VARIABLE GROUP ⇒

LIM#_limit *Limiting value*

These variables hold the limit values for the corresponding data blocks; for example, the limit value for LIM3 is placed in the variable LIM3_limit.

8.3 LIM FUNCTIONS

FUNCTION ⇒

automate_limit(3) *Automate a limit function*

This function sets one or both of the values in a limit block to be automatically updated. Setting either *Source* or *Limit* to zero will remove it from the automate function and put it under program control.

Parameters

LIM# Limit block ID (e.g., LIM3).

@Source Source variable.

@Limit Limit variable.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **get_limit(2)** *Get the limited value*

This function returns the input as clamped by the limit.

Parameters

LIM# Limit block ID (e.g., LIM3).
Value Limit value.

Returns

0 Limited value = 0, or error.
n Limited value.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **set_limit(2)** *Set the limit value*

This function installs a new value for a limit.

Parameters

LIM# Limit block ID (e.g., LIM3).
Value Limit value.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **setup_limit(3)** *Setup a limit function*

This function specifies the type of limit for a limit block.

Parameters

LIM# Limit block ID (e.g., LIM3).
Type Type of limit (e.g., HARD_UPPER_LIMIT); setting
this to 0 disables the limit block.
Value Initial value of limit.

Returns: 1, 0.

Reported Errors: BAD_ID, TYPE_SETUP.

B•9

MAP

2-DIMENSIONAL MAPS

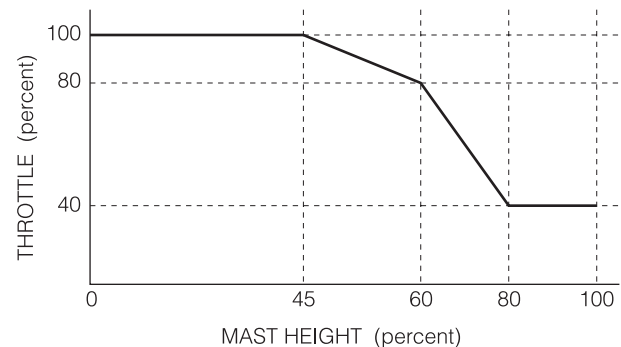
9.1 USING MAPS

The functions in this group allow you to setup 2-D maps by specifying a set of X-Y points. 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. To use the map, you pass in an X value and get back the Y value. Points that are not explicitly set are determined by using linear approximation.

Setting up a map

As an example of map usage, consider the problem of limiting a forklift's speed as a function of mast height. You might allow full speed when 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).

Suppose we want to allow 100% throttle until the mast reaches 45% of its height. Then we want to reduce (linearly) the throttle to 80% of its requested value up to 60% of mast height. Then we want to reduce (again, linearly) the throttle to 40% of its requested input when the mast height is at 80% of its upper limit.



When you setup the map, you define the relationship of inputs to output by specifying up to seven pairs of points. The definition actually starts with the ID of the map you want to use followed by the number of point pairs used. This is followed by exactly seven pairs of points. For each pair, the X value comes first, followed by its corresponding Y. By convention, unused point pairs are set to zero. Here is the `setup_map()` function definition for this particular mapping; it assumes that the largest integer = 32767 = 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
```

Notice that the definition covers several lines, with comments interspersed as a reminder of the intent of the code. If you follow this convention, your code will be much more readable.

Returning values from a map

To use the map, you must specify the ID of a map and an X value. You can call the map function explicitly using the `get_map()` function. For example:

```
user2 = get_map_output (MAP1, POT1_output)
```

Or, more commonly, you will specify the map as part of an automatic signal chain, as in this example:

```
automate_map(MAP1, POT1_output)
```

Of course, when you use any of the automate instructions, you must place them in the one-time initialization section of code. And to use the value, you will have to reference the map's output (in this case `MAP1_output`).

Finding the applicable segment

Another aspect of maps in VCL is that you can find out the segment into which a given X-value falls. Segments in this sense are the lines connecting point pairs. If the X input is less than or equal to the lowest X-value defined, the segment number will be 0. If the X input is greater than the highest X-value, the segment number will be equal to the number of X-Y pairs in the table (i.e., 1 greater than the number of segments). For example, consider the following mapping, where `POT1_output` holds the signed value of a throttle input:

```
;Map 2: used as a wigwag (Full Plus - Zero - Full Plus)
;-----
; -This is a common mapping for wigwag throttles when you must
; select forward and reverse using binary inputs. In this case,
;
;
setup_map(MAP2,6,
    0,0x7FFF, ; Full Positive           X = 0%      Y = +100%
    1638,0x7333, ; Almost Full Positive       X = 5%      Y = +95%
    14746,0x0000, ; Zero Speed                 X = 45%     Y = +0%
    18022,0x0000, ; Zero Speed                 X = 55%     Y = -0%
    31129,0x7333, ; Almost Full Positive       X = 95%     Y = +95%
    32767,0x7FFF, ; Full Positive              X = 100%    Y = +100%
    0, 0) ; Last point pair not used
automate_map(MAP2, POT1_Output)
```

This mapping sets up what is commonly known as a wigwag throttle. It has a zero point in the middle of the X-value's range. As you move the X-value away from the center, in either the positive or negative direction, the value increases. Generally, in this kind of system, the motor controller will always drive in one direction: The direction is selected using a relay to switch the polarity to the motor. You can use the `get_map_segment()` function to control the direction relay by noting that:

```
Segments 1 and 2 are in the reverse range
Segment 3 is neutral
Segments 4 and 5 are in the forward range.
```

9.2 MAP DATA

CONSTANT GROUP ⇒ **MAP#** *Block ID*

These constants define the four MAP data blocks: MAP1, MAP2, MAP3, MAP4.

VARIABLE GROUP ⇒ **MAP#_output** *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of MAP3 is placed in the variable MAP3_output.

VARIABLE GROUP ⇒ **MAP#_segment** *Output segment*

These variables hold the segment numbers from the corresponding data blocks; for example, the segment number into which the present value of MAP3 falls is placed in the variable MAP3_segment. Zero indicates that the value is below the lowest defined segment. If the value is one greater than the last segment, that means the value is above the highest defined segment. Otherwise, the value will be the number of the segment in which the output falls.

9.3 MAP FUNCTIONS

FUNCTION ⇒ **automate_map(2)** *Setup the mapping to be done automatically*

Execution rate is 32ms.

Parameters

MAP# Map block ID (e.g., MAP3).

Source Source variable that will be mapped.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **get_map_output(2)** *Interpolate within a 2D table of N entries*

This function maps the input variable to a new value using the specified map.

Parameters

MAP# Map block ID (e.g., MAP3).

Value Value along the X-axis.

Returns

0 Mapped value = 0, or error.

n Mapped value.

Reported Errors: BAD_ID, UNIT_UNDEF, AUTO_RUN.

FUNCTION ⇒ **get_map_segment(2)** *Return the segment containing X*

This function returns the segment (0 to 7) of the input variable using the specified map. If the X input is less than the lowest X value in the map, the segment number will be 0. If the X input is greater than the highest X value, the segment number will be equal to the number of X-Y pairs (i.e., 1 greater than the number of segments). For example, if you have three X-Y pairs (44,123; 100,230; 300,5000) then setting:

<i>Value</i> = 33	would return	0
<i>Value</i> = 88	" "	1
<i>Value</i> = 150	" "	2
<i>Value</i> = 400	" "	3.

Parameters

<i>MAP#</i>	Map block ID (e.g., MAP3).
<i>Value</i>	Value along the X-axis.

Returns

0	Segment = 0, or error.
n	The segment containing the X parameter (<i>Value</i>).

Reported Errors: BAD_ID, UNIT_UNDEF, AUTO_RUN.

FUNCTION ⇒ **setup_map(16)** *Setup a map*

All seven pairs of points **must** be specified. Unused points should be set to 0.

Parameters

<i>MAP#</i>	Map block ID (e.g., MAP3).
<i>Length</i>	Length of map, defined as the number of data pairs (e.g., 3 = three X-Y pairs).

<i>X1</i>	X data point 1.
<i>Y1</i>	Y " " 1.
<i>X2</i>	X " " 2.
<i>Y2</i>	Y " " 2.
<i>X3</i>	X " " 3.
<i>Y3</i>	Y " " 3.
<i>X4</i>	X " " 4.
<i>Y4</i>	Y " " 4.
<i>X5</i>	X " " 5.
<i>Y5</i>	Y " " 5.
<i>X6</i>	X " " 6.
<i>Y6</i>	Y " " 6.
<i>X7</i>	X " " 7.
<i>Y7</i>	Y " " 7.

Returns: 1, 0.

Reported Errors: BAD_ID.

B•10

RMP
RAMPING

Ramping provides you with values that vary linearly with time. You can set the rate of change, the current value, the target value, and the hold control. While the hold control is True, the current value will remain frozen. When the hold control is False, the output will approach the target value at the specified rate. Ramps are useful when you want a variable's value to change without any dependency on the timing of your program.

In order for the ramp to be free to move, the target, hold, and rate parameters must be setup—either individually or by running `automate_ramp()` or `setup_ramp()`.

10.1 USING RAMPS

The primary task of the ramp function is to alter the “current value” at some fixed rate until it reaches a “target value”; and, occasionally, it may be necessary to arrest (or hold) the progress of the current value.

Four parameters are used to control a ramp:

<i>Target</i>	The final value of the ramp output.
<i>Current</i>	The present value of the ramp output.
<i>Hold</i>	Used to hold the ramp at its current value.
<i>Rate</i>	Controls the rate at which the ramp changes.

There are a number of ways to initialize, modify, and control these parameters. To begin with, there are two fundamental ways to setup the ramp function: under program control (“normally”) and automatically.

Setting up the ramp to run normally

To setup a ramp to run normally, you will use the `setup_ramp()` function. This function accepts the following four parameters:

<i>RMP#</i>	ID of the ramp block to be setup.
<i>Start value</i>	The start value is used to initialize target <u>and</u> current values. By convention, they are initialized to the desired current value (output value of the ramp).
<i>Hold</i>	Ramp hold value (0=move, !0=hold).
<i>Rate</i>	Ramp rate (number of steps per millisecond).

The typical way to setup the ramp function is to set *Start value* and *Rate* equal to the desired values and *Hold* to zero (to enable the ramp).

Using the ramp in normal operation

There are also functions to independently control the ramp's target value, hold control, and rate. Be aware that these functions only apply to ramps setup using the normal setup routine.

For example, if you setup the ramp with the hold control set to 0 (no-hold = movement enabled), you could use the `set_ramp_target()` function to install a new target value, which will cause the current value to immediately start moving toward the new target.

The ability to hold the ramp at a particular value is important in certain applications. For example, the speed of certain types of machines is controlled using Up and Down speed buttons. In this case, as long as the Up button is held, the machine will increase in speed until it reaches its maximum speed. As soon as the button is released, the machine's speed will no longer increase, but will remain at its current value. The hold control is very useful for this type of application. For example, you would have code to determine which button was pressed and set the target value accordingly: 0 if the Down button was pressed, 100% if the Up button was pressed. Once the target has been set, you would clear the hold control, allowing the current value to approach the target until the button was released, at which point you will again assert the hold control.

Ramp rate control is also useful in certain applications. It is sometimes the case that the rates for increasing values are different from the rates for decreasing values. This is easy to accommodate using the rate control function.

Setting up the ramp to run automatically

The automatic version of the ramp is useful in a number of situations—in particular, when setting up signal chains. The automatic version of the ramp is mainly different in that all of the parameters are controlled indirectly.

Before you setup a ramp to run automatically, you will have to decide on which user variables you will use to hold its parameters. More to the point, you will need to initialize the user variables *before* you run the automate ramp function. You must also initialize the appropriate ramp output (e.g., `RMP2_output=123`). This is very important because the output of the function will begin changing as soon as the setup function is executed. If you don't properly initialize all of the elements first, you will have an undefined output — not a good thing!

The values specified by the `automate_ramp()` function are very similar to those of the individual normal functions:

<i>RMP#</i>	ID of the ramp block to be setup to run automatically.
<i>@Target</i>	Predefined target (final) value.
<i>@Hold</i>	Predefined ramp hold control value.
<i>@Rate</i>	Predefined ramp rate value.

There are two critical differences to notice. Most obviously, the values are specified indirectly. Second, only the target value is specified (not the target and current value). This means that it is up to you to set the target equal to the current value (using `set_ramp_target()` or `setup_ramp()`) before you run the `automate` function.

Using the ramp in automatic operation

In automatic mode, the individual Set functions for *Target*, *Hold*, and *Rate* have no effect. It is up to you (or the other components in your signal chain) to control these elements directly (i.e., by directly altering the identified user variables). Other than that, there is no difference between the functionality offered by the normal and automatic versions of the ramp functions.

10.2 RMP DATA

CONSTANT GROUP ⇒

RMP# *Block ID*

These constants define the four RMP data blocks: RMP1, RMP2, RMP3, RMP4.

VARIABLE GROUP ⇒

RMP#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of RMP3 is placed in the variable RMP3_output.

VARIABLE GROUP ⇒

RMP#_hold *Hold control*

These variables hold flags for the corresponding data blocks; for example, the flag for RMP3 is placed in the variable RMP3_hold. A flag is used to hold the ramp at its present value. If the value is zero, the ramp is free to move. If the value is non-zero, the ramp is held at its current value.

10.3 RMP FUNCTIONS

FUNCTION ⇒ **automate_ramp(4)** *Setup ramping to be done automatically*

This function allows you to setup a ramp that runs automatically on each clock tick. You can specify one variable that will be the target and another that will control the ramp hold. A value of zero for a given parameter allows you to maintain program control over it (by using the setup function or by writing the variable directly). For example, to maintain local control over the target but have automatic control over the ramp hold (using whatever value is written to user37), you'd set *Target* to 0 and *Hold* to user37.

Parameters

<i>RMP#</i>	Ramp block ID (e.g., RMP2).
<i>@Target</i>	Ramp target (final) variable.
<i>@Hold</i>	Ramp hold control variable.
<i>@Rate</i>	Ramp rate variable.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **set_ramp_hold(2)** *Set the ramp hold value*

This function installs a new value for the hold control. A value of zero enables the ramps. Any non-zero value will hold the ramp at its present value.

Parameters

<i>RMP#</i>	Ramp block ID (e.g., RMP2).
<i>Hold</i>	Ramp hold value (0=move, !0=hold)

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **set_ramp_rate(2)** *Set the ramp rate value*

This function allows you to alter value for the ramp's rate. This is particularly useful when you have dual ramp rates (e.g., normal and deceleration).

On each 1ms clock tick, the ramp rate value you set will be added or subtracted from the current ramp value if it is not already equal to the target value.

Parameters

RMP# Ramp block ID (e.g., RMP2).

Rate The maximum ramp rate increment per millisecond.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **set_ramp_target(2)** *Set the target ramp value*

This function installs a new target value.

Parameters

RMP# Ramp block ID (e.g., RMP2).

Target Ramp target value.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **setup_ramp(4)** *Setup a ramp*

This function initializes the ramp parameters.

Parameters

RMP# Ramp block ID (e.g., RMP2).

Start value Initial value; see page 22.

Hold Ramp hold value (0=move, !0=hold).

Rate Ramp rate (number of steps per millisecond).

Returns: 1, 0.

Reported Errors: BAD_ID.

B•11

SEL
SELECTOR SWITCH

11.1 USING SELECTOR SWITCHES

The selector switch functions are the software equivalent of mechanical switches. These are very useful in building adaptive single paths that run automatically.

Two types of switches are emulated: a 1-pole, double-throw switch (1PDT) and a 4-pole double-throw (4PDT) switch.

11.2 SEL DATA

CONSTANT GROUP ⇒

SEL# *Block ID*

These constants define the eight SEL data blocks: “switches” SEL1 through SEL8.

VARIABLE GROUP ⇒

SEL#_output *Output value, 1PDT switch*

These variables hold the output from the corresponding data blocks for 1PDT switches. For example, assuming SEL3 is a 1PDT switch, its output is placed in the variable SEL3_output.

VARIABLE GROUP ⇒

SEL#_4p_n_output *Output value, 4PDT switch*

These variables hold the outputs from the corresponding data blocks for 4PDT switches. For example, assuming SEL2 is a 4PDT switch, its four outputs are placed in the variables SEL2_4p_1_output, SEL2_4p_2_output, SEL2_4p_3_output, and SEL2_4p_4_output.

VARIABLE GROUP ⇒

SEL#_select *Steering flag, 1PDT switch*

These variables hold the steering flag for the corresponding data blocks for 1PDT switches. For example, assuming SEL3 is a 1PDT switch, its steering flag is placed in the variable SEL3_select. This flag then selects the primary (zero) or secondary (non-zero) output.

VARIABLE GROUP ⇒

SEL#_4p_select *Steering flag, 4PDT switch*

These variables hold the steering flag for the corresponding data blocks for 4PDT switches. For example, assuming SEL2 is a 4PDT switch, its steering flag is placed in the variable SEL2_select. This flag then selects the primary (zero) or secondary (non-zero) output for each pole.

11.3 SEL FUNCTIONS

FUNCTION ⇒ **automate_select(2)** *Automate the selector switch operation (1PDT switch)*

This function causes the input selection to be performed automatically.

If you set *SwitchID* to zero, the automatic operation of the switch will be disabled (i.e., it will return to normal program control, and `setup_select()` can be used to configure the switch).

Parameters

SEL# Selector switch ID (e.g., SEL3).

@SwitchID Variable to use as the steering flag.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **set_select(2)** *Enter a new value for the steering flag (1PDT switch)*

This function sets the value of the steering flag that is used to choose which input appears at the output (*SEL#_output*).

Parameters

SEL# Selector switch ID (e.g., SEL3).

Selector New value for the steering flag.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **setup_select(3)** *Setup a selector switch (1PDT switch)*

This function installs new values for a 2-position selector switch.

If you set *Zero* to 0 the selector switch functions will be disabled.

Parameters

SEL# Selector switch ID (e.g., SEL3).

@Zero Variable to copy to output if the steering flag is zero.

@NonZero Variable to copy to output if the steering flag is non-zero.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **automate_4p_select(2)** *Automate the selector switch operation (4PDT switch)*

This function causes the input selection to be performed automatically.

If you set *SwitchID* to zero, the automatic operation of the switch will be disabled (i.e., it will return to normal program control, and `setup_select()` can be used to configure the switch).

Parameters

SEL# Selector switch ID (e.g., SEL2).

@SwitchID Variable to use as the steering flag.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **set_4p_select(2)** *Enter a new value for the steering flag (4PDT switch)*

This function sets the value of the steering flag that is used to choose which input set appears at the output (*SEL#_output*): the “zero” set or the “non-zero” set.

Parameters

SEL# Selector switch ID (e.g., SEL2).

Selector New value for the steering flag.

Returns: 1, 0.

Reported Errors: BAD_ID, AUTO_RUN.

FUNCTION ⇒ **setup_4p_select(3)** *Setup a selector switch (4PDT switch)*

This function installs new values for a 4-pole, double-throw selector switch.

You can set any *Zero* or *NonZero* parameter to 0 to disable its transfer, and the value of the output will remain unchanged. For example, if you setup SEL2 with P3_Zero and P3_NonZero both set to zero, the value of SEL2_4p_3_output will not be changed. If you set P3_Zero to zero (but have assigned a variable to P3_NonZero), the value of SEL2_4p_3_output will only be changed when the steering flag is non-zero.

Parameters

<i>SEL#</i>	Selector switch ID (e.g., SEL2).											
<i>@P1_Zero</i>	Variable to copy to output if the steering flag is 0.											
<i>@P2_Zero</i>	"	"	"	"	"	"	"	"	"	"	"	"
<i>@P3_Zero</i>	"	"	"	"	"	"	"	"	"	"	"	"
<i>@P4_Zero</i>	"	"	"	"	"	"	"	"	"	"	"	"
<i>@P1_NonZero</i>	Variable to copy to output if the steering flag is !0.											
<i>@P2_NonZero</i>	"	"	"	"	"	"	"	"	"	"	"	"
<i>@P3_NonZero</i>	"	"	"	"	"	"	"	"	"	"	"	"
<i>@P4_NonZero</i>	"	"	"	"	"	"	"	"	"	"	"	"

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

• C •

TIMING FUNCTIONS

Time is a major defining factor in system control. Vehicle and system control often requires precise and repeatable time-activated actions. VCL's timing functions provide the necessary precision and reliability. They should be used in lieu of counting loops whenever greater time accuracy is required.

- 12. The Delay (**DLY**) functions allow you to setup
delays relative to the system clock.32
- 13. The Timer (**TMR**) functions allow you to setup
timers to record elapsed time.34

C•12

DLY
TIME DELAYS

12.1 USING DELAYS

The functions in this group allow you to setup timers whose values will automatically decrement relative to the system clock. This allows you write programs and subroutines that have well-behaved timing characteristics.

12.2 DLY DATA

CONSTANT GROUP ⇒

DLY# *Block ID*

These constants define the 16 DLY data blocks: DLY1 through DLY16.

VARIABLE GROUP ⇒

DLY#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of DLY5 is placed in the variable DLY5_output.

12.3 DLY FUNCTIONS

FUNCTION ⇒

setup_delay(2) *Setup a time delay*

This function installs a new time delay. The length of time for a single tick is 1ms by default; however, this can be changed using the setup_delay_prescale() function.

Parameters

DLY# Delay block ID (e.g., DLY3).
Ticks Number of ticks to delay.

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ **setup_delay_prescale(2)** *Setup a delay timer's pre-scale*

This routine sets up a pre-scale for a time delay block. The default pre-scale is one (i.e., the timer counts down at a 1ms rate).

Using the default settings, a call to `setup_delay()` with a *Ticks* value of 25 would result in a 25ms delay period. If you set *PreScale* to 10, a call to `setup_delay()` with a *Ticks* value of 25 would result in a 250ms delay period.

Parameters

<i>DLY#</i>	Delay block ID (e.g., DLY3).
<i>PreScale</i>	Multiplier used with the <i>Ticks</i> parameter in <code>setup_delay()</code> .

Returns: 1, 0.

Reported Errors: BAD_ID.

C•13

TMR
TIMERS (HOURMETERS)

The timers provide the ability to record elapsed time up to 200,000 hours.

13.1 USING TIMERS

These functions allow you to set up to three different hourmeters. The timers provide the ability to record elapsed time up to 200,000 hours.

13.2 TMR DATA

CONSTANT GROUP ⇒

TMR# *Access ID*

These constants identify three timers: TMR1, TMR2, TMR3.

VARIABLE GROUP ⇒

TMR#_ms_output *Output value in milliseconds*

One variable exists for each timer, and holds that timer's present output in milliseconds.

The range is 0 to 6000 (0 to 1/10 minute).

VARIABLE GROUP ⇒

TMR#_sec_output *Output value in 1/10 minutes*

One variable exists for each timer, and holds that timer's present output in 6sec increments (i.e., 1/10 of a minute).

The range is 0 to 6000 (0 to 10 hours).

VARIABLE GROUP ⇒

TMR#_hr_output *Output value in tens of hours*

One variable exists for each timer, and holds that timer's present output in tens of hours.

The range is 0 to 20000 (0 to 200,000 hours)

VARIABLE GROUP ⇒

TMR#_enable *Timer enable values*

These locations hold the current status of the timers (enabled/disabled).

If the value is non-zero, the timer is running.

If the value is zero, the timer is disabled.

13.3 TMR FUNCTIONS

FUNCTION ⇒ **disable_timer(1)** *Disable a timer*

This function disables a timer (stops it from timing).

Parameters

TMR# ID of timer to initialize.

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ **enable_timer(1)** *Enable a timer*

This function enables a timer (starts it timing).

Parameters

TMR# ID of timer to initialize.

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ **reset_timer(2)** *Reset a timer to zero*

This function resets all the timer's values (milliseconds, seconds, minutes, hours, and days) to zero and then sets its TMR_enable flag equal to the *Enable* parameter.

Parameters

TMR# ID of timer to initialize.

Enable Enable (non-0) or disable (0) the timer.

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ **setup_timer(4)** *Setup a timer*

This function initializes a timer to a specific value.

Parameters

TMR# ID of timer to initialize.

Ms Initial value of milliseconds (6000max=0.1 minute).

Min " " " 1/10 minutes (6000max=10 hours).

Hr " " " 10's of hours (20000max=200,000 hours).

Returns: 1, 0.

Reported Errors: BAD_ID.

• D •

CONTROL FUNCTIONS

Classic control systems use feedback to drive the process. The standard way to do this is with a PID loop. VCL's PID functions help to ensure high-speed and accurate calculations.

14. The Proportional, Integral, Derivative (**PID**) functions
allow you to setup a time-dependent control system.37

D•14

PID PROPORTIONAL, INTEGRAL, DERIVATIVE

14.1 USING PID FUNCTIONS

The PID loop provides the classic method of controlling time-dependent systems.

14.2 PID DATA

CONSTANT GROUP ⇒

PID# *Block ID*

These constants define the two PID data blocks: PID1, PID2.

VARIABLE GROUP ⇒

PID#_output *Output value*

These variables hold the output from the corresponding data blocks; for example, the output of PID2 is placed in the variable PID2_output.

14.3 PID FUNCTIONS

FUNCTION ⇒ **automate_PID(8)** *Setup a PID loop to run automatically*

This function allows you to setup a PID loop that runs automatically on each clock tick.

If you pass a value of zero for *Setpoint*, *Feedback*, *Kp*, *Ki*, or *Kd*, the present PID value will not be altered. This can save you some variables and make your program a little easier to understand.

The value of *Kp* is normalized to a value of 16384. This means that when the *Setpoint* and *Feedback* both range from 0 to 32767, a value of 16384 for *Kp* will cause the loop to track the setpoint. A lower value for *Kp* will result in the output being less than the setpoint, while a higher value for *Kp* will result in the output being greater than the setpoint.

You may need to use the Multiply-then-Divide (MTD) and/or scaling (SCL) functions to adjust the *Setpoint* and *Feedback* values.

Place the automate function in your code's one-time initialization section.

Parameters

<i>PID#</i>	PID block ID (e.g., PID2).
<i>@Setpoint</i>	Setpoint (input) variable.
<i>@Feedback</i>	Feedback source variable.
<i>@Kp</i>	Proportional scaling coefficient.
<i>@Ki</i>	Integral scaling coefficient.
<i>@Kd</i>	Derivative scaling coefficient.
<i>FB sign</i>	Feedback sign (0=positive feedback, !0=negative).
<i>Delay</i>	Number of 512 μ s ticks between PID loop iterations.

Returns: 1, 0.

Reported Errors: BAD_ID, PT_RANGE.

FUNCTION ⇒ **reset_PID(1)** *Reset the PID loop variables*

This function resets all the PID loop variables to zero.

Parameters

<i>PID#</i>	PID block ID (e.g., PID2).
-------------	----------------------------

Returns: 1, 0.

Reported Errors: BAD_ID.

• E •

MEMORY ORGANIZATION FUNCTIONS

The microprocessors in Curtis controllers have several types of memory. Some (such as firmware code) is fixed; some (RAM) is temporary; and some (EEPROM) retains values even after power off. VCL includes functions to help you ensure that your data is safely stored.

- 15. The Nonvolatile Memory (**NVM**) functions allow you to configure and manipulate the two sections of nonvolatile memory that are accessible to VCL: the section containing the 40 NVM user blocks and the section containing the persistent parameters.40
- 16. The System-Level (**SYS**) functions are related to data management at system startup and shutdown.48
- 17. The **VCL** interpreter functions allow you to access individual bytes within variables.51

E•15

NVM NONVOLATILE MEMORY

15.1 USING NONVOLATILE MEMORY

Two sections of nonvolatile memory are accessible to VCL: the section made up of the 40 NVM user blocks, and the section containing the persistent parameters.

All but one of the functions in this group apply to the nonvolatile memory section containing the NVM user blocks; the last function applies only to the persistent parameter section. Some of the variables apply to only one or the other of these sections, while many apply to both.

Access to the nonvolatile memory is scheduled. So, unlike most operations in VCL, it will usually take some time for a given operation to complete. Consequently, you must monitor the return status of the function call to see if the request was accepted and then monitor the variable `NVM_status` to find out when the operation has completed. Nonvolatile memory functions generally return a non-zero result if the requested operation was actually initiated. The `NVM_status` value will remain non-zero until the operation completes.

For example, let us assume that you wish to read the contents of NVM4, elements 3 through 8, into the user variables `user23` through `user28`. The proper way to do this (assuming that your program can tolerate hanging) is by writing:

```
while (NVM_block_read(NVM4,3,8, user23) = 0) {}
while (NVM_status <> 0) {}
```

The first line waits for the operation to be accepted. The second line waits for the operation to be completed. (A similar operation could be done in the persistent parameter section, by substituting `NVM_write_parameter()` for `NVM_block_read()` in this example.)

NVM User Block Section

Restoring the NVuser# variables

The values `NVuser1` through `NVuser15` can be used as you would any of the other user variables (`user1` through `user120`). However, this 15-element block of variables is different in that it is automatically saved to nonvolatile memory periodically and on power failure. If your nonvolatile memory needs are modest (15 variables or fewer), then all you need to do is to use the `NVM_nvuser_restore()` function to satisfy your application's needs for nonvolatile memory.

By default, once every six minutes, the `NVuser` variables are saved to `NVM2`. This rate can be altered using the `set_NVM_save_rate()` function; however, care should be taken to not exceed the maximum number of writes to the EEPROM within the lifetime of the system. For the memory currently in use, the maximum

number of writes is 1 million. Thus, one write every 6 minutes implies a service life of 100,000 hours.

When the system detects that the power has failed (see the `setup_power_fail()` routine description) the NVuser variables are saved to NVM1.

Although NVuser1 through NVuser15 are automatically saved, they are not automatically *restored*. To do this, you must execute the `NVM_nvuser_restore()` function. This function attempts to restore the NVuser variables from NVM1 first. If that fails, the function will attempt to restore the NVuser variables from NVM2.

The reason for this recovery strategy is the expectation that the save on power-failure will occasionally be unsuccessful. It is quite possible that the combination of loads on a system will cause the power to a given system to fail so quickly there is simply not enough time to write the NVuser variables. In this case, we simply fall back to the last periodic save—which, on average, will be current within 3 minutes.

Here is an example of how to restore the NVuser variables:

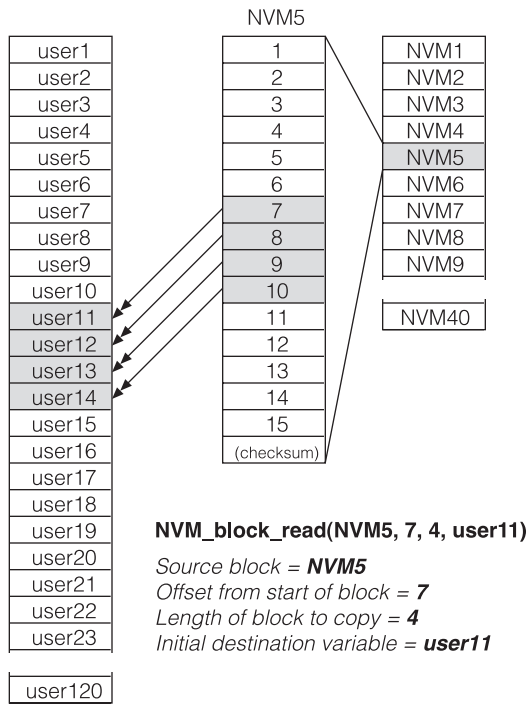
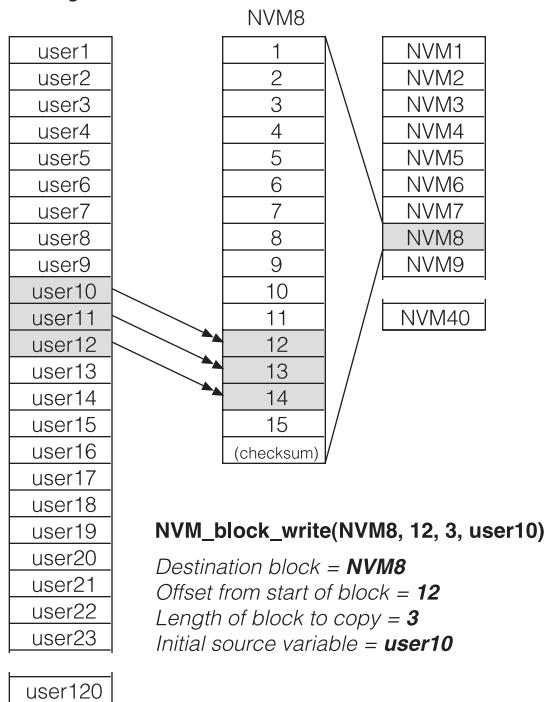
```
while (NVM_nvuser_restore() = 0) {}
while (NVM_status <> 0) {}

if (NVM_restore_source = NVM_RESTORED_NVM1)
{
    ; restored from NVM1
}
else if (NVM_restore_source = NVM_RESTORED_NVM2)
{
    ; restored from NVM2
}
else
{
    ; NVM_restore_source will = NVM_RESTORED_NONE (neither NVM1 nor
    NVM2 was restored)
}
```

Finally, be aware that the process of restoring the NVuser variables will cause the destruction of NVM1. When the `NVM_nvuser_restore()` routine successfully recovers the information from NVM1, it intentionally corrupts the block. The behavior helps to guarantee that the most recent block (NVM1-power-fail-save-block or NVM2-periodic-save-block) is restored. Consider the following example. The system has been running for eight hours when the power fails so quickly that the system is unable to even start to save to the power-fail-save block (NVM1). If NVM1 were not intentionally corrupted, it could still be read successfully. In this case, the contents of NVM1, which are eight hours old, would be restored. The contents of NVM2 (the periodic-save block), which are only 6 minutes old, would be ignored.

Reading from and writing to NVM user blocks

The most important thing to keep in mind is that read and write operations to the nonvolatile memory take time. This means that if you request an operation, you must not assume that it has been accepted. Instead, you must test the return value of the function to make sure it was accepted.

Reading from NVM user blocks:**Writing to NVM user blocks:**

Aside from timing considerations, nonvolatile reads and writes are easy. You must specify the nonvolatile block you want to access (NVM1 through NVM40), the offset within that block (0 to 15), the number of consecutive items you want to read/write (again, 0 to 15), and the initial destination variable (for reads) or initial source variable (for writes).

If your application can wait for the nonvolatile operation to complete, the code becomes quite simple (see the example in the introduction to this section). If your application cannot wait, your code will be a little more complex, depending on your needs.

For example, consider the case where you only occasionally need to write data to nonvolatile memory (i.e., you don't have to stack requests). The typical strategy is to put the nonvolatile memory access routines in their own subroutine. Within this subroutine you would have a state variable. When you need to access the nonvolatile memory, you call this routine after setting a request flag.

```
write_NVM_block:

; Request the operation
if (NVM_request = 1)
{
  if (NVM_block_write(NVM4,3,8, user23) <> 0)
  {
    NVM_request = 2
  }
}

; Determine if the operation has completed
else if (NVM_request = 2)
{
  if (NVM_status <> 0)
  {
    NVM_request = 0
  }
}

return;
```

Formatting NVM user blocks

The nonvolatile memory is organized as blocks of 15 variables with a checksum. The format command operates on an entire blocks in one of two ways.

You can use the format command to initialize a block, setting all the elements to a common value and recalculating the checksum. For example, to format NVM block 3 with the value 0xFFFF (or -1 in decimal), you would write:

```
while (NVM_block_format(NVM3, 1, 0xFFFF) = 0) {}
```

Alternatively, you can use the format command in an attempt to recover a corrupted block by having it only recalculate the

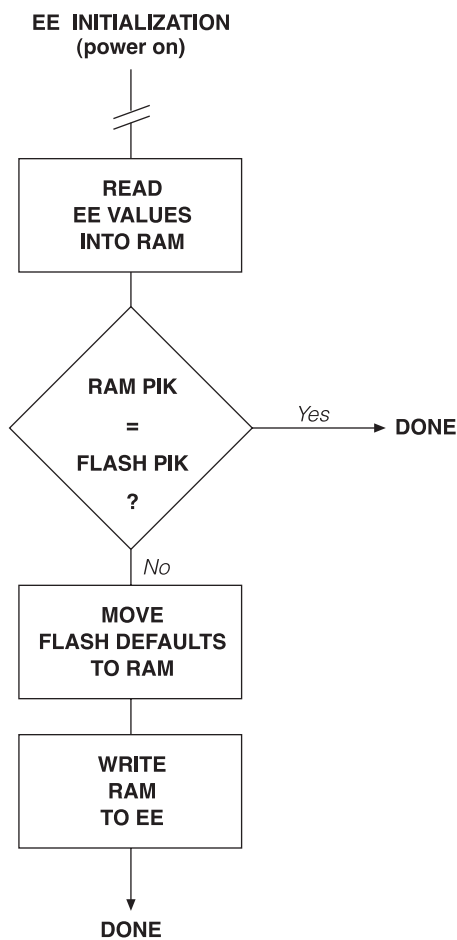
checksum. For example, to try to recover NVM block 3 without altering the contents, you would write:

```
while (NVM_block_format(NVM3, 0, 0) = 0) {}
```

Although the last parameter is ignored, it is set to zero by convention.

Persistent Parameter Section

This section of nonvolatile memory holds a variety of parameters, which have in common the characteristic of being persistent. These parameters include 1311 parameters, CAN objects, and variables that have Curtis-predefined default values.



[PIK = Parameter Interlock Key]

Initialization of persistent parameters

When the nonvolatile memory (EE) is initialized on startup, the variable `parameter_interlock_key` is compared to the corresponding parameter interlock key (PIK) in Flash; if the two PIKs match, the existing EE values for the persistent parameters are maintained. If the two PIKs do *not* match, the Flash default values for the persistent parameters are written into EE, replacing the earlier non-matching values. This ensures that the persistent parameter values are the correct ones for the current OS.

The variable `parameter_defaults_restore_source` indicates the source from which the persistent parameters were restored:

DEFAULTS_FROM_EE	<i>Restored from EE</i>
DEFAULTS_FROM_FLASH	<i>Restored from Flash</i>
DEFAULTS_NOT_RESTORED	<i>Defaults were not restored.</i>

Writing persistent parameters

The function `NVM_write_parameter()` can be used to write a new value of a persistent parameter to nonvolatile memory. This function allows you to modify the value of any persistent parameter; the new value will be written from RAM to nonvolatile memory and will replace the previous value.

15.2 NVM DATA

CONSTANT GROUP ⇒ **NVM#** *Block ID*

These constants identify the 40 NVM user data blocks: NVM1 through NVM40. The first two are predefined:

NVM1 *Holds NVuser variables upon power fail*

NVM2 *Holds NVuser variables from most recent periodic write.*

NVM3 through NVM40 are user defined.

VARIABLE ⇒ **NVM_restore_source** *Restore source indicator*

This variable shows the source of the restored values when the NVM_nvuser_restore() function is executed. The following codes are set in this variable:

NVM_RESTORED_NVM1 *Restored from power-fail-save block (NVM1)*

NVM_RESTORED_NVM2 *Restored from periodic-save block (NVM2)*

NVM_RESTORED_NONE *Unable to restore from either NVM1 or NVM2; NVuser variables are set to zero.*

VARIABLE ⇒ **NVM_result** *Result of the current operation*

This variable shows the result of the intrinsic nonvolatile memory access routines. These routines are used by the user nonvolatile memory access routines and the error handler, and are also used by the system function group (SYS) in the process of saving and restoring the system block.

The following codes are set in this variable:

NV_OP_ACCESS *Couldn't access SPI bus*

NV_OP_CHECKSUM *Checksum error*

NV_OP_MIN_KSI *KSI too low to allow write*

NV_OP_SPI *SPI Bus error*

NV_OP_SUCCESS *Successful completion.*

VARIABLE ⇒ **NVM_status** *Current NVM access status*

This variable indicates the current NVM operation being performed, using the following codes:

NVM_FORMAT *Format in progress*

NVM_IDLE *No operation in progress*

NVM_PARAMETER *Writing parameter*

NVM_READ *Read in progress*

NVM_RESTORE *Restore in progress*

NVM_WRITE *Write in progress.*

NVM_status must be NVM_IDLE before the read or write routines will accept a new job.

Warning! You should only **read** NVM_status; you should never **write** to it.

VARIABLE GROUP ⇒ **NVuser#** *Automatically saved user variables*

These 15 variables (NVuser1 through NVuser15) are automatically saved into the power-fail-save block (NVM1) and periodic-save block (NVM2).

VARIABLE ⇒ **parameter_interlock_key** *OS-unique key in EE to match with Flash key*

This variable is used to determine whether the persistent parameters need to be refreshed.

VARIABLE ⇒ **parameter_defaults_restore_source** *Source of restored parameters*

This variable indicates the source from which persistent parameters were restored on initialization, using the following codes:

DEFAULTS_FROM_EE	<i>Defaults were restored from EE</i>
DEFAULTS_FROM_FLASH	<i>Defaults were restored from Flash</i>
DEFAULTS_NOT_RESTORED	<i>Defaults were not restored.</i>

15.3 NVM FUNCTIONS

FUNCTION ⇒ **NVM_block_format(3)** *Format an NVM user block*

This function formats an NVM user block.

The *Fill type* parameter allows you to choose how the block will be formatted. If *Fill type* is zero, the block is read and rewritten. (This is useful to fix checksum errors.) If *Fill type* is non-zero, the block is filled with your choice of fill character. Be aware that if you write to NVM1 or NVM2 the block will be automatically rewritten.

Writes typically take 6ms to complete.

Parameters

<i>NVM#</i>	Destination block ID (NVM#).
<i>Fill type</i>	Type of fill: 0= read & rewrite, 1= fill with <i>Fill character</i> .
<i>Fill character</i>	Fill character (only used when <i>Fill type</i> =1).

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ NVM_block_read(4) *Read an NVM user block*

This function reads from one to fifteen consecutive values from an NVM user block into local storage (as user# variables).

Parameters

<i>NVM#</i>	Source block ID (NVM1 to NVM40).
<i>Offset</i>	Offset from start of block.
<i>Length</i>	Length of block (number of consecutive variables).
<i>@Destination</i>	Initial destination variable (e.g., user6; if <i>Length</i> is 4, the destination would be user6 through user9).

Returns: 1, 0.

Reported Errors: BAD_ID, PARAM_RANGE, PT_RANGE.

FUNCTION ⇒ NVM_block_write(4) *Write an NVM user block*

This function writes from one to fifteen variables from local storage (these can be any user variables) into one of the 40 NVM user blocks. Be aware that if you write to NVM1 or NVM2 the block will be automatically rewritten.

Writes typically take 6ms to complete.

Parameters

<i>NVM#</i>	Destination block ID (NVM1 to NVM40).
<i>Offset</i>	Offset from start of block.
<i>Length</i>	Length of block (number of consecutive variables).
<i>@Source</i>	Initial source variable (e.g., user7; if <i>Length</i> is 8, the source would be user7 through user14).

Returns: 1, 0.

Reported Errors: BAD_ID, PARAM_RANGE, PT_RANGE.

FUNCTION ⇒ NVM_nvuser_restore(0) *Restore NVuser block*

This function restores the NVuser variables from either NVM1 (the power-fail-save block) or NVM2 (the periodic-save block).

The variable NVM_restore_source tells you which block was used: NVM1, NVM2, or no restore possible (see page 44).

Parameters: none.

Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ set_NVM_save_rate(1) *Set save rate for NVM2 block*

This function allows you to adjust the rate (in 256ms ticks) at which the periodic-save block is saved. The default save rate value is 1406 which translates to a save every 6 minutes ($1406 * 0.256\text{sec} = 360\text{sec} = 6\text{min}$).

Setting the save rate to zero turns off saving to the periodic-save block.

Note: The current nonvolatile storage element has a specified maximum of 1,000,000 write cycles. So, with the current value, you can expect the nonvolatile storage element to last about 100,000 hours.

$$6\text{min} * 1,000,000 \text{ write-cycles} = 6,000,000\text{min.}$$

$$6,000,000\text{min} / 60\text{min/hr} = 100,000\text{hr.}$$

Parameters

Rate Save rate in 256ms ticks; default = 1406 = 359.936sec.

Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ NVM_write_parameter(1) *Write a persistent parameter to nonvolatile memory*

This function is the only one in the NVM group that does not apply to the 40 NVM user blocks.

This function writes the value of any persistent parameter to nonvolatile memory. Be aware, writes typically take 6ms to complete. You should monitor NVM_status to determine when the operation finishes, or check the function return value to make sure the write was accepted.

Parameters

Parameter Name of persistent parameter to write.

Returns: 1, 0.

Reported Errors: PARAM_RANGE.

E•16

SYS
SYSTEM-LEVEL GROUP

16.1 USING THE SYSTEM-LEVEL FUNCTIONS

This group holds “housekeeping” data and functions, mainly concerned with data management when the system starts up and shuts down. Also included here are data and functions related to fault logging.

16.2 SYS DATA

VARIABLE ⇒ **source_of_system_block** *Source of nonvolatile system block*

When the system starts up, it automatically restores the system block from nonvolatile memory. (The system block holds information about the BDI, timers, master timer for fault log, fault log index, etc.) The system block is saved on power failure to one dedicated block of nonvolatile memory, and it is periodically saved to a second dedicated block. We prefer to recover the information from the power-fail-save block, as it is the most current; however, sometimes this block was not properly saved. In that case, we try to recover from the periodic-save block. If this also fails, we revert to the default values.

The `source_of_system_block` variable indicates the source of the recovered values, using these codes:

<code>SYS_RESTORED_PFB</code>	<i>Restored from power-fail-save block</i>
<code>SYS_RESTORED_PER</code>	<i>Restored from periodic-save block</i>
<code>SYS_RESTORED_NOT</code>	<i>Unable to restore from either block.</i>

VARIABLE GROUP ⇒ **See List** *Extended fault logging*

These variables are updated when you view the fault log.

Extended_log_fault_code	<i>Fault code</i>
Extended_log_time_stamp	<i>Time (MTM) at which the fault occurred</i>
Extended_log_value_1	<i>Value of optional variable 1 at time of fault</i>
Extended_log_value_2	" " " " 2 " " " "
Extended_log_value_3	" " " " 3 " " " "
Extended_log_value_4	" " " " 4 " " " "
Extended_log_value_5	" " " " 5 " " " "
Extended_log_map_length	<i>Number of optional variables defined (up to 5)</i>
Extended_log_1_map_object_ID	<i>CAN object ID of optional variable 1</i>
Extended_log_2_map_object_ID	" " " " " " 2
Extended_log_3_map_object_ID	" " " " " " 3
Extended_log_4_map_object_ID	" " " " " " 4
Extended_log_5_map_object_ID	" " " " " " 5

Extended_log_write_index	<i>ID of last block written</i>
Extended_log_read_index	<i>ID of last block read.</i>

16.3 SYS FUNCTIONS

FUNCTION ⇒ **clear_diaghist(0)** *Clear fault history*

This function clears the entire fault history file.

Parameters: none.

Returns: none.

Reported Errors: none.

FUNCTION ⇒ **get_fault_code(1)** *Get fault code*

This function allow you to recover fault codes from the fault stack by passing the *Index* to the fault stack and getting back the fault code. The fault codes are listed toward the end of the SysInfo.

The size of the fault stack ranges from zero to the number of faults currently active.

To use this function, start with an *Index* of zero. If a value of zero is returned, there are no faults. If a value other than zero is returned, report the fault and increment the index. Repeat this sequence until a value of zero is returned.

Parameters

Index Index of fault code.

Returns

Fault code Selected fault code, if *Index* is in range.

MAX Maximum size of fault stack, if *Index* = -1.

0 *Index* is out of range.

Reported Errors: none.

FUNCTION ⇒ **reset_controller(0)** *Reset hardware*

This function performs a software reset.

Parameters: none.

Returns: none.

Reported Errors: none.

FUNCTION ⇒ **set_sys_save_rate(1)** *Set save rate for system block*

This function allows you to adjust the rate (in 256ms ticks) at which the system block is saved. The default save rate value is 1406, which translates to a save every 6 minutes ($1406 * 0.256\text{sec} = 360\text{sec} = 6\text{min}$).

There is no way to turn off system saves. However, you may install a very large number; the largest number is -32768, which will increase the interval between saves to 4.6 hours.

Warning! The current nonvolatile storage element has a specified maximum of 1,000,000 write cycles. So, with the current value, you can expect the nonvolatile storage element to last about 100,000 hours.

$$6 \text{ min} * 1,000,000 \text{ write-cycles} = 6,000,000\text{min.}$$

$$6,000,000 \text{ min} / 60 \text{ min/hr} = 100,000\text{hr.}$$

Parameters

Rate Save rate in 256ms ticks; default = 1406 = 359.936 sec.

Returns: 1, 0.

Reported Errors: none.

E•17

VCL INTERPRETER

17.1 USING THE VCL FUNCTIONS

The variables under the control of the VCL interpreter do not share a common 3-letter identifier (i.e., they do not all include “VCL” in their names).

The functions, on the other hand, do include VCL in their names. They allow you to access and manipulate individual bytes within variables.

17.2 VCL DATA

VARIABLE GROUP ⇒

user# *User accessible variables*

This group of variables (user1 to user120) is available for you to use in your programs.

BIT VARIABLE GROUP ⇒

user_bit# *User accessible bit variables*

These variables (user_bit1 to user_bit10) are available for applications needing 1311 bit-level access.

VARIABLE GROUP ⇒

p_user# *Persistent user accessible variables*

The values of these variables (p_user1 to p_user100) are automatically restored from nonvolatile memory on power-up. This group of variables is primarily intended for use with the 1311 programmer. When these variables are altered by the 1311, the altered values are automatically saved in nonvolatile memory.

Note: Alterations of these values under VCL program control will **not** be automatically saved in nonvolatile memory. You must run the NVM_write_parameter() function to save VCL-set p_user values to nonvolatile memory.

If you wish to use the p_user# variables as normal variables (i.e., like the user# variables), you must take care to initialize the values on each power-up.

BIT VARIABLE GROUP ⇒

p_user_bit# *Persistent user accessible bit variables*

These variables (p_user_bit1 to p_user_bit10) are available for applications needing 1311 bit-level access.

VARIABLE ⇒

VCL_app_ver *VCL application version*

This variable shows you the version of the VCL application program.

VARIABLE GROUP ⇒ **See List** *VCL run rate*

These variables are available for applications.

VCL_ipms	<i>Number of VCL instructions per millisecond</i>
VCL_ipms_min	<i>Minimum number of VCL instructions per millisecond</i>

17.3 VCL FUNCTIONS

FUNCTION ⇒ **VCL_get_size(1)** *Return size of a variable*

This function returns the size (in bytes) of a variable.

Parameters

Variable Name of the variable.

Returns

Size Size of the variable, in bytes.

Reported Errors: none.

FUNCTION ⇒ **VCL_get_byte(2)** *Return one byte from a variable*

This function returns one byte from a variable.

Parameters

Variable Name of the variable.

Byte index Index to the byte (0 = 1st byte).

Returns

Byte Value of selected byte.

Reported Errors: PT_RANGE, PARAM_RANGE.

FUNCTION ⇒ **VCL_put_byte(3)** *Insert one byte into a variable*

This function inserts a byte into a variable.

Parameters

Variable Name of the variable.

Byte index Index to the byte (0–3).

New byte Value to be inserted.

Returns: 1, 0.

Reported Errors: PT_RANGE, PARAM_RANGE, PARAM_RO.

• F •

COMMUNICATIONS FUNCTIONS

It is often important to communicate what is going on in the software to the outside world of the operator. VCL includes serial port communications functions that allow data to be sent out over the controller's UART, packaged specifically for the Curtis 840 Display.

- 18. The Serial Port (**SER**) functions allow you
to configure your controller's serial port.54
- 19. The Spyglass (**SPY**) functions allow you
to setup and display messages on the Spyglass
(i.e., the Curtis 840 Display) when it is
connected to your controller's serial port.55

F•18

SER
SERIAL PORT CONTROL

18.1 USING THE SERIAL PORT FUNCTION

The serial port function allows you to explicitly configure the baud rate of the asynchronous serial port. This is rarely necessary because the system takes care of switching baud rates and transmission formats adaptively, depending on the device connected to the serial port (typically a Spyglass or a 1311 programmer).

18.2 SER DATA

CONSTANT GROUP ⇒

See list Baud rate selector

These constants are used to set the baud rate from within VCL. The default baud rate is 9600.

BR9600	<i>9600 baud</i>
BR19200	<i>19200 baud</i>
BR38400	<i>38400 baud</i>
BR57600	<i>57600 baud</i>
BR115200	<i>115200 baud</i>

1311 PARAMETER ⇒

Default Serial Port Baud Rate *Default serial port baud rate*

This parameter can be set using the 1311; the new baud rate does not take effect until the next power-on. The highest baud rate you are able to achieve will depend on the hardware you are using.

18.3 SER FUNCTION

FUNCTION ⇒

setup_serial(3) *Setup the serial port characteristics*

This function allows you to setup the serial port baud rate. If there was no error, the serial I/O buffers and the port are cleared before returning.

Parameters

<i>Baud rate</i>	Holds the baud rate ID (BR#; see above).
<i>Reserved</i>	(set to 0; previously was number of stop bits).
<i>Reserved</i>	(set to 0; previously was number of milliseconds between characters).

Returns: 1, 0.Reported Errors: PARAM_RANGE.

F•19

SPY

SPYGLASS SUPPORT

19.1 USING THE SPYGLASS FUNCTIONS

The functions in this group allow you to setup and display messages on the Spyglass (i.e., the Curtis 840 Display), when it is connected to the controller's serial port.

The `put_spy_message()` function supercedes five earlier functions:

```
put_spy_bin()
put_spy_dec()
put_spy_hex()
put_spy_mixed()
put_spy_text()
```

These functions are currently still accepted by VCL, but may not be available in subsequent versions.

Spyglass messaging is automatically disabled while a 1311 programmer is connected to the controller, as both devices use the same port.

19.2 SPY DATA

CONSTANT GROUP ⇒

See list *Communications Protocol Selector*

These IDs are used to select the appropriate protocol for the Spyglass output messages (i.e., displays).

`Protocol_TTY` omits the leading “:S”, LED codes, and checksum. There is a carriage return line feed sequence terminating these messages.

`Protocol_1` is the protocol for the original Spyglass: a leading “:S” with 8 ASCII characters followed by a single LED code followed by a checksum.

`Protocol_2` is the protocol for the new Spyglass; it is identical to `Protocol_1` except that it has two LED codes instead of one.

PROTOCOL_TTY *Raw format*

PROTOCOL_1 *Original Spyglass protocol*

PROTOCOL_2 *Generation 2 Spyglass protocol (this is the default, unless you run the `setup_SPY_protocol()` function)*

CONSTANT GROUP ⇒

See List *Spyglass put_spy_message() Nformat specifiers*

These constants are used to select the desired format when printing numbers using the put_spy_message() function.

PSM_DECIMAL	<i>Decimal, no decimal point</i>
PSM_DECIMAL_0	<i>Decimal, decimal point at far right</i>
PSM_DECIMAL_1	<i>Decimal, decimal point before 1st digit from right</i>
PSM_DECIMAL_2	<i>Decimal, decimal point before 2nd digit from right</i>
PSM_DECIMAL_3	<i>Decimal, decimal point before 3rd digit from right</i>
PSM_BINARY	<i>Binary</i>
PSM_HEX	<i>Hexadecimal</i>
PSM_TEX_ONLY	<i>Print only the leading text field</i>

19.3 SPY FUNCTIONS

FUNCTION ⇒

put_spy_LED(1) *Enter a new value for the Spyglass LED display*

This function installs a new value for the Spyglass's LED display.

Parameters

Value New value for the Spyglass LED display.

Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒

put_spy_message(4) *Display text and a number on the Spyglass*

This function installs a new message for the Spyglass with the pre-text on the left, the number in the middle, and the post-text on the right.

The number and post-text will be printed only if adequate space remains after the pre-text has been printed.

The number will be printed only if there is adequate space remaining after the post-text has been printed.

All fields are left justified (i.e., all truncation occurs on the furthest right characters).

The Nformat-specifier constants are listed above.

Parameters

<i>PreText</i>	String to display on the left.
<i>Number</i>	Number to display.
<i>PostText</i>	String to display on the right.
<i>Nformat</i>	Numerical format specifier (see page 56).

Returns

-1	Invalid string index.
0	Message did not print.
1	Message successfully printed.
2	Message printed with numeric value truncated.

Reported Errors: none.

FUNCTION ⇒ **put_spy_text_offset(1)** *Print text message with offset*

This function is used to print a previously entered message. *Offset* is the offset from the left-hand side of the string. For example, if you had previously defined TEXT string “123456789” and then issued put_spy_text(TEXT) you would see “12345678” on the Spyglass. Now issue put_spy_text_offset(1) and you will see “23456789”.

This function can be used to create rolling marquee banner displays.

The return value indicates whether the message was sent. If a message is currently being sent (i.e., the transmit-buffer still contains characters from the last message), this function will return a 0 to indicate failure; otherwise it will return 1 to affirm that the message was sent.

Parameters

<i>Offset</i>	Number of characters the previously entered text will be offset from the left-hand edge of the display.
---------------	---

Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **put_spy_timer(1)** *Display the value of a timer on the Spyglass*

This function displays the value of the selected timer on the Spyglass.

The timer is displayed in standard format: HHHHHH.T where H=hours and T=tenths of hours. The display is left-justified with a leading zero.

The return value indicates whether the message was sent. If a message is currently being sent (i.e., the transmit-buffer still contains characters from the last message), this function will return a 0 to indicate failure; otherwise it will return 1 to affirm that the message was sent.

Parameters

TMR# ID of the timer to display (see page 34).

Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **setup_spy_protocol(1)** *Setup serial communication protocol*

This function allows you to setup the Spyglass communication protocol; the protocols are listed on page 55.

Parameters

Protocol ID ID of the protocol to use with the Spyglass (e.g., PROTOCOL_2).

Returns: 1, 0.

Reported Errors: PARAM_RANGE.

19.4 SUPERCEDED SPY FUNCTIONS

Five of the original SPY functions have been superceded by the multipurpose new function put_spy_message(). The superceded functions are currently still accepted by VCL, but may not be available in subsequent VCL versions.

The return value indicates whether the message was sent. If a message is currently being sent (i.e., the transmit buffer still contains characters from the last message), then this function will return a 0 to indicate failure; otherwise it will return 1 to affirm that the message was sent.

SUPERCEDED FUNCTION ⇒

put_spy_bin(1) *Enter a new binary message for the Spyglass*

This function converts the parameter *Number* to binary and displays it on the Spyglass with the least significant bit on the far left.

Parameters

Number Number to display.

Returns: 1, 0.

Reported Errors: none.

SUPERCEDED FUNCTION ⇒

put_spy_dec(3) *Enter a new decimal message for the Spyglass*

This function converts the parameter *Number* to decimal and displays it on the Spyglass with both prefix and postfix characters. The display will be LnnnnnnT where L=*Prefix* (leading character), nnnnnn=*Number*, and T=*Postfix* (trailing character).

Alias: put_spy_numb()

Parameters

Prefix Leading character to display.

Number Number to display.

Postfix Trailing character to display.

Returns: 1, 0.

Reported Errors: none.

SUPERCEDED FUNCTION ⇒

put_spy_hex(3) *Enter a new hex message for the Spyglass*

This function converts the parameter *Number* to hexadecimal and displays it on the Spyglass with both prefix and postfix characters. The display will be LnnnnnnT where L=*Prefix* (leading character), nnnnnn=*Number*, and T=*Postfix* (trailing character).

Parameters

Prefix Leading character to display.

Number Number to display.

Postfix Trailing character to display.

Returns: 1, 0.

Reported Errors: none.

SUPERCEDED FUNCTION ⇒ **put_spy_mixed(3)** *Display text and a number on the Spyglass*

This function installs a new message for the Spyglass with text on the left and numeric information on the right. The third parameter allows you to scale the number if you wish.

The format is (T.....N), where T=*Text* and N=*Number*. The numeric field is truncated if more room is needed for the text display. The *Scaling* parameter is a signed 16-bit number (-32768 to 32767). The *Number* parameter is multiplied by this number and the product is then divided by 32767. So, the final value = $Number * (Scaling / 32767)$.

Setting *Scaling* to 0 turns off the scaling function.

Alias: put_spy_text_num()

Parameters

<i>Text</i>	String to display.
<i>Number</i>	Number to display.
<i>Scaling</i>	Scaling factor for the number to display.

Returns: 1, 0.

Reported Errors: none.

SUPERCEDED FUNCTION ⇒ **put_spy_text(1)** *Enter a new text message for the Spyglass*

This function installs a new text message for the Spyglass to output. The text will be left-justified within the display.

You can install a message directly by using the form put_spy_text("Message") or you can define a message using the string directive and then use the string ID (e.g., glop string "message", put_spy_text(glop)).

Parameters

<i>Text</i>	String to display.
-------------	--------------------

Returns: 1, 0.

Reported Errors: none.

• G •

CONTROLLER AREA NETWORK (CAN) FUNCTIONS

CAN is a widely used industrial and automotive communication protocol that allows multiple modules to communicate to each other on the same two signals. The CAN network interface allows you to set the characteristics of CAN system devices, assess their status, and conduct data transfers. You can do this with CANopen (included with your controller), with VCL CAN, or with a combination of the two.

We strongly recommend that you use CANopen as your CAN network interface.

- 20. **CANopen** provides a highly efficient,
ready-made CAN network interface.62
- 21. **VCL CAN** allows you to build your own
customized CAN network interface.70
- 22. It is possible to **combine CANopen & VCL CAN**
in order to add additional mailboxes beyond those
provided by the CANopen program.90

CAN acronyms:

CAN	Controller Area Network
COB	Communication Object
MISO	Master In, Slave Out
MOSI	Master Out, Slave In
NMT	Network Management Transmission
PDO	Process Data Object
SDO	Service Data Object

E•20

CANOPEN

20.1 USING CANOPEN

CANopen provides a ready-made library of CAN constants and variables, allowing you to interface the devices in your CAN network. With these constants and variables, you can set the characteristics of CAN system devices, assess their status, and setup data transfers.

Note: On a very few specific controller models, CANopen has been disabled by default. If your controller is one of these, you will need to enable CANopen; see `suppress_CANopen_init`, page 65.

For more information on the CANopen interface, see Curtis's OEM Technical Reference, *Generic CANopen Implementation*.

20.2 CANOPEN DATA

CONSTANT GROUP ⇒

See List CANopen mailbox IDs

These constants define the CANopen mailbox assignments.

CANOPEN_EMERGENCY_MAILBOX
CANOPEN_NMT_MAILBOX
CANOPEN_HEARTBEAT_MAILBOX
CANOPEN_SDO_MISO_MAILBOX
CANOPEN_SDO_MOSI_MAILBOX
CANOPEN_PDO_MISO_1_MAILBOX
CANOPEN_PDO_MOSI_1_MAILBOX
CANOPEN_PDO_MISO_2_MAILBOX
CANOPEN_PDO_MOSI_2_MAILBOX

CONSTANT GROUP ⇒

See List CANopen mailbox masks

These constants are used with the `CAN_msg_rcvd`, `CAN_msg_sent`, and `CAN_response_error` variables (see page 81) to isolate a selected object. For example, you could use a mask to find out whether an NMT has been received.

CANOPEN_EMERGENCY_MASK
CANOPEN_NMT_MASK
CANOPEN_HEARTBEAT_MASK
CANOPEN_SDO_MISO_MASK
CANOPEN_SDO_MOSI_MASK
CANOPEN_PDO_MISO_1_MASK
CANOPEN_PDO_MOSI_1_MASK
CANOPEN_PDO_MISO_2_MASK
CANOPEN_PDO_MOSI_2_MASK

Example: if (CAN_msg_rcvd & CANOPEN_NMT_MASK) <>0, the NMT was received.

VARIABLE GROUP ⇒

See List *CANopen PDO MISO # specification and mapping variables*

These variables specify and map PDO MISO 1 and PDO MISO 2 objects.

	CAN_pdo_miso_#_map_errors	<i>Bit array indicating mapping errors</i>
structure	CAN_pdo_miso_#_struct_length	<i>Number of PDO MISO structure variables (= 1)</i>
	CAN_pdo_miso_#_cob_ID	<i>PDO MISO COB ID</i>
	CAN_pdo_miso_#_length	<i>PDO MISO byte length (max=8)</i>
	CAN_pdo_miso_#_map_1	<i>PDO object ID 1</i>
	CAN_pdo_miso_#_map_2	<i>" " " 2</i>
	CAN_pdo_miso_#_map_3	<i>" " " 3</i>
	CAN_pdo_miso_#_map_4	<i>" " " 4</i>
	CAN_pdo_miso_#_map_5	<i>" " " 5</i>
	CAN_pdo_miso_#_map_6	<i>" " " 6</i>
	CAN_pdo_miso_#_map_7	<i>" " " 7</i>
	CAN_pdo_miso_#_map_8	<i>" " " 8</i>

VARIABLE GROUP ⇒

See List *CANopen PDO MOSI # specification and mapping variables*

These variables specify and map PDO MOSI 1 and PDO MOSI 2 objects.

	CAN_pdo_mosi_#_map_errors	<i>Bit array indicating mapping errors</i>
structure	CAN_pdo_mosi_#_struct_length	<i>Number of PDO MOSI structure variables (= 1)</i>
	CAN_pdo_mosi_#_cob_ID	<i>PDO MOSI COB ID</i>
	CAN_pdo_mosi_#_length	<i>PDO MOSI byte length (max=8)</i>
	CAN_pdo_mosi_#_map_1	<i>PDO object ID 1</i>
	CAN_pdo_mosi_#_map_2	<i>" " " 2</i>
	CAN_pdo_mosi_#_map_3	<i>" " " 3</i>
	CAN_pdo_mosi_#_map_4	<i>" " " 4</i>
	CAN_pdo_mosi_#_map_5	<i>" " " 5</i>
	CAN_pdo_mosi_#_map_6	<i>" " " 6</i>
	CAN_pdo_mosi_#_map_7	<i>" " " 7</i>
	CAN_pdo_mosi_#_map_8	<i>" " " 8</i>

VARIABLE GROUP ⇒ **See List** *CANopen general status variables*

These read-only variables indicate message/process timing.

CANopen_system_status_flags *Returns the following flags:*

CANOPEN_NMT_ENABLED
 CANOPEN_SDO_ENABLED
 CANOPEN_PDO_ENABLED
 CANOPEN_HEARTBEAT_ENABLED
 CANOPEN_EMERGENCY_ENABLED
 CANOPEN_ENABLED

For example, if (CANopen_system_status_flags & CANOPEN_SDO_ENABLED) <>0, the SDOs are enabled.

CANopen_bus_cycle_time *CANopen processing rate (in ms)*

CAN_pdo_response_period *Max response time to PDO (in CANopen_bus_cycle_time units)*

CAN_nmt_state *0=initialization, 4=stopped, 5=operational, 127=pre-operational*

VARIABLE GROUP ⇒ **See List** *CANopen general control variables*

The rates and periods in this group are in CANopen_bus_cycle_time increments; the value of CANopen_bus_cycle_time is in milliseconds. Thus if CANopen_bus_cycle_time is set at 4 ms and CANopen_heart_beat_rate is set at 5, heartbeats will be produced every 20 ms.

CANopen_heart_beat_rate *Sets heartbeat rate*

CANopen_emergency_rate *Sets emergency rate*

CAN_pdo_timeout_period *Sets PDO timeout period*

VARIABLE GROUP ⇒ **See List** *CANopen SDO block transfer variables*

The variables in this group are used to initiate and control SDO block transfers.

start_CAN_flash_download *Initiates a flash download (OS or APP)*

start_CAN_flash_upload_os *Initiates a flash OS upload*

start_CAN_flash_upload_app *Initiates a flash APP upload*

VARIABLE GROUP ⇒ **See List** *CANopen EE write status and control variables*

These variables are related to the interaction between CANopen objects and nonvolatile memory.

CANopen_ee_status_flags *Indicates the status of the EE write, using the following flags:*

CANOPEN_EE_WRITE_IN_PROGRESS

CANOPEN_EE_WRITE_OVERFLOW

CAN_ee_writes_enabled *To enable automatic EE write of a CANopen object, set this variable to a non-zero value*

VARIABLE ⇒ **suppress_CANopen_init** *CANopen suppression variable*

To disable CANopen initialization, set this variable to:

SUPPRESS_CANOPEN_KEY

If your controller has CANopen disabled by default and you want to run CANopen, you will have to change the setting of this variable. The most direct way to do this is by including the following parameter block definition in your VCL application:

```
; parameter_entry    "Suppress CANopen Init"
;   type              no_display
;   address            suppress_canopen_init
;   default            0
; end
```

When the VCL is built and downloaded, this will permanently change the nonvolatile memory's default value from CANopen disabled to CANopen enabled.

20.3 VCL FUNCTIONS APPLICABLE TO CANOPEN

Although CANopen runs independent of VCL, you may want to use VCL to limit its scope of operation. You can execute the VCL functions `disable_enable_CANopen()`—or individually execute the disable/enable functions for individual CANopen components (emergency, heartbeat, nmt, pdo, sdo).

FUNCTION ⇒ **enable_CANopen(0)** *Enable all CANopen processing*

This function enables all CANopen processing.

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen(0)** *Disable all CANopen processing*

This function disables all CANopen processing.

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **enable_CANopen_emergency(0)** *Enable CANopen emergency processing*

This function enables CANopen emergency processing by setting up the following mailbox:

CANOPEN_EMERGENCY_MAILBOX.

Parameters: none.

Normal Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen_emergency(0)** *Disable CANopen emergency mailbox*

This function disables CANopen emergency mailbox processing by releasing the following mailbox:

CANOPEN_EMERGENCY_MAILBOX.

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **enable_CANopen_heartbeat(0)** *Enable CANopen heartbeat processing*

This function enables CANopen heartbeat processing:

CANOPEN_HEARTBEAT_MAILBOX.

Parameters: none.

Normal Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen_heartbeat(0)** *Disable CANopen standard heartbeat mailbox*

This function releases the following mailbox from CANopen heartbeat processing:

CANOPEN_HEARTBEAT_MAILBOX.

Parameters: none.

Normal Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **enable_CANopen_nmt(0)** *Enable CANopen NMT processing*

This function enables CANopen NMT processing in the following mailbox:

CANOPEN_NMT_MAILBOX.

Parameters: none.

Normal Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen_nmt(0)** *Disable CANopen standard NMT mailbox*

This function releases the following mailbox from CANopen NMT processing:

CANOPEN_NMT_MAILBOX.

Parameters: none.

Normal Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **enable_CANopen_pdo(0)** *Setup CANopen PDO processing*

This function sets up the standard CANopen PDO processing in these four mailboxes:

CANOPEN_PDO_MISO_1_MAILBOX
CANOPEN_PDO_MOSI_1_MAILBOX
CANOPEN_PDO_MISO_2_MAILBOX
CANOPEN_PDO_MOSI_2_MAILBOX.

Parameters: none.

Normal Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen_pdo(0)** *Disable all CANopen standard PDO mailboxes*

This function releases these four mailboxes from CANopen PDO processing:

CANOPEN_PDO_MISO_1_MAILBOX
CANOPEN_PDO_MOSI_1_MAILBOX
CANOPEN_PDO_MISO_2_MAILBOX
CANOPEN_PDO_MOSI_2_MAILBOX.

Parameters: none.

Normal Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **enable_CANopen_sdo(0)** *Enable CANopen SDO processing*

This function enables CANopen SDO processing, by allocating these two mailboxes to CANopen SDO processing:

CANOPEN_SDO_MISO_MAILBOX
CANOPEN_SDO_MOSI_MAILBOX.

Parameters: none.

Normal Returns: 1, 0.

Reported Errors: none.

FUNCTION ⇒ **disable_CANopen_sdo(0)** *Disable CANopen standard SDO mailboxes*

This function releases these two mailboxes from CANopen SDO processing:

CANOPEN_SDO_MISO_MAILBOX
CANOPEN_SDO_MOSI_MAILBOX.

Parameters: none.

Normal Returns: 1.

Reported Errors: none.

E•21

VCL CAN

21.1 USING VCL CAN

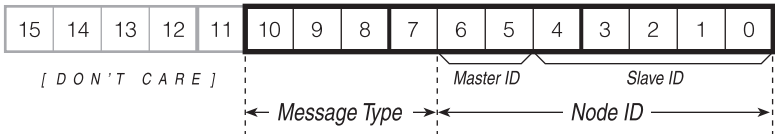
As an alternative to the ready-made CANopen interface, you can create your own customized CAN interface using the constants, variables, and functions of VCL CAN. (We assume CANopen is disabled (see `suppress_CANopen_init`, page 65.)

Section 21.1 is in four parts: (a) COB-ID structure, (b) a brief introduction to the key VCL CAN variables, (c) how to use the core VCL CAN functions, and (d) a CAN system example.

COB-ID Structure

Each message has an 11-bit message identifier, called a COB-ID (Communication Object ID). The most common COB-ID structure consists of three fields. The upper four bits hold the message type. The next two bits hold the ID of the master device. The last five bits hold the ID of the slave device.

11-bit message identifier field:



Alternative allocation and assignment of ID bits is possible, but you may require Curtis support and modification if your node IDs are to be accessible or programmable using a Curtis 1311 programmer where maximum values must be defined.

Using VCL CAN Variables

These two VCL CAN variables provide status information:

<code>CAN_bus_on</code>	CAN Bus Status (0=off, non-0=on)
<code>CAN_error_status</code>	Bit0 = Tx or Rx error count > 96), Bit1 = CAN bus Off.

These three VCL CAN variables provide information about the status of a given mailbox; examples of bit definitions are given in parentheses:

<code>CAN_response_error</code>	Response timeout flags (bit= <code>CAN1_TIMEOUT</code>).
<code>CAN_msg_sent</code>	Sent flags (bit= <code>CAN1_SENT</code>).
<code>CAN_msg_rcvd</code>	Received flags (bit= <code>CAN1_RECEIVED</code>).

The `CAN_response_error` variable holds fourteen bit-addressable timeout flags (`CAN1_TIMEOUT` through `CAN14_TIMEOUT`) that are set to 1 when a CAN response is not received within the response timeout period set by the `setup_mailbox()` function. For example, assume you setup a mailbox with the following parameters:

```
Setup_MailBox(CAN1, PDO_MOSI, MasterID, SlaveID, C_CYCLIC,
C_XMT, 8, CAN2)
```

This mailbox will be sent every 20ms (assuming the cyclic clock remains set to its default value). When the message is sent, a timer is started. If a response message doesn't arrive in the CAN2 mailbox within 32ms, a `CAN1_TIMEOUT` flag will be set.

Similarly, the `CAN_msg_sent` variable holds fourteen bit-addressable flags (`CAN1_SENT` through `CAN14_SENT`) that are set each time the corresponding mailbox is sent. These flags are very useful if you want to send messages manually (i.e., using the `send_mailbox()` function) as rapidly as possible. There are no queues or stacks for CAN buffers; consequently, you must either wait at least 4ms (the service time for the CAN subsystem) or wait on the `CAN#_SENT` flag for that mailbox to guarantee that you don't overwrite a message before it's sent.

There are two warnings about using the `CAN_msg_sent` flags. First, these flags are only set by the system, never cleared; you must therefore remember to clear the flag before you use it. For example, you might write:

```
CAN3_SENT = OFF
send_mailbox(CAN3)
while (CAN3_SENT == OFF) {}
```

This clears the message-sent flag, tells the system that you want to send the contents of mailbox 3, and then waits until the system sets the flag saying that mailbox 3 has actually been sent.

A second warning concerning the use the message sent flags is that many CAN nodes may only be able to process messages at a limited rate. For example, the 1243 CAN controller will only process CAN messages every 20ms. In other words, sending messages more frequently than every 20ms to a 1243 CAN controller will result in lost messages.

The `CAN_msg_rcvd` variable also holds a group of twelve flags, indicating that a message has been received in the specified mailbox. As with the `CAN_msg_sent` flags, `CAN_msg_rcvd` flags are only set by the system, never cleared.

Using VCL CAN Functions

• setup_CAN(5)

The first VCL CAN command that should be issued is the setup_CAN() function. It initializes the CAN system and allows you to tailor its operation according to the following parameters:

<i>Baud</i>	Holds the index baud_rate (CAN_125KBAUD, CAN_250KBAUD, CAN_500KBAUD).
<i>Sync</i>	Holds the sync period (0=off, n=number of 4ms ticks between syncs).
<i>Reserved</i>	Not currently used (set to 0).
<i>Reserved</i>	Not currently used (set to 0).
<i>Restart</i>	CAN bus auto-restart after bus-off (0=yes, !0=no).

The *Baud* rate must, of course, be the same for all elements on the CAN bus.

The *Sync* period refers to one of two periodic rates available to schedule mailboxes to be sent automatically (the other rate is the cyclic period). The default rate for the sync period is the same as for the cyclic period: 20ms (i.e., *Sync*=5).

The *Restart* parameter determines what the CAN subsystem does when it encounters bus errors. If set to 0, the CAN subsystem will automatically try to restart the CAN bus in the event of a bus error. If set to non-zero, the CAN subsystem will not try to restart the bus (i.e., the CAN bus will become inactive if an error is encountered) and will set a flag indicating a CAN bus error.

• setup_mailbox(8)

This function sets up the initial parameters for a mailbox. **This function must be called before defining the data for the mailbox** (see setup_mailbox_data) or trying to use the mailbox. This function accepts the following parameters:

<i>CAN#</i>	Mailbox block ID (e.g., CAN3).
<i>Message Type</i>	Type of message (e.g., PDO_MOSI) goes into address bits 7-10.
<i>Master ID</i>	Master ID goes into address bits 5-6.
<i>Slave ID</i>	Slave ID goes into address bits 0-4.
<i>Service</i>	When to send this message (C_EVENT, C_CYCLIC, or C_SYNC).
<i>Direction</i>	Data direction (C_RCV or C_XMT).
<i>Timeout</i>	Replay timeout in 4ms increments (e.g., 5 = 20ms timeout).
<i>Reply</i>	Handle of reply buffer (e.g., CAN2).

CAN# refers to the predefined ID of the mailbox you wish to setup. There are presently fourteen mailboxes (CAN1 through CAN14) available. All mailboxes behave identically.

The next three parameters—*Message Type*, *Master*, and *Slave*—are combined to make a single 11-bit identifier, as shown on page 70.

The 4-bit *Message Type* field determines both how the message will be processed and its priority. The message types are listed below in priority order:

NMT	Network Management Transmission			
SYNC_ERR	Both Sync (ID=0) & Error (ID = Error) messages			
PDO_MISO	Process Data Object (PDO), Master In Slave Out			
PDO_MOSI	"	"	"	Master Out Slave In
PDO_SYS1	"	"	"	System Broadcast 1
PDO_SYS2	"	"	"	System Broadcast 2
PDO_SYS3	"	"	"	System Broadcast 3
PDO_SYS4	"	"	"	System Broadcast 4
SDO_MISO	Service Data Object (SDO), Master In Slave Out			
SDO_MOSI	"	"	"	Master Out Slave In
NODE	Node Guard Message			

As you can see there are two groups of types (PDO and SDO) plus three unique message types (NMT, SYNC_ERR, and NODE). In the most general sense, the types simply define the priority level with NMT being the highest, and NODE being the lowest. However, according to common agreement, each group and each unique type has a specific use.

In many of these messages, you see MOSI and MISO appended to the basic message type (e.g., PDO_MOSI). These tags reinforce the intended flow of data on the CAN bus. The master transmits a PDO_MOSI (Master Out, Slave In), while the slave receives the PDO_MOSI. The slave may then respond with a PDO_MISO.

NMT: The NMT message type is used to control bus activity and initiate certain actions within nodes connected to the bus. For example, you would use an NMT message to bring a Curtis controller's CAN system online.

SYNC_ERR: This type is used for two purposes. Receipt of a SYNC message should cause you to send sync information (e.g., information in response to a master clock/requestor). ERROR messages are used to broadcast the advent of an error within a controller to other bus elements.

PDO: A **P**rocess **D**ata **O**bject is a collection of CAN objects; usually these are parameters that reflect the changing state of the machine. Because of

the rigorous error checking in a CAN message, the data portion of the message is usually 50% or less. PDOs allow up to 8 CAN objects to be sent all at once. The destination of the data (the index or address) for each element of a PDO is pre-defined at design time; consequently, all that must be sent is the data. It is often the case that receipt of a PDO carries the expectation that a PDO will be sent in response. For example, the default configuration will cause a PDO_MISO and a PDO_SYS1 to be sent in response to each PDO_MOSI sent.

SDO: A **S**ervice **D**ata **O**bject has a format that is identical to that of a PDO; however, SDOs generally refer to parameters within the machine that do not change as the machine runs (i.e., system constants). Often, SDOs refer to data that is stored in nonvolatile memory.

NODE: The node type is reserved for automatic responses in response to a node-guarding message, usually sent out at the sync frequency. These messages are used to guarantee that nodes are still available.

The 2-bit Master ID field is the next part of the ID field. This identifies which of the four possible bus masters is generating the message.

The final part of the ID field is the Slave ID field, which identifies the intended recipient of the message.

VCL uses all three fields in the 11-bit identifier. For most cases, you will enter a value for each field: i.e., (. . . message type, master id, slave id, . . .). The Message Type field is shifted automatically into the top 4 bits. The Master ID field is shifted into a 2-bit field just below the Message Type, and the Slave ID field is placed in the lowest 5 bits. Both the Message Type and Master ID fields are masked before they are shifted into place; however, the Slave ID field is simply or'ed into the final identifier. This allows you to set both the Message Type and Master ID fields to 0 and then place a pre-built 11-bit identifier into the Slave ID field: i.e., (. . . 0, 0, cob-id, . . .). This is necessary to implement unique COB-IDs (communication object IDs) and can also be used to accommodate other protocol ID strategies.

The Service parameter is used by the system to determine when to send the data in the mailbox. There are three possibilities:

C_SYNC	Send on sync tick
C_CYCLIC	" " clock tick
C_EVENT	" " event

As mentioned previously, there are two periodic clocks in the CAN subsystem, the cyclic clock and the sync clock. These clocks can be set independently (the default for both is 20ms). Usually, periodic events are scheduled using the cyclic clock (C_CYCLIC), with the sync clock being reserved for node-guarding. Node-guarding refers to the practice of sending a message out to each node in

the system and then checking that the node responds (i.e., to see if a node has failed). This use of the two periodic clocks is the convention; however, it is not enforced by the VCL implementation. You can, if you wish, use the sync clock as a second cyclic clock.

The `C_EVENT` flag is used to tell the system that the data in the mailbox is to be sent on demand, using the `send_mailbox()` command.

The *Direction* parameter tells the system whether the data in this mailbox is to be transmitted (`C_XMT`) or received (`C_RCV`).

The *Timeout* and *Reply* parameters are used together; their interpretation depends on the direction flag (`C_XMT` or `C_RCV`).

If the data direction is *out* (`C_XMT`), you can tell the system to expect a reply within a certain period of time. Set the timeout parameter equal to the maximum time (in 4ms increments) that you're willing to wait for the reply. Then set the reply parameter to the mailbox ID (e.g., `CAN2`) in which you expect the reply to be received.

If, on the other hand, the data direction is *in* (`C_RCV`), you can set the reply field to the ID of a mailbox that is to be automatically transmitted in response to the message you've just received. In this case, set the timeout field to 0.

If neither of the two previous conditions are of interest to you (you don't demand a response and you don't automatically reply), then set the timeout parameter and reply parameter both to 0.

• **setup_mailbox_data(10)**

This function is used to define the data pointers for a given mailbox. Remember: The `setup_mailbox` function() must have been called before defining the data with this function. This function takes the following parameters:

<i>CAN#</i>	Holds the ID of the mailbox (e.g., <code>CAN4</code>).
<i>Count</i>	Holds the number of bytes actively used by this mailbox.
<i>@Data1</i>	Address of byte 1's value.
<i>@Data2</i>	Address of byte 2's value.
<i>@Data3</i>	Address of byte 3's value.
<i>@Data4</i>	Address of byte 4's value.
<i>@Data5</i>	Address of byte 5's value.
<i>@Data6</i>	Address of byte 6's value.
<i>@Data7</i>	Address of byte 7's value.
<i>@Data8</i>	Address of byte 8's value.

CAN# is the same predefined mailbox identifier that you used in the `setup_mailbox()` function call (e.g., `CAN2`)

The *Count* parameter tells the system how many bytes you will be sending/receiving with this mailbox. Depending on the format, the message length can be anything from 0 to 8 bytes of data.

Remember that all variables within VCL are 16-bits long; however, the mailbox data pointers refer to 8-bit values. If you just give the name of a variable (e.g., DLY1_output) you will be referring to the lower 8 bits of that variable. If you want to refer to the upper 8 bits, you must use the USEHB operator. To reference the upper 8 bits of DLY1_output, you would write @DLY1_output+USEHB.

Note: In most cases, you can safely omit the address operator “@” in your code; the program automatically inserts the operator where appropriate. However, you **must** type the address operator whenever an expression is involved (e.g., @DLY1_output+USEHB).

Because all VCL functions have fixed length parameter lists, you must always supply values for them all. By convention, any unused parameters should be set to 0.

- **startup_CAN(0)**

When the system first starts up, the CAN system is disabled. Disabled in this sense means that the subsystem does not send any messages, nor does it check if any messages have been received.

Generally, you will execute the setup_CAN() function first, followed by one or more pairs of setup_mailbox() and setup_mailbox_data() functions. Once you have setup all the mailboxes, you can startup the CAN system by using the startup_CAN() function.

- **startup_CAN_cyclic(0)**

Executing the startup_CAN() function allows the CAN subsystem to send messages on demand and begin automatically checking for messages received; however, it does not enable automatic (cyclic or sync) message transmissions. To do this, you must execute the startup_CAN_cyclic() function.

The reason that we don't start up both the CAN system and its cyclic functions has to do with the way we normally use the system. It is often the case that after you've setup the CAN system (setup_CAN()) and configured all your mailboxes (setup_mailbox() and setup_mailbox_data()) you will need to configure other elements of the CAN network before you begin full automatic operation. For example, you may have to send one or more NMT messages to startup CAN services on another device. Or you may have to send a series of SDOs to make sure that another device on the bus is properly configured. Initiating cyclic messaging at this point would, at best, delay these operations (because of the increased bus traffic).

Consequently, you will often see the `startup_CAN()` function call followed by one or more NMT messages (or even a call to an extensive set of SDO message invocations) before you see the `startup_CAN_cyclic()` function call.

• `send_mailbox(1)`

This function forces the mailbox contents to be sent. Its only parameter is the block ID of the mailbox you are sending (e.g., CAN4). You will need to use this function for any mailbox that has a service field set to `C_EVENT` (as opposed to `C_CYCLIC` or `C_SYNC`).

CAN System Example

This code shows how to setup the CAN buffers to send SDOs on demand. This requires five CAN-related function calls: `setup_CAN()`, `setup_mailbox()`, `setup_mailbox_data()`, `startup_CAN()`, and `send_mailbox()`. We also discuss the `setup_CAN_cyclic()` function.

The definitions section

In this section, we define all user variable synonyms and constants. In this case, we need two variables: one to hold the ID part of the SDO and one to hold the data part of the SDO. As you can see, these have been put into `user1` and `user2`.

We also need to define the Master and Slave IDs. This is done using constant definitions. Defining these values once, using a constant, makes it much easier to modify these values should the need arise.

```

;=====
; CAN System Demonstration
;=====
;
; -This code demonstrates how to setup the CAN system to send SDOs on
; demand.
;
;-----
; Local Definitions
;-----
;
; System Constants
;-----
;
; CAN Bus IDs
MASTER_ID    constant 2      ; ID of the master unit
SLAVE_ID     constant 5      ; ID of the slave unit
;
; PDO Variables
;-----
;
; These variables are used when sending and receiving SDOs.
sdo_id       equals user1    ; Holds the ID when sending SDOs
sdo_data     equals user2    ; Holds the data to be sent
;
sdo_in_id    equals user1    ; Holds the ID when receiving SDOs
sdo_in_data  equals user2    ; Holds the data received

```

The one-time setup section

This is the classic initialization section. This code is executed once, before we fall into the main loop.

```

;-----
; One Time Setup
;-----
; -In this section of code we do all one-time initializations.

call Startup_CAN_System      ; Start CAN System Messaging
user3 = 0                    ; User3 triggers sending the mailbox

```

The main loop

The main loop code is continuously executed. If user3's value becomes non-zero, mailbox 6 is sent (explicitly, using the send_mailbox() function).

```

;-----
; Main Program Loop
;-----
; -Each time user3 is set, the mailbox will be sent. You can examine
; the CAN_msg_sent variable to see that the bit corresponding to CAN6
; has been set.
;
main_loop:
    if (user3 <> 0)
    {
        user3 = 0
        send_mailbox(CAN6)
    }
goto main_loop

```

System support subroutines

There is only one support routine in this example: the CAN initialization subroutine that is called in the one-time setup section.

```

;-----
; Startup_CAN_System - Startup the CAN subsystem
;-----
; -This routine starts up the CAN system. We send out NMTs and SDOs
; here to initially configure the system.
;
Startup_CAN_System:

setup_can(CAN_125KBAUD,0,0,0,1) ;125kb,no sync,null,null,no auto-
restart

; Setup an SDO mailbox so we can Send Data
setup_mailbox(CAN6,SDO_MOSI,MASTER_ID,SLAVE_ID,C_EVENT,C_XMT,0,0)
setup_mailbox_data(CAN6,4,
@sdo_id+USEHB,
sdo_id,
@sdo_data+USEHB,
sdo_data,
0,0,0,0)

; Setup an SDO mailbox so we can Receive Data
setup_mailbox(CAN7,SDO_MISO,MASTER_ID,SLAVE_ID,C_EVENT,C_RCV,0,0)
setup_mailbox_data(CAN7,4,
@sdo_in_id+USEHB,
sdo_in_id,
@sdo_in_data+USEHB,
sdo_in_data,
0,0,0,0)

; Start CAN Messaging (non-cyclic only)
startup_can()

return

```

21.2 VCL CAN DATA

CONSTANT GROUP ⇒ **CAN#** *Block ID*

These constants identify the fourteen CAN mailboxes: CAN1 to CAN14. (The fifteenth mailbox is reserved as read-only: “CANRO”).

CONSTANT GROUP ⇒ **See List** *Message type specifier*

These constants are used in the process of setting up a mailbox; they are listed in priority order.

NMT	<i>Network Management</i>		
SYNC_ERR	<i>Both Sync (ID=0) and Error (ID=Error)</i>		
PDO_MISO	<i>Process Data Object</i>		<i>(Master In Slave Out)</i>
PDO_MOSI	”	”	<i>(Master Out Slave In)</i>
PDO_SYS1	”	”	<i>(System Broadcast 1)</i>
PDO_SYS2	”	”	<i>(System Broadcast 2)</i>
PDO_SYS3	”	”	<i>(System Broadcast 3)</i>
PDO_SYS4	”	”	<i>(System Broadcast 4)</i>
SDO_MISO	<i>Service Data Object</i>		<i>(Master In Slave Out)</i>
SDO_MOSI	”	”	<i>(Master Out Slave In)</i>
NODE	<i>Node Guard Message</i>		

CONSTANT GROUP ⇒ **See List** *Message execution schedule*

These constants are used to schedule transmission of messages.

C_SYNC	<i>Send on sync tick</i>		
C_CYCLIC	”	”	<i>clock tick</i>
C_EVENT	”	”	<i>event</i>

CONSTANT GROUP ⇒ **See List** *Transfer direction*

These values are used to set the direction of transfer (see `setup_mailbox()`).

C_RCV	<i>Receive bit</i>
C_XMT	<i>Transmit bit</i>

CONSTANT ⇒ **USEHB** *Use the high byte of a variable*

This is used in conjunction with the `setup_mailbox_data()` function to tell the system to use the high byte of a variable. When using `setup_mailbox_data()`, you must specify each byte of data to be transmitted. By default, the system will access the low byte of the word specified (each word is 16 bits). Using this constant, you can tell the system that you want to use the high byte instead; see page 87.

VARIABLE ⇒ **CAN_baud_rate** *Data transmission rate*

This variable sets the effective baud rate, using the following constants:

CAN_125KBAUD
CAN_250KBAUD
CAN_500KBAUD.

VARIABLE ⇒ **CAN_bus_on** *Bus status*

The CAN_bus_on variable has two values, 0 and 1. The value is zero until the startup_CAN() routine is run, at which point CAN_bus_on is set to 1.

If the CAN bus ever goes Off (i.e., CAN_error_status = 3), CAN_bus_on is reset to 0. CAN_bus_on will stay at 0 until the next time startup_CAN() is called. Because of this, if you leave the CAN Bus auto-restart enabled (see setup_CAN()) you can still use CAN_bus_on to tell whether the bus was ever forced off.

VARIABLE ⇒ **CAN_error_mailbox** *Bus error mailbox ID*

This variable provides the mailbox ID at the time the CAN bus error occurred. A value of -1 indicates “no mailbox”.

VARIABLE ⇒ **CAN_error_status** *Bus error status*

This variable is used to provide information about the number of errors and the current status of the CAN bus. The variable’s two bits have the following values:

- bit 0 If non-zero, there have been more than 96 transmit or receive errors.
- bit 1 If non-zero, the CAN bus has been automatically turned off.

This yields the following status values for CAN_error_status:

- 0 = No error.
- 1 = There have been more than 96 transmit or receive errors.
- 3 = The CAN bus is Off (there have been more than 256 errors).

VARIABLE ⇒ **CAN#_length** *Length of last message*

This variable holds the number of data bytes received in the indicated mailbox. For example, CAN4_length holds the number of data bytes received in the CAN4 mailbox.

BIT-VARIABLE ⇒ CAN_msg_rcd *Message received flags*

This variable is used to indicate that a message has been received in the specified mailbox. Each of these bits has a predefined bit constant consisting of the block ID plus the word “received”:

CAN#_RECEIVED

For example, the message-received bit definition for CAN5 is CAN5_RECEIVED.

Note: Flags are set—but not cleared—by the system. It is up to you to clear them.

BIT-VARIABLE ⇒ CAN_msg_sent *Message transmitted flags*

This variable is used to indicate that a message has been sent. CAN messages are sent out periodically rather than instantly; a message may be sent later than the time your program specifies the message is to be sent. This delay could be a problem when you need to send a number of messages in sequence; however, using this variable, you can “loop” until the Sent flag for the buffer of interest is set, thus avoiding data overruns.

Each of these bits has a predefined bit constant consisting of the block ID plus the word “SENT”:

CAN#_SENT

For example, the message-sent bit definition for CAN5 is CAN5_SENT.

Note: Flags are set—but not cleared—by the system. It is up to you to clear them before issuing new Send commands.

BIT-VARIABLE ⇒ CAN_response_error *Response timeout flags*

This variable is used to indicate that there was a message response timeout. When you setup a mailbox, you can also specify that you expect a response within a certain time period (see the `setup_mailbox()` function’s reply and timeout parameters). If the response is not received within the specified time period, the bit corresponding to the mailbox is set.

Each of these bits has a predefined bit constant consisting of the block ID plus the word “timeout”:

CAN#_TIMEOUT

For example, the timeout bit definition for CAN5 is CAN5_TIMEOUT.

Note: Flags are set—but not cleared—by the system. It is up to you to clear them.

CONSTANT ⇒ **CANRO** *Block ID*

The CAN Read-Only (RO) block is used to access a special read-only CAN mailbox. This CAN mailbox has a more sophisticated acceptance mask that can be set using the `setup_CANRO_mask()` function; see page 89.

VARIABLE ⇒ **CANRO_ID** *ID of last message received in CANRO*

This variable holds the actual ID received in the CANRO mailbox.

VARIABLE ⇒ **CANRO_length** *Length of last message received in CANRO*

This variable holds the number of data bytes received in the CANRO mailbox.

21.3 VCL CAN FUNCTIONS

FUNCTION ⇒ **CAN_last_error(0)** *Return the last error seen on the CAN bus*

This function returns the last error that occurred on the CAN bus.

Parameters: none.

Returns

- 0 No error.
- 1 Stuff error (more than 5 bits of the same polarity).
- 2 Form error (fixed part of message has wrong format).
- 3 Ack error (transmitted message not acknowledged).
- 4 Bit1 error (tried to send a 1 unsuccessfully).
- 5 Bit0 error (tried to send a 0 unsuccessfully).
- 6 CRC error (bad CRC check).

Reported Errors: none.

FUNCTION ⇒ **CAN_set_cyclic_rate(1)** *Set the period of cyclic servicing*

This function allows you to change the cyclic service period. When cyclic servicing has been started (with a call to the startup_CAN_cyclic() function), the default service period is 20ms.

A value of 0 (or less than 0) causes the system to reset to the default service period of 20ms.

The service period is set in 4ms increments: a value of 2 results in a service period of 8ms, and a value of 20 results in a period of 80ms.

Parameters

Rate Service rate in 4ms increments.

Returns

Rate The new service rate.

Reported Errors: none.

FUNCTION ⇒ **disable_mailbox(1)** *Disable a message buffer*

This function allows you to explicitly disable a mailbox. This is necessary when you want to redefine the mailbox using the `setup_mailbox()` function.

Note: Disabling a mailbox while redefining the data pointer (using the `setup_mailbox_data()` function) is not necessary.

Parameters

CAN# Block ID of the mailbox to disable.

Returns: 1, 0.

Reported Errors: BAD_ID.

FUNCTION ⇒ **enable_mailbox(1)** *Enable a message buffer*

This function allows you to explicitly re-enable a mailbox. This will only be necessary when you've explicitly disabled the mailbox using the `disable_mailbox()` function.

Note: The mailbox must have been setup using `setup_mailbox()` and its data pointers defined using `setup_mailbox_data()` before this function is called.

Parameters

CAN# Block ID of the mailbox to enable.

Returns: 1, 0.

Reported Errors: BAD_ID, MO_INACTIVE.

FUNCTION ⇒ **send_mailbox(1)** *Request that a mailbox be sent*

This function forces the mailbox contents to be sent. This is necessary for any mailboxes that have been setup with the `C_EVENT` constant. Note: The mailbox must have been setup and its data pointers defined (`setup_mailbox()` and `setup_mailbox_data()` functions called) before using this function.

Parameters

CAN# Block ID of the mailbox to send.

Returns: 1, 0.

Reported Errors: BAD_ID, MO_INACTIVE, MO_SENDING.

FUNCTION ⇒ **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

<i>Baud</i>	Baud rate (CAN_125KBAUD, CAN_250KBAUD, or CAN_500KBAUD).
<i>Sync</i>	Sync period (0=off, n=number of 4ms ticks between syncs).
<i>Reserved</i>	(set to 0)
<i>Slave ID</i>	This module's slave ID (set to -1 to ignore the slave ID in this function and use the current slave ID instead).
<i>Restart</i>	CAN bus auto-restart after bus Off (0=yes, !0=no).

Returns: 1, 0.

Reported Errors: BAD_BAUD.

FUNCTION ⇒ **setup_CAN_resync_width(1)** *Set the bit cell re-synchronization width*

This function allows you to set the maximum number of bit cells over which a re-synchronization can occur. The normal value is three.

Parameters

<i>Width</i>	Number of bit cells over which re-sync is possible (0 to 3).
--------------	--

Returns: 1, 0.

Reported Errors: PARAM_RANGE.

FUNCTION ⇒ **setup_CAN_sample_delay(1)** *Set the bit cell sample point*

This function allows you to change the bit cell sample point (the ratio between TSeg1 and TSeg2). The normal sample point is slightly delayed from the center of the bit cell. Under certain conditions (e.g., long or very capacitive transmission lines), it may be advisable to move this sampling point back in the bit cell.

The range of values for this function is from 0 to 5. Zero is the normal sample point: Sync=1, TSeg1=11, TSeg2=8. As the value is increased, the sample point is shifted back. At a value of 5 the sample points are: Sync=1, TSeg1=16, TSeg2=3.

Parameters

<i>Delay</i>	How far to delay the sampling. The range is from 0 (normal) to 5 (max).
--------------	---

Returns: 1, 0.

Reported Errors: PARAM_RANGE.

FUNCTION ⇒ **setup_mailbox(8)** *Setup parameters for a mailbox*

This function sets up the initial parameters for a mailbox. **This function must be called before defining the data for the mailbox** (see setup_mailbox_data()) **or trying to use the mailbox.**

Parameters

<i>CAN#</i>	Mailbox block ID (e.g., CAN3).
<i>Message Type</i>	Type of message (e.g., PDO_MOSI) that goes into address bits 7-10.
<i>Master ID</i>	Master ID that goes into address bits 5-6.
<i>Slave ID</i>	Slave ID that goes into address bits 0-4.
<i>Service</i>	When to send this message (C_EVENT, C_CYCLIC, or C_SYNC).
<i>Direction</i>	Data direction (C_RCV or C_XMT).
<i>Timeout</i>	Reply timeout in 4ms increments (e.g., 5=20ms timeout).
<i>Reply</i>	Block ID of reply mailbox (e.g., CAN2).

Returns: 1, 0.

Reported Errors: BAD_ID, BAD_MO_ID.

FUNCTION ⇒ **setup_mailbox_byte_select(9)** *Select the byte used for mailbox data*

This function allows you to set which byte will be used for each value defined for the mailbox. Run this function after executing setup_mailbox_data().

Parameters

<i>CAN#</i>	Mailbox block ID (e.g., CAN3).					
<i>Sel1</i>	Selector for byte 1's value (0-3).					
<i>Sel2</i>	"	"	"	2's	"	"
<i>Sel3</i>	"	"	"	3's	"	"
<i>Sel4</i>	"	"	"	4's	"	"
<i>Sel5</i>	"	"	"	5's	"	"
<i>Sel6</i>	"	"	"	6's	"	"
<i>Sel7</i>	"	"	"	7's	"	"
<i>Sel8</i>	"	"	"	8's	"	"

Returns: 1, 0.

Reported Errors: BAD_ID, MO_INACTIVE.

FUNCTION ⇒ **setup_mailbox_data(10)** *Define message buffer data*

This function is used to define the “data pointers” (i.e., addresses) for a given mailbox. Note: The setup_mailbox() function must have been called before defining the data with this function.

The constant USEHB is used with this function to specify when a word’s high bytes are to be used; see constant description on page 79.

You may have noticed that it is typically not necessary to prefix address parameters with the address operator (@); this is because the program automatically inserts the operator where appropriate. However, the program cannot do this if the parameter includes a calculation. **You must type the address operator with any parameter that includes a calculation (i.e., +USEHB).** For example, user27+USEHB will not work properly for the @Data6 parameter; you must type the “@”: @user27+USEHB.

Parameters

CAN#	Mailbox block ID (e.g., CAN4).
Count	The number of bytes actively used by this mailbox.
@Data1	Address of byte 1’s value.
@Data2	” ” ” 2’s ”
@Data3	” ” ” 3’s ”
@Data4	” ” ” 4’s ”
@Data5	” ” ” 5’s ”
@Data6	” ” ” 6’s ”
@Data7	” ” ” 7’s ”
@Data8	” ” ” 8’s ”

Returns: 1, 0.

Reported Errors: BAD_ID, BAD_MO_LEN, MO_INACTIVE, PARAM_RO.

FUNCTION ⇒ **shutdown_CAN(0)** *Shutdown the CAN system*

This function allows you to turn off the CAN system. You can use this function to stop the system while you reconfigure it. Note: You can redefine mailboxes without using this function as the mailbox is automatically disabled while the changes are taking place.

Parameters: none.

Returns: 1.

Reported Errors: none.



If you use shutdown_CAN() to shut down CANopen, you must use the function startup_CAN() to restart it.

FUNCTION ⇒ shutdown_CAN_cyclic(0) *Stop CAN cyclic messages*

This function allows you to turn off all currently active cyclic messages. Note: You can redefine mailboxes without using this function as the mailboxes are automatically disabled while the changes are taking place.

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ startup_CAN(0) *Startup the CAN system*

The CAN system does not start automatically. You must use this function to start up CAN services. This allows you to reset the system to different baud and sync rates as well as a chance to setup the mailboxes before the system starts.

Notice that to start cyclic messages, you must use the startup_CAN_cyclic() function (which is independent of the startup_CAN() function).

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ startup_CAN_cyclic(0) *Startup CAN cyclic messages*

This function is used to explicitly start up the CAN system's cyclic messages. This includes both C_CYCLIC and C_SYNC messages. These messages are not started automatically.

Parameters: none.

Returns: 1.

Reported Errors: none.

FUNCTION ⇒ **setup_CANRO_mask(1)** *Install new mask for receive-only mailbox*

CANRO is a receive-only mailbox. You can't use it to transmit data.

CANRO has its own mask register. This mask corresponds on a bit-for-bit basis to its 11-bit identifier field. Each place in the mask that is set to 0 is a "don't care" bit in the identifier field; each bit in the mask that is set to 1 is a "do care". When a new message comes in, only those bits set to 1 in the mask register are used to determine if the message should go into CANRO.

This function allows you to setup CANRO as a catch-all buffer, or as a global error message collector. For example, by making the mask equal to 0x0780, CANRO will only be sensitive to the Message Type field (bits 7-10). As shown below, bits 0-4 (slave ID) and 5-6 (master ID) will be ignored.

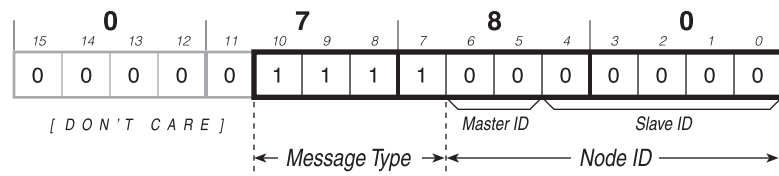
Parameters

Mask New mask.

Normal Returns: 1, 0.

Reported Errors: MO_INACTIVE.

Example: *CANRO Mask = 0x0780*



E•22

COMBINING
CAN_{OPEN} & VCL CAN

Nine of the fourteen CAN mailboxes are pre-allocated to CAN_{Open}; the other five (CAN1 through CAN5) are always available for you to define as you wish with VCL CAN.

The simplest hybrid system combining CAN_{Open} and VCL CAN is therefore to leave CAN_{Open} fully enabled and define CAN1-CAN5 to provide additional CAN mailboxes.

If your application requires more than five customized mailboxes—and does not use all the predefined CAN_{Open} mailboxes—you can reassign the unused CAN_{Open} mailboxes.

The fourteen CAN mailboxes are allocated as follows:

Available to VCL CAN	CAN1	Constant 0	(0x0000)	<i>user defined</i>
	CAN2	Constant 1	(0x0001)	<i>user defined</i>
	CAN3	Constant 2	(0x0002)	<i>user defined</i>
	CAN4	Constant 3	(0x0003)	<i>user defined</i>
	CAN5	Constant 4	(0x0004)	<i>user defined</i>
Pre-allocated to CAN _{Open}	CAN6	Constant 5	(0x0005)	CANOPEN_EMERGENCY_MAILBOX
	CAN7	Constant 6	(0x0006)	CANOPEN_NMT_MAILBOX
	CAN8	Constant 7	(0x0007)	CANOPEN_HEARTBEAT_MAILBOX
	CAN9	Constant 8	(0x0008)	CANOPEN_SDO_MISO_MAILBOX
	CAN10	Constant 9	(0x0009)	CANOPEN_SDO_MOSI_MAILBOX
	CAN11	Constant 10	(0x000a)	CANOPEN_PDO_MISO_1_MAILBOX
	CAN12	Constant 11	(0x000b)	CANOPEN_PDO_MOSI_1_MAILBOX
	CAN13	Constant 12	(0x000c)	CANOPEN_PDO_MISO_2_MAILBOX
	CAN14	Constant 13	(0x000d)	CANOPEN_PDO_MOSI_2_MAILBOX

For example, if you don't need CAN_{Open}'s PDO functionality, you could run the function `disable_CANOpen_pdo()`. This will free up for reassignment these four mailboxes:

```
CANOPEN_PDO_MISO_1_MAILBOX
CANOPEN_PDO_MOSI_1_MAILBOX
CANOPEN_PDO_MISO_2_MAILBOX
CANOPEN_PDO_MOSI_2_MAILBOX
```

to which you can now assign your own IDs: e.g., `CANOPEN_PDO_MISO_1_MAILBOX` equals `MY_NEW_MAILBOX`. You are encouraged to use the mailbox's name, as in this example, not its numerical constant.