

CONTROL TECHNOLOGY CORPORATION

Model 5300 Enhancements Overview

Model 5300 Enhancements Overview

Copyright 2006, 2009 © Control Technology Corporation
All Rights Reserved.

Blank



WARNING: Use of CTC Controllers and software is to be done only by experienced and qualified personnel who are responsible for the application and use of control equipment like the CTC controllers. These individuals must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes and/or standards. The information in this document is given as a general guide and all examples are for illustrative purposes only and are not intended for use in the actual application of CTC product. CTC products are not designed, sold, or marketed for use in any particular application or installation; this responsibility resides solely with the user. CTC does not assume any responsibility or liability, intellectual or otherwise for the use of CTC products.

The information in this document is subject to change without notice. The software described in this document is provided under license agreement and may be used and copied only in accordance with the terms of the license agreement. The information, drawings, and illustrations contained herein are the property of Control Technology Corporation. No part of this manual may be reproduced or distributed by any means, electronic or mechanical, for any purpose other than the purchaser's personal use, without the express written consent of Control Technology Corporation.

The information in this document is current as of the following Hardware and Firmware revision levels. Some features may not be supported in earlier revisions. See www.ctc-control.com for the availability of firmware updates or contact CTC Technical Support.

Model Number	Hardware Revision	Firmware Revision
5300	All Revisions	>= 5.00.90R60

TABLE OF CONTENTS

OVERVIEW **7**

VARIANTS **9**

REGISTER SUPPORT 9
PROPERTIES AND ARRAY SUPPORT 10
VARIANT STORAGE 12
VARIANT STRING EXTENSIONS 12
SCRIPT LANGUAGE - REGISTERS 13
FLOATING POINT SUPPORT 13
LOG SELECTION REGISTER – READING/WRITING VARIANT DATA TABLES 14

SCRIPT LANGUAGE COMMANDS **15**

DATA TABLE 15
load datatable [Variant regnum] [filename] *15*
save datatable [Variant regnum] [filename] *16*
DIAGNOSTICS 16
disktest 1 [file size] [block size] [/path/file] *16*
disktest 2 [file size] [block size] [/path/file] *16*
QUICKSTEP 17
enable quickstep2 ..... *17*
disable quickstep2 ..... *17*
FILE SYSTEM 17
set close nvariant [Variant #]" *17*
set logpath [path] *18*
set scriptspath [path] *18*
set nvariantpath [path] *18*
set emailspath [path] *18*
set webpath [path] *18*
set firmwarepath [path] *18*
set programspath [path] *19*
set datatablespath [path] *19*
copy [source path/file] [destination path/file] *19*
MONITOR 19
mon tfs init *19*
mon tfs rm *19*
mon tfs ls *19*
mon reboot *19*
MISCELLANEOUS 19
get vproperties [Variant #] *19*
printf [format string…] *20*
clear startup project *20*
get project *20*
get project info [project file] *20*
get startup project *20*
run project [opt. project file] *20*
set startup project [opt. project file] *20*

MONITOR **21**
BOOTING 21
DIAGNOSTIC PORT 22
USEFUL MONITOR COMMANDS 23
help *23*
tfs init *23*
tfs ls *23*
tfs rm *23*
reset *23*
version *24*
xmodem *24*
RE-FLASHING CONTROLLER WITH MONITOR SERIAL PORT 24
CONTROLLER LED OPERATIONS 25
MEMORY MAP **27**
C-API **29**
REGISTERS ACCESS 29
READ/WRITE REGISTERS 30
regVRead *30*
regVWrite *31*
Example Read/Write *32*
REGISTER LOCKING/UNLOCKING 33
regLock *33*
regUnlock *33*
MEMORY ALLOCATION 34
RESOURCE FILTERS 34
TASK CONTROL 35
createTask *36*
getTaskFromHandle *37*
taskSetBranch *37*
taskStepLock *38*
killTask *39*
SCRIPT EXECUTION 39
CTC_allocateCommandParser *40*
CTC_releaseCommandParser *40*
CTC_runCommandParser *41*
CTC_getCommandParser *42*
VARIANT INTERNAL MEMORY STORAGE (REFERENCE ONLY) 43
SERIAL PORT ADMINISTRATIVE FUNCTIONS **47**
SERIAL ADMIN ACCESS 47
FILE TRANSFERS 47
DNS SUPPORT **49**
DNS & 5300 50
CTNET VARIANT INTERFACE **51**
INTERFACE DEFINITIONS 51
VB6 DLL Function Definitions *51*
Constants *51*
Structure – CT_VARIANT *52*
Structure – CT_VARIANT_BLOCK… *52*
DLL Function Declarations *54*
DLL FUNCTIONS 54
CtGetVProperties *54*

CtGetVRegister *55*
CtGetVRegisters *56*
CtGetVRegisterBlock… *57*
CtSetVRegister *61*
CtSetVRegisters *61*
CtRunCommand *62*

CTNET BINARY PROTOCOL (SERVER INTERFACE) 65

BINARY PROTOCOL 66
Protocol Framing *66*
Binary Protocol Commands *68*
Variant Packets *70*
Register and Flag Access Command/Response definitions *71*
Variant Structures *73*
Variant Access Commands *76*
Register and Flag Access Commands *85*
Digital Input/Output Access Commands *90*
Analog Input and Output Access Commands *93*
Servo Access Commands *95*
Data Table Access Commands *98*
System and Controller Status Access Commands *101*
IP ENCAPSULATION 106

QUICKSTEP 2 & QUICKBUILDER SYMBOLS 109

QUICKSTEP 2 SYMBOL TABLE 109
QUICKSTEP 2 HMI COMMUNICATIONS 110
QUICKBUILDER SYMBOL TABLE 111
QUICKBUILDER HMI COMMUNICATIONS 112

CHAPTER

1

Overview

The 5300 series controllers are the first to allow a user to select between the existing Quickstep™ development environment and that of the newly released QuickBuilder™. Numerous functional enhancements have been introduced with the 5300 to leverage the power of QuickBuilder, where possible, Quickstep programmers may also make use of these features. This document details the available features from a register and script perspective, thus allowing access from both development environments. Additionally it provides 5300 specific information with regards to Monitor operation, re-flashing, SDISK operation (TBD), 'C' and Visual Basic programming, etc.

This guide is meant to be used in conjunction with the following manuals, noting exceptions and additions when relevant:

WebMON 2.0 User's Guide	Applications Guide	951-520012
Model 5200 Remote Administration Guide	Applications Guide	951-520001
Model 5200 Communications Guide	Applications Guide	951-520002
Model 5200 Script Language Guide	Applications Guide	951-520003
Model 5200 'C' User Programming Guide	Applications Guide	951-520004
Model 5200 Logging & FTP Client Applications Guide	Applications Guide	951-520015

Blank

CHAPTER

2

Variants

A new form of register has been added to the Quickstep/QuickBuilder environment, currently supported only on the 5300 series controller. These registers are called Variants. Much like variants from the old Microsoft Visual Basic® programming language, these registers change their appearance and type depending upon how they are used. Both public volatile and non-volatile Variant registers exist; additionally each task now supports 100 volatile local Variant registers (not public to other tasks). One and two dimensional arrays are also supported, thus allowing for multiple tables of mixed type cell storage.

Register Support

Similar to existing legacy registers, Variant registers default to a standard 32 bit signed integer register. The significant difference is that they may also be used for string and floating point storage as well as one and two dimensional arrays. Conversion is automatic depending upon how it was last used and 'C' API calls can be used to read and write any type of storage desired, with automatic conversion.

The 36001 register block is reserved for control and access to Variant registers. A new concept of volatile local registers is being introduced, where the first 100 registers, 36001 to 36100, are local to a task, meaning each has its own private copy that others can not view or modify. Registers 36101 to 36700 are 600 public volatile registers, similar to the existing controller registers in scope. Registers 36701 to 36800 are 100 non-volatile public Variant registers. In summary:

36001 – 36100: 100 volatile local Variant registers (repeated for each task).
36101 – 36700: 600 volatile public Variant registers
36701 – 36800: 100 non-volatile public Variant registers

In addition there are special registers within the 36000 block to both manipulate Variants as well as access special functions. For example:

36804 to 36811 – Special integer based registers to modify and access variants
36820 – QuickBuilder variant properties array (internal use)
36821 – QuickBuilder property name string array (internal use)
36822 – Motion control access where row is axis (0 is first) and column is property.

Properties and Array Support

Arrays are supported for Variant registers and created under program control, simply by writing a value to them. Arrays can be useful to read a row of information from a file, recipe parameters, etc. File access of a record would create a separate field automatically in an array of whatever type is available, much like a database record. All Variant registers support both single and two dimensional arrays, similar to a Quickstep 2 data table, having a row and column (single dimensional arrays are a column size of 1 and expand with regards to the row size).

When working with Variants they can be thought of as a single register when referenced by its base register number, such as 36001. That same base element is equivalent to 36001[0] and 36001[0][0]. All would return the same value. The array braces are referenced as [row][column, or [row] for single dimensional arrays. Each row, and or column, may be any combination up to 32768 cells as long as memory is not exceeded.

Access to the indexes of a Variant is accomplished by using special purpose registers. These registers are private to a Quickstep task (independently duplicated for each task). A public version is used for communications programs and remote access, such as CTCMON such that they will not corrupt those used by a task. QuickBuilder hides these special registers from your programming, simplifying their uses. With QuickBuilder signed integers are automatically allocated to the legacy 32 bit registers and all float, double and strings use variant storage.

Access to a Variant array can be done in a number of ways:

1. QuickBuilder
2. Variant property registers
3. Script execution
4. ‘C’ API calls
5. CTNet Communications DLL
6. Telnet command line

Variant property register access:

Variant Selection Register – Register 36804. Pointer to a Variant register (36001 to 36800) whose property to access when referencing the base variant. This should be the very first operation when modifying properties via individual register access.

Variant Column Size Register – Register 36805. Read only. Number of Columns in Variant.

Variant Row Size Register – Register 36806. Read only. Number of Rows in Variant.

Variant IndexCol Register – Register 36807. Index value of Variant selected, base is 0, along the Column axis, columns. Set to 0 for single dimensional arrays

Variant IndexRow Register – Register 36808. Index value of Variant selected, base is 0, along the Row axis.

Variant Indirection Register – Register 36809. Nonzero value makes the contents of the Variant reference a register number. Read/Writes will occur to that register, not the Variant. A zero clears the flag, disabling indirection. The existing IndexCol and IndexRow of the final accessed register are used to select the Variants specific element should an array exist.

Variant Deletion Register – Register 36810. Writing a 0x55AA (21930 decimal) to this register will cause the register pointed to by the 'Variant Selection Register' to be deleted and cleared, returning storage to the system.

Variant NVClose Register – Register 36811. Writing a 0x55AA to this register will cause a non-volatile variant register file to be closed/flushed. This is typically done prior to replacing a file on SDISK (TBD) and/or copy the file contents to another system or directory.

Note that the Selection Register must be written to first, prior to setting any property of a particular Variant.

Once an index property is configured a reference to the base Variant register will automatically index into the array, if present. If not present a 0 will be returned.

 ***Non-volatile variants*** *are a bit unique in their creation. If using an array you must write to the last cell ([row][column]) desired prior to access. Any value may be written and this is used to create an index table for faster access. You may expand the size of a non-volatile variant at any time by increasing the [row] dimension and writing to the last [row][column] cell desired. Note that once the [column] dimension is set it can not be changed, just the adding of rows. This limitation is not imposed on volatile variants but to optimize access speed it is highly recommended to follow the same procedure, although the column may be expanded with volatile variants. The purpose of writing to the last desired cell is to optimize the size of access caches. If you expand by 1 all the time then space is wasted given each cache is up to 1024 entries in size. Thus 2048 rows would have 2 cache tables but if you expanded by 1 for 2048 iterations then there would be 2048 tables instead of just 2. A separate table exists for each column.*

 *Volatile variant array cell storage only occupies memory consumption when something is written. Writing to the last cell only creates the caching tables. Thus creating a large array will not waste space until actual data is present.*

Variant Storage

Variant storage is allocated as needed, thus initially it may not execute as fast as a regular register due to dynamic memory allocation. There are currently 11 Megabytes of space available for allocating volatile Variants, with each cell consuming about 44 bytes (52 bytes non-volatile). A string will add 224 bytes, regardless of length (223 max with 0x00 string terminator) and is allocated only when needed. Non-volatile Variants are stored in battery backed static ram for a single cell and CRC'd upon each write access to ensure integrity at power up or reset. Non-volatile Variants are limited to a single cell, arrays are not supported, when the SDISK (TBD) option is not installed. If SDISK is available the entire storage area of the SDISK may be used to read/write large non-volatile Variants. The SDISK is used to stream data from the array file, as needed.

Variant String Extensions

Reading Variant strings uses the Controller enhanced version of 'sprintf' as a copy. Although slower, this means that special embedded register references, such as those used by 'message.ini' files are fully valid. This allows Variant strings to mix both dynamic and variable information as each read will cause an indirect reference to any embedded register strings:

Register references:
<register> = R####
<array index> = #### or R####

All below are valid references -
<register> or <register>[<array index>] or <register>[<array index>][<array index>]

For example:
If a Variant contains "The analog value is %dR8501." Then the value of Register 8501, Analog Input 1 will be read and inserted into the string prior to return.

Refer to 'message.ini' documentation but in general %s, %d, %c, %x, %X, %f, %e, %E, %g, %G, %T (time/date), and %c (ASCII character retrieval) are supported.

If scripting or the message.ini file, itself, were to be used to initialize a Variant then an extra % would have to be inserted into the string to prevent parsing during the read operation of the string, prior to writing to the Variant due to the Script engine support for the String Extensions as well. Thus %dR8501 would become %%dR8501. The first %

will be used as an escape character as is standard with 'sprintf' conversion rules. This is standard 'C' programming rules.

Script Language - Registers

Since Quickstep 2 is not able to initialize and directly write to a Variant (only indirectly using property registers) other methods are required to optimize the initialization of an element. Using the Script language, *'C-API',* or CTNet Communications DLL you may reference a Variant in numerous ways, although Quickstep 2 encounters problems in this regard. This section discusses variant register access using the script command language.

By default all Variants are integers until written differently. Special registers are supplied to load data either via a file (reference 'Log Selection Register' section) and/or a script. When using a script the normal scripting language is supported with the addition that all register syntax can optionally include array references and register indirection.

The previous script language allowed for "1 = 5" to cause an integer to be written to register 1. Referencing the Variant register block, "36101 = 5" results in the same operation. This may be expanded to now store:

Float - 36101=5.0 (with precision of 1)
String - 36101="this is a string when quotes are used"
Indirection – 36101=I2001 (where 2001 references another register)

Additionally arrays may be referenced as [### / R###], where R### is an indirect way to retrieve an array index, using the contents of any register, including non-variant.

Float – 36101[5] = 5.0 (6^{th} element set to 5.0, precision of 1)
String – 36101[R2][5] = "this is referencing a register contents for Row index"



R### may also be used to reference registers, for example, R36101 = 5.0.

Floating Point Support

Any Variant register, from 36001 to 36800 fully supports floating point storage and operations. Both 32 bit floats and 64 bit doubles are available. By default a double will be used, unless explicitly specified, when using the *'C-API'*. Operation when using Quickstep 2 instructions will be based upon what was last stored into the Variant element. The script language may be used at initialization to change a Variant from integer to floating point simply by executing the line "R#### = 0.0"

- *Constants*: A Script may be run to initialize constant registers and declare type, other than integer, in Quickstep 2. This is run using the same _startup.ini file that exists now.
  - 36101 = 3.14
  - 36102 = 5.67932

You may initialize as many registers as desired using the script file. The major enhancement is the Script engine supports floating point during register assignment, as well as arrays.

- *Printf conversion*: message.ini, log.ini files can reference the floating point register values (%f) to provide logging of float strings as well as sending them out serial ports. Variant register reference is fully supported.
- *I/O*: Writing a float to an analog output or other I/O will cause it to be converted to an integer, rounded appropriately, unless floating support is added at a later date for that particular I/O.

Log Selection Register – Reading/Writing Variant Data Tables

The Log Selection Register, 12325, has been enhanced to allow for writing and reading data tables stored in the ".tab" QS2 file format. This allows the contents of a variant to be written to a file in text format. Previously 000 to 999 was used to Log###.log file name references using the log.ini file. The added implementation now allows for array Variants to be written to the same log file as well as the ability to write and read a text based data table file.

In summary Log Selection Register is as follows:

000 to 999 – Same as before, log.ini file referenced.
1000 to 1999 – Variant array is written to log file, Log###.log, log.ini is not referenced.
2000 to 2999 – Reserved.
3000 to 3999 – Variant array is written to QS2 data table format using the file name, datatable###.tab.
4000 to 4999 – Reserved.
5000 to 5999 – Variant array is read from QS2 data table format using the file name, datatable###.tab.

The transaction would be initiated by writing the desired Variant register # to the "Log String Transfer Register", 12326.

 *Reference the "5200 Logging and FTP Application Guide", 951-520015, for additional information.*

CHAPTER

3

Script Language Commands

Numerous Script commands have been added to enhance the operation of the controller. Many of these are Variant related, now supporting strings and floats. Basic script execution and existing commands can be found in the *5200 Script Language Guide, 951-520003*.

Data Table

load datatable [Variant regnum] [filename]

The "load datatable" script command can be used to load a variant array from a file. This file is stored in the same format as the QS2 .TAB file format except that floats have a decimal point and strings are enclosed in quotes. A current QS2 file would be read in as integers given that is a limitation of the QS2 data table. The enhanced table that uses variants for storage is shown below:

```
data_table[4][6] =
{
      84     104     105     115      32     105
     115      32     114     111     119      32
      49       0       0       0       0       0
       0       0       0       0   6.789   "string"
}
```

Note that tabs or spaces may be used as a separator as all whitespace characters < '0' are ignored except LF, which designates the end of a line/row. There is no restriction on the line/row length except as required by each Variant cell (string 223 bytes). The first 2 lines are ignored and the actual table size is set by the data found in the file. A CR LF combination should follow the last '}' to denote the end of the file.

The last two cells show examples of a float, 6.789, and a string format "string". Remember the string may reference other registers using the %d, message.ini, format but an extra % is needed, %%d. The first % will be stripped when the string is parsed.

In the above example 84 would be loaded into array location [0][0], “string” into [3][5].

 *The load/save datatable commands list the array size as [row][column] for compatibility with QS2.*

save datatable [Variant regnum] [filename]

The ‘save datable’ script command operates exactly the same as the ‘load datatable’ command except that the Variant register contents are written to a file. Any unknown cells will contain a “?”. Two separate formats are available, QS2 compatibility mode and CSV (comma delimited, similar to a log file).

In QS2 compatibility mode seven spaces will be placed between cell data and a ‘.tab’ file extension must be used. Upon writing any existing file will be first deleted. The use of any other file extension will cause the CSV format to be used, where each cell is separated by a command and a single space. End of line is the same as the QS2 format, CR LF. There is no header information in CSV format thus the first array location will be the first bytes of the file.

 *The load/save datatable commands list the array size as [row][column] for compatibility with QS2.*

Diagnostics

disktest 1 [file size] [block size] [/path/file]

This command is used to perform a test of the file system. A file of the size [file size] will be written using blocks of size [block size] to the file [/path/file]. An incrementing byte pattern is written and verified. The complete write is done first, file closed and then read back and verified. For example to test a 1M file with a block size of 512 bytes:

Example: disktest 1 1000000 512 /SDISK/test1.bin

disktest 2 [file size] [block size] [/path/file]

This command is used to perform a test of the file system. A file of the size [file size] will be written using blocks of size [block size] to the file [/path/file]. An incrementing byte pattern is written and verified. The complete write is done first, file closed and then read back and verified. For example to test a 1M file with a block size of 512 bytes:

Example: disktest 1 1000000 512 /SDISK/test1.bin

Quickstep

enable quickstep2

This command enables QS2 operation at the next reboot, assuming a 1 is written to register 20096. Quickstep may be enabled and disabled from normal operation in order to conserve CPU cycles. This is typically done in a QuickLink application where no program is running on the remote client node.

System default is enabled.

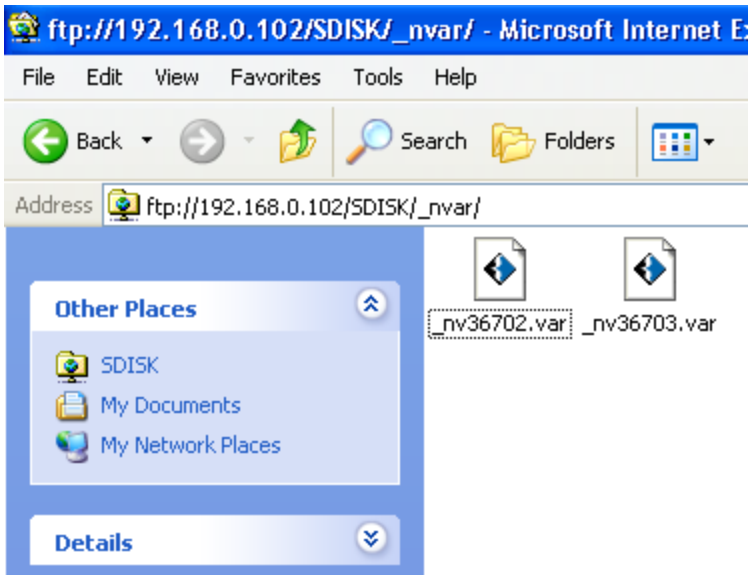
disable quickstep2

This command disables QS2 operation at the next reboot, assuming a 1 is written to register 20096. Quickstep may be enabled and disabled from normal operation in order to conserve CPU cycles. This is typically done in a QuickLink application where no program is running on the remote client node.

File System

set close nvariant [Variant #]"

Non-Volatile array variants are stored in files which may be transferred among controllers, deleted, copied, etc. The contents of the non-volatile variant are independent of the actual register number. The file name determines its assignment. Hence _nv36702 will be used for register 36702.



If that file was copied to file _nv36703 then the data has now been duplicated and register 36702 and 36703 now have the same data. When replacing a file it is important to close

it first. Not closing it means it can not be deleted. You may copy a non-closed file but make sure no task is writing to it or the data may be changing in the background.

Closing a file is also important if you are replacing non-volatile variants data since upon access if the file is closed it will re-attempt to open it and in this instance find the new file. Open files can not be replaced or deleted.

Example: set close nvariant 36702

This would close the file _nv36702 should it be open using the default path of /SDISK/_nvar.

To close all open non-volatile specify a -1 as the [Variant #]. Should an error occur the ERROR_OO_RANGE flag is set.

set logpath [path]

Sets an alternate log file storage path. Power up default is /_system/Messages.

Example: set logpath /SDISK/myLogDir

set scriptspath [path]

Sets an alternate scripts file storage path. Power up default is /_system/Scripts.

Example: set scriptspath /SDISK/myScriptsDir

set nvariantpath [path]

Sets an alternate log file storage path. Power up default is /SDISK/_nvar which will automatically be created at power up if it does not exist (one level only in directory tree).

Example: set nvariantpath /RAM/mylogdir

set emailspath [path]

Sets an alternate emails file storage path. Power up default is /_system/Emails.

Example: set emailspath /SDISK/myEmailsDir

set webpath [path]

Sets an alternate web file storage path. Power up default is /_system/Web.

Example: set webpath /SDISK/myWebDir

set firmwarepath [path]

Sets an alternate log file storage path. Power up default is /_system/Firmware.

Example: set firmwarepath /SDISK/myFirmwareDir

set programspath [path]

Sets an alternate programs file storage path. Power up default is /_system/Programs.

Example: set programspath /SDISK/myProgramsDir

set datatablespath [path]

Sets an alternate datatables file storage path. Power up default is /_system/Datatables.

Example: set datatablespath /SDISK/myDatatablesDir

copy [source path/file] [destination path/file]

This command is used to copy a file from one location to a new location, creating a new file and overwriting any existing.

Example: copy /SDISK/_nvar/_nv36702 /SDISK/_nvar/_nv36750

A new non-volatile Variant, 36750 now exists with the same contests of register 36702. SCRIPT_ERRMASK_FILE error bit is set if an error occurs.

Monitor

mon tfs init

Initialize and clear the controller monitor file system.

mon tfs rm

Remove a file from the controller monitor file system.

mon tfs ls

List all the files within the controller monitor file system, typically just one, which is the application program to execute.

mon reboot

Exit the controller application and reboot into the monitor, re-loading the entire controller program once again. Basically a full reset where VBIAS will be shut off and execution of application programs immediately terminated without notification.

Miscellaneous

get vproperties [Variant #]

Since there are currently no utilities to view variant information, such as how big it is, array size, floating point precision, etc, this command will display that information:

Example: get vproperties 36702

SUCCESS: Nonvolatile Variant 36702, Size: rows - 50 / columns - 200, precision: 6.

printf [format string…]

The 'printf' command allows for a string format to be tested prior to inclusion in a variant cell or message.ini file. The string will be parsed exactly as it would when used in these applications. This command is typically used for testing only as it has no effect other than visual final string presentation.

Example: printf "The contents of register 100 = %dR100"

The " " are required.

clear startup project

This command will clear the currently default project which is invoked at reset or power up, thus none will be executed upon power up.

get project

This command will display the currently active project that is running.

get project info [project file]

This command is used to determine the contents of a QuickBuilder project file.

get startup project

This command will display the project set to run at power up or reset.

run project [opt. project file]

This command is used to load and run a QuickBuilder project file. The controller is restarted. If not project is specified the last saved project will be run.

set startup project [opt. project file]

This command is used to save a specific project file name/location upon with to run at power up and reset, as the default. If none is specified then the last executed path/name will be saved.

CHAPTER

4

Monitor

Upon power up or system reset the controller boots into a system monitor. This monitor maintains its own flash file system as well as the main controller execution image in the industry standard ELF format. Typically a user never needs to access the monitor as the boot process is fully automatic. Occasionally a problem may occur, such as power loss during a firmware upgrade that may result in a controller that can not boot normally, requiring low level access.

Booting

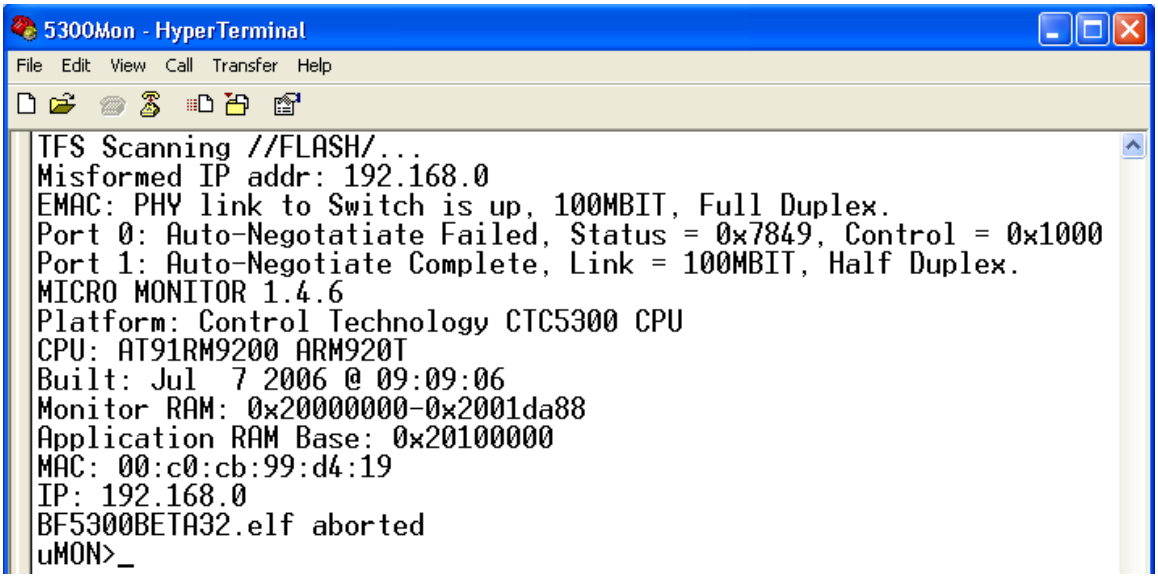
Upon power up or reset the integrity of the controller execution image is confirmed. After successful diagnostics a normal boot process will begin. LED patterns are defined as follows:

5300 BOOT LED OPERATIONS	
ALL ON	Hardware Reset
ST1/ST2 ON	Program Mode - Stopped, awaiting special boot abort key sequence ^X ESC ESC
ST1	Program Mode - Loading SDRAM from FLASH
ST1/ ST2 ON	Program Mode - Bootloader Stopped, transferring control to main program
ST1/ST2/ST3	Program Mode - Resetting
ST1/ST3, FAULT fast flashing (50%)	Program Mode - DHCP in progress
ST1/ST2/ST3	Program Mode - Reseting
ALL OFF	Program Mode - Running, normal operation

While the ST1/ST2 LED's are lit, initially after hardware reset, a user may abort the boot process by quickly sending the CNTL X, ESC, ESC key sequence, on the COM4 serial port, to the controller. The COM4 port operates as a diagnostic port during booting and may be connected to HyperTerminal or similar programs.

Diagnostic Port

The diagnostic port, controller COM4, operates at 38400 baud, 8 data bits, 1 stop bit, and no parity. Under normal operation no characters will be sent out this port unless an error occurs. To force entry into diagnostics mode send a CNTL X, ESC, ESC key sequence (0x18, 0x1b, 0x1b) to the controller, while in the 'Stopped Program Mode'. When data begins to appear on the terminal display a CNTL C (0x03) but be pressed to fully abort otherwise the boot sequence will only appear in verbose mode and booting will continue:



Analyzing the message that appeared, the first line is checking the integrity of the execution image as it resides within the file system. The Misformed IP addr: 192.168.0 is proper as this disables the bootp feature of the controller (not supported) and forces it to remain off an Ethernet network while booting. The monitor version of 1.4.6 is shown as being the present revision followed by the date and time it was built. The MAC address assigned to the Ethernet adapter is display (MAC: 00:c0:cb:99:d4:19) and is unique to all controllers. The controller image file that was aborted at boot, BF5300BETA32.elf, is also displayed, varying based upon the revision level. The uMON> prompt now allows for additional monitor commands to be completed by the user.

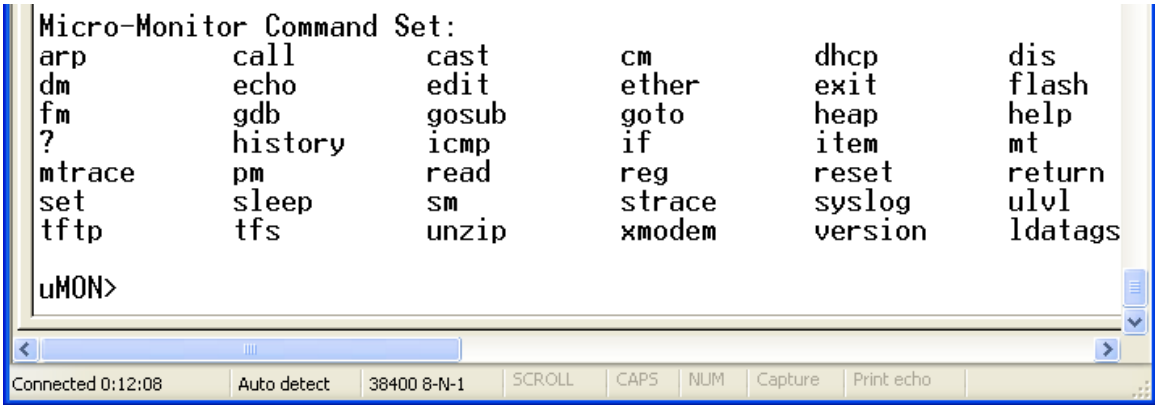
When re-flashing the monitor the existing MAC-ID is preserved but header information will be sent out the serial port for verification purposes. The COM4 cable should always be disconnected from external communications if the monitor is re-flashed and this would cause a communications issue with that host system.

Useful Monitor Commands

Only a short summary of available commands will be detailed below since the user does not need to interact with the monitor under normal operating conditions.

help

By entering 'help' a general command summary is made available. Additionally entering 'help' followed by one of the command listed will provide additional help screens.



As can be seen from the 'help' screen Micro-Monitor is being used as the boot monitor. This is an industry standard, open source, monitor. For detailed command descriptions you may reference their web site at:

http://www.microcross.com/html/micromonitor.html

Micromonitor V1.4 is the current revision being used in the controller (version command). Full compliance is not guaranteed nor supported.

tfs init

Initialize the file system and erase all files. Typically done prior to loading a new image via the serial port to ensure enough file space. Similar to formatting a disk drive.

tfs ls

Get a directory of all files in the monitor file system.

tfs rm

Delete a file.

reset

Reset the controller and reboot.

version

Return the Micromonitor version being used.

xmodem

xmodem is an industry standard file transfer protocol and may be used to receive a new controller image file via the COM4 serial port. The process begins with entering the following on the uMON> command line:

uMON> tfs init (format the disk)
uMON> xmodem –k –d –F BF5300V0501.elf –f bE (where the .elf file is one loading).

Reference the next section for further detail on xmodem usage.

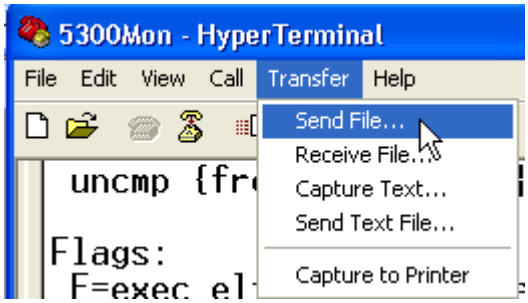
Re-Flashing Controller with Monitor Serial Port

The monitor may be use to re-flash the main controller execution image. This is not the typical procedure as normally Kermit file transfer protocol will be used on a serial port or FTP when using Ethernet.

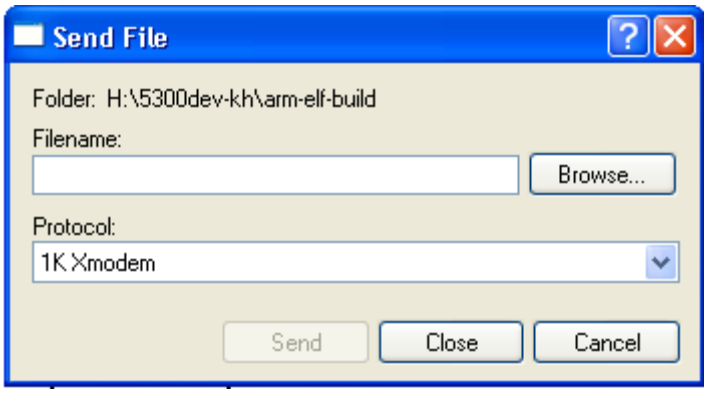
To enter this mode at power up us the method described in the "Diagnostic Port" section to obtain a monitor prompt and proceed as follows:

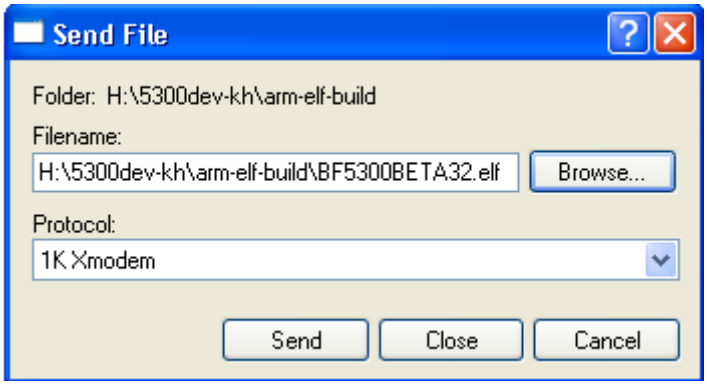
uMON> tfs init (format the disk)
uMON> xmodem –k –d –F BF5300V0501.elf –f bE (where the .elf file is one loading).

Upon hitting the enter key some strange characters will appear under the prompt, this is part of the protocol. To abort press the CNTL C keys. To transfer the file using Hyperterminal select Transfer -> Send File



A dialog box will appear, ensure "1K Xmodem" is the active protocol and then select the Browse button to find the file to transfer:





Click the 'Send' button when ready and a progress screen will appear. The uMON> prompt will once again appear when the file is fully received. The 'reset' command may be used to reboot or cycle power on the controller.

Controller LED Operations

5300 CPU and Expansion Boards - LED OPERATIONS		
LED Label	**State**	**Description**
PWR (Backplane Power)	Steady OFF	BackplaneRack is not powered up
	Steady ON	Backplane is powered up
FLT (Backplane Fault)	Steady OFF	Normal Operations: No fault on the local backplane.
	Steady ON	Hardware fault on the local backplane.
	Slow Flash (50% Duty)	Software fault on the local backplane.
	Fast Flash (50% Duty)	*CPU Only: DHCP in progress*
	Blink TBD	*CPU Only: Flash reprogramming in progress.*
ST1 - ST3 (Global Status Code)	Steady Binary Code 0 (Off, Off, Off)	*CPU Only:* Program Mode - Normal/Running operation
	Steady Binary Code 1 (Off, Off, On)	*CPU Only:* Fault - Global Hardware (i.e. one of the backplanes has a hardware fault)
	Steady Binary Code 2 (Off, On, Off)	*CPU Only:* Fault - Global Software (i.e. one of the backplanes has a software fault)

ST1 - ST3 (Global Status Code)	Steady Binary Code 3 (Off, On, On)	*CPU Only:* Fault - Corrupt user program or data table or NVRAM
	Steady Binary Code 4 (On, Off, Off)	*CPU Only:* Program Mode - Loading program or flashing flash, SL# is slot, if booting, loading program to SDRAM from flash.
	Steady Binary Code 5 (On, Off, On)	*CPU Only:* Program Mode - DHCP in progress, program not running
	Steady Binary Code 6 (On, On, Off)	*CPU Only:* Program Mode - Stopped or if booting awaiting abort boot escape sequence.
	Steady Binary Code 7 (On, On, On)	***CPU Only:* Power Up / Reset State or Program Mode Restarting/Reset (SL# off)**
SL1 - SL3 (Backplane Slot Code)	Steady Binary Code 0 (Off, Off, Off)	Local Slot #1 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 1 (Off, Off, On)	Local Slot #2 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 2 (Off, On, Off)	Local Slot #3 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 3 (Off, On, On)	Local Slot #4 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 4 (On, Off, Off)	Local Slot #5 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 5 (On, Off, On)	Local Slot #6 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 6 (On, On, Off)	Local Slot #6 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.
	Steady Binary Code 7 (On, On, On)	Local Slot #8 fault when local **FLT** or global **ST1-3** are in a non-normal operation state.

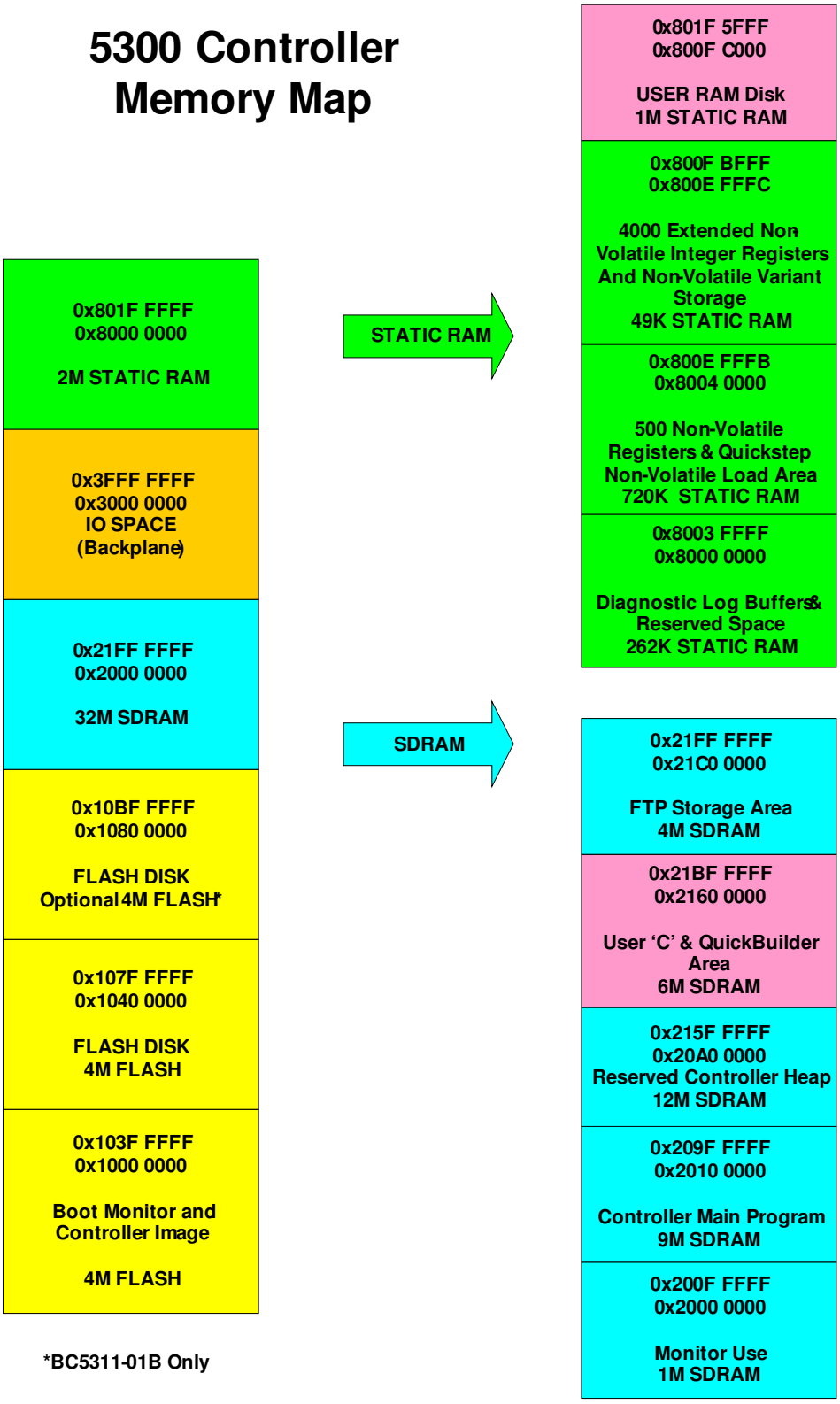
CHAPTER

5

Memory Map

The 5300 Controller is made up of differing types of memory. Primarily consisting of Flash, SDRAM, and Static RAM. Most areas are reserved for controller and QuickBuilder use but there are user areas available. Certain portions of the Static RAM are non-volatile and may be used for a RAM Disk. Flash Disk is available as either a 4M (BC5311-01A CPU) or 8M (BC5311-01B CPU) disk. There is also 6M of SDRAM space that may be used for 'C' API development when QuickBuilder is not in use.

5300 Controller Memory Map



*BC5311-01B Only

CHAPTER 6

C-API

The functionality of the C-API has been extended to allow for better operational control of tasks as well as providing a full variant interface. This same interface is currently used by QuickBuilder. Re-linked 5200 C-API 'C' code is backward compatible. Presently 6 Meg of SDRAM is reserved for C-API operation. The 5200 'C' User Programming Guide, 951-520004 still applies except that a new 5300UserC development environment is available for download from Control Technology's web site. The sections below detail additional 'C' API calls that are available.

Registers Access

All registers are treated as Variants for read and write operations with the proper conversion occurring should a register be restricted to a limited functionality, such as only supporting integers.

Previously, in the 5100/5200 'C' API, the functions regWrite and regRead were available. A new extended function to allow for Variant access has been added. These functions are regVWrite and regVRead. It is recommended that this call be used for all register access from 'C'. The currently available Variant types are:

VARIANT_INTEGER - (32 bit signed integer)
VARIANT_UINTEGER - (32 bit unsigned signed integer)
VARIANT_FLOAT - (32 bit storage)
VARIANT_DOUBLE - (64 bit storage)
VARIANT_STRING - (223 character limit)
VARIANT_STRING_PTR - (223 character limit, pointer to string)
VARIANT_STORAGE_TYPE - enhanced structure to directly specify elements within an array and the type directly)

The most flexible storage type that will be used is the VARIANT_STORAGE_TYPE. Using the other types directly means the base index registers of a Variant will be referenced for element access which generally is restricting and actually meant more to provide a means for QS2 to utilize variants. If no array is used then this is the most

efficient but if there is an array the use of a VARIANT_STORAGE_TYPE allows a single call to perform the full index and element access. Its structure is:

```
// Universal variant storage structure for 'C' regRead/regWrite, aligned 4 byte boundry
#define VARIANT_MAX_STRING 223
typedef struct
{
        int type;       // type of storage being used or requested
                        // If -1 on read then return current, else set to type want.
                        // On write must set to type that is stored within this structure

        // double to string conversion precision %.6f default
        // On read is what is presently set, write what want.
        unsigned char precision;

        // special flags for processing so far only
        // VARIANT_INDIRECTION_FLAG used, can be used to set property
        // in ->settings on write operation, no effect on read.  Written
        // value becomes register to reference for further operations.
        unsigned char flags;

        // 00, no operation other than read/write specified, else do defined
        // operation.  Currently have write for properties access to 'settings'
        // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
        // write value ignored.
        unsigned char cmd;

        // task number (offset in task array + 1, where 0 is 1) or handle thus usable from
        // remote or 'C' API, 4096 to 65535, set to 0 for public reg.  Task number is QS2
        // style reference, valid for 1 to N tasks (typically 96 max).  Use 'get tasks to
        // retrieve a tasks handle.
        unsigned short taskHandle;

        // this is reserved for later use and possible string length if want unsigned
        // char, 0 - 255 values, VARIANT_BYTE, future type
        unsigned short slength;

        unsigned int indexCol;   // Column dimension index reference
        unsigned int indexRow;   // Row dimension index reference
        union                   // Data that was read or has been written of 'type'
        {
                int iValue;
                unsigned int uiValue;
                float fValue;
                double dValue;
                char sValue[VARIANT_MAX_STRING+1];
        } data;
} VARIANT_STORAGE;
```

Read/Write Registers

regVRead

```
/*************************************************************************/
/*                                                                       */
/*  FUNCTION (public)                                                    */
/*                                                                       */
/*    RETVAL regVRead                                                    */
/*                                                                       */
/*  DESCRIPTION                                                          */
/*                                                                       */
/*    This function allows the caller to read any type of data within    */
/*    a variant register, converting from the last written, if different. */
/*                                                                       */
/*  PARAMETERS                                                           */
```

```
/*                                                                          */
/*    TASK *task - Pointer to current Quickstep task or NULL if none.       */
/*                 Local registers are stored within the TASK structure     */
/*    UINT16 RegNum - Register desired, 1 based.                            */
/*    void *RegVal - pointer to store result of 'type' to, if string        */
/*                   then an array of at least 256 characters is required   */
/*    int type - Information on what 'RegVal' points to:                    */
/*               VARIANT_INTEGER, VARIANT_FLOAT, VARIANT_DOUBLE,            */
/*               VARIANT_STRING, VARIANT_STORAGE_TYPE, or                   */
/*               VARIANT_BASE_PROPERTIES_TYPE                               */
/*                                                                          */
/*               Note 1:  When using VARIANT_STORAGE_TYPE its internal      */
/*               settings effect the call:                                  */
/*               ->type = Specifies the storage type desired where          */
/*                        VARIANT_CURRENT_TYPE means last written type.     */
/*                        Upon return this field is updated with type.      */
/*                        Specific types may also be requested (VARIANT_..*/
/*               ->indexCol= -1 if use base Variant index pointer otherwise */
/*                        specifies the array element to write to.          */
/*               ->indexRow= if not used leave at 0, else row array dimension */
/*               ->precision= Number of digits of double/float precision    */
/*                        if reading that type.  Used for string conv.      */
/*                        when needed.  Default is 6.                       */
/*               Note 2:  When using VARIANT_BASE_PROPERTIES_TYPE it is     */
/*                        used to retrieve the current internal properties*/
/*                        of the variant based upon index/Y.                */
/*                                                                          */
/*  CALLS                                                                   */
/*    Quickstep OS virtual table function pointer                           */
/*                                                                          */
/*  RETURNS                                                                 */
/*                                                                          */
/*    SUCCESS if conversion successful                                      */
/*    ERROR_NO_REG if invalid register                                      */
/*    ERROR_NOT_DEFINED there is no API access???                           */
/*                                                                          */
/****************************************************************************/
RETVAL regVRead( void *task, UINT16 RegNum, void *RegVal, int type);
```

regVWrite

```
/****************************************************************************/
/*                                                                          */
/*  FUNCTION (public)                                                       */
/*                                                                          */
/*    RETVAL regVWrite                                                      */
/*                                                                          */
/*  DESCRIPTION                                                             */
/*                                                                          */
/*    This function allows the caller to read any type of data within       */
/*    a variant register, converting from the last written, if different.   */
/*                                                                          */
/*  PARAMETERS                                                              */
/*                                                                          */
/*    TASK *task - Pointer to current Quickstep task or NULL if none.       */
/*                 Local registers are stored within the TASK structure     */
/*    UINT16 RegNum - Register desired, 1's based                           */
/*    void *RegVal - pointer to get data to store of 'type' to, if string   */
/*                   then an array of at least 256 characters is required   */
/*    int type - Information on what 'RegVal' points to:                    */
/*               VARIANT_INTEGER, VARIANT_FLOAT, VARIANT_DOUBLE,            */
/*               VARIANT_STRING, or VARIANT_STORAGE_TYPE                    */
/*                                                                          */
/*               Note:  When using VARIANT_STORAGE_TYPE its internal        */
/*               settings effect the call:                                  */
/*               ->type = Specifies the storage type within the structure   */
/*               ->indexCol= -1 if use base Variant index pointer otherwise */
/*                        specifies the array element to write to.          */
/*               ->indexRow= if not used leave at 0, else row array dimension */
```

```
/*               ->precision= Number of digits of double/float precision  */
/*                         if writing that type.  Used for string conv.    */
/*                         when needed.                                    */
/*                                                                         */
/*  CALLS                                                                  */
/*    Quickstep OS virtual table function pointer                          */
/*                                                                         */
/*  RETURNS                                                                */
/*                                                                         */
/*    SUCCESS if conversion successful                                     */
/*    ERROR_NO_REG if invalid register                                     */
/*    ERROR_NOT_DEFINED there is no API access???                          */
/*                                                                         */
/***************************************************************************/
RETVAL regVWrite( void *task, UINT16 RegNum, void *RegVal, int type);
```

Example Read/Write

Example to create an array [3][50]:

```
VARIANT_STORAGE vStorage;

        vStorage.indexCol = 49;  // one less since 0 based
        vStorage.indexRow = 2;
        vStorage.type = VARIANT_INTEGER;
        vStorage.iValue = 55;
        vStorage.taskHandle = 0;
        int regnum = 36701;
        // simply write a value and all newly created elements are ints.
        regVWrite(NULL, regnum, (void *)&vStorage, VARIANT_STORAGE_TYPE);
```

You do not need to specifically create a variant prior to use, simply write to it with the value to store and type. The above simply creates an array of all integers with the last one having a 55 stored. If the array already existed then it will be expanded along whatever axis or axis's is greater, preserving the storage values of the existing Variants. New elements which are not being directly written to but are created to access the one that is, are initialized to VARIANT_INTEGER with a value of 0.

To read a register (36701) as a string, regardless of what the current storage type is:

```
VARIANT_STORAGE vStorage;

        vStorage.indexCol = 0;  // read the first element, [0][0]
        vStorage.indexRow = 0;
        vStorage.type = VARIANT_STRING;
        vStorage.sLength = 0;
        int regnum = 36701;
        regVRead(NULL, regnum, (void *)&vStorage, VARIANT_STORAGE_TYPE);
```

To use the default index setting presently in the Variant:

```
        regVRead(NULL, regnum, vStorage.sValue, VARIANT_STRING);
```

Register Locking/Unlocking

Additional locking functions have also been made available to reserve atomicity beyond a single register access. Typically a register is locked only during its normal regRead/regWrite function call. In some cases it may be necessary to read/write modify a register and ensure atomicity. Thus the following functions are available:

regLock

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    regLock                                                PORTABLE C      */
/*                                                              1.1          */
/*  DESCRIPTION                                                              */
/*    Request a lock on a register group by specifying a specific reg.       */
/*    Lower level mutex will be obtained, if not already owned, suggest      */
/*    unlocking in the reverse order locked.                                 */
/*                                                                           */
/*  INPUT                                                                    */
/*    UINT16 RegNum - Register number from 1 to 64536 to read                */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    NONE                                                                   */
/*                                                                           */
/*  CALLS                                                                    */
/*    Quickstep OS virtual table function pointer                            */
/*                                                                           */
/*  CALLED BY                                                                */
/*    As required by user code                                               */
/*                                                                           */
/*  RELEASE HISTORY                                                          */
/*                                                                           */
/*    DATE              NAME                      DESCRIPTION              */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
void regLock(UINT16 RegNum);
```

regUnlock

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    regUnlock                                              PORTABLE C      */
/*                                                              1.1          */
/*  DESCRIPTION                                                              */
/*    Request an unlock on a register group by specifying a specific reg.    */
/*    Lower level mutex will be obtained, if not already owned, suggest      */
/*    unlocking in the reverse order locked.                                 */
/*                                                                           */
/*  INPUT                                                                    */
/*    UINT16 RegNum - Register number from 1 to 64536 to read                */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    NONE                                                                   */
/*                                                                           */
/*  CALLS                                                                    */
```

```
/*    Quickstep OS virtual table function pointer                          */
/*                                                                         */
/*  CALLED BY                                                              */
/*    As required by user code                                             */
/*                                                                         */
/*  RELEASE HISTORY                                                        */
/*                                                                         */
/*    DATE              NAME                      DESCRIPTION              */
/*                                                                         */
/*                                                                         */
/***************************************************************************/
void regUnlock(UINT16 RegNum);
```

Note that this locks all registers that are in the specific group. Groups with individual locks are normal quickstep registers, volatile Variant registers, non-volatile Variant registers, and standard I/O register access. Local task registers do not need to be locked and a call will have not effect. Locking a register group will cause it to be locked from all other threads thus keep the length of time locked to a minimum.

Memory Allocation

The 'C' API can allocate memory using the standard 'malloc' and 'free' function calls. These routines have been re-written to not use the GNU libraries when linked with the 'C' API base 'C' code and include files.. C++ operations should also work as well, 'new' and 'delete'. Note that there is no optimization for block allocation thus be careful of fragmentation. Reserved heap size can be adjusted by setting the `MALLOC_DATA_SIZE` define statement in the CoreFunc.h file. The 'C' API is responsible for its own heap cleanup, reference the END_ALL_TASKS_HOOK.

Resource Filters

Resource Filters are supported in the 5100 and 5200 products. Their current implementation supports the requirements of QuickBuilder, thus allowing for a way to invoke 'C' functions when a unique register is read or written to. The only enhancement provided is the use of the FILTERPARAMS 'void *option' reference, allowing for access to the current TASK pointer.

```
/*************************************************************
*  struct filterparams - Resource Filter Function callback block*
*                                                             *
*  Registered resource functions are called with a pointer    *
*  to this structure in order to pass information important   *
*  to the callback funciton.  This is typically information   *
*  with regards to the type of access being performed on the  *
*  resource, resource number, type, etc                       *
*************************************************************/
typedef struct filterparams
{
        int type;       // Type of resource calling, RESOURCE_ANALOGIN, RESOURCE_ANALOGOUT,
                        // etc...

        int resourceNum;    // Quickstep resource number for this resource, 1 to 65536 for
                            // registers, 0 to nn analog, ...

        int mode;       // Type of access being performed, RESOURCE_READ, RESOURCE_WRITE
        int row;        // If RESOURCE_DATATABLE then the row and column values being
                        // accessed are passed
```

```
        int col;
        void *option;  // Reserved for future expansion

}FILTERPARAMS;
```

The void *option will now point to a structure:

```
typedef struct

{
        TASK *task;     // current TASK *, or NULL if not a task (CTCMON, etc)
        void *RegVal;   // same as what is passed in regVRead and regVWrite
        int type;       // same as what is passed in regVRead and regVWrite
        void *vptr;     // Actual current Variant structure pointer for read operations

} REGVINFO;
```

The 'C' API may specify any combination of registers it may wish to intercept, thus allowing for expanded function control. Based upon values read and written to specific registers, Registers 36990/36991 are reserved for QuickBuilder.

Task Control

Resource filters supply a pointer to a structure called the Task Control Block, or TASK:

```
typedef struct tcb
{
                QCODE       *step_ptr;  /* pointer to start of current step */
                STEPNUM     step_1st;   /* number of 1st step task executed */
                STEPNUM     step_num;   /* current step number */
                           STEPNUM    step_next;  /* next step number */
                UINT8       repeat;     /* repeat count of DO */
                TASK_STATE  state;      /* task state */
                RETVAL      status;     /* operation status */
                short     stepLockFlag;  // If set will not task switch when enter next
                                         // step, stays with this task
                RETVAL      stepC_APIBranch;
                              // if this is nonzero then 'taskStepBranch' was called
                              // and this is the return status to use after executing
                              // a step, typically, END_OF_STEP for branch to occur
                              // but could be a fault condition.
                short     peeridx;       /* peer.ini index */

                // BIT0 - read
                // BIT1 - write
                // BIT2 - Operand 1
                 // BIT3 - Operand 2
                // BIT4 - Store inst.
                // BIT5 - IF instruct.
                // BIT6 - MONITOR instruct.
                // BIT7  - FLAG instruct.
                // BIT32 - PERSISTENT across steps
                int       peerrules;

                /* rules for instruction currently executing */
                int         activeAccessRules;
                RETVAL      peerstatus;
                BREAKINFO  *breakInfo;
                void    *localRegs;           // local registers
                void *     qs4TaskLocal;      // Reserved QS4 storage
                void *      userCStorage;     // General userCStorage
                short     taskNum;            // Task number, base 0
                unsigned int taskHandle;
```

```
                    // every task has a unique task handle, assigned upon creation

                UINT8       wait_for[MAX_TASKS_8];  /* Each bit represents an offspring */
                } TASK;

typedef enum
{
                 TASK_UNUSED=0,  /* task is not in use */
                TASK_DONE,      /* task is complete */
                TASK_ERROR,     /* error occurred, task halted */
                TASK_PAUSED,    /* task is paused (used for single-step) */
                TASK_STOPPED,   /* task stopped */
                TASK_CREATED,   /* new task created this cycle */
                TASK_START,     /* task is started/restarted at first step */
                TASK_GOTOSTEP,  /* task is branching to new step */
                TASK_NEWSTEP,   /* task is running first pass of step */
                TASK_RUNSTEP,    /* task is running step */
                TASK_SS_START,  /* task is started/restarted at 1st step, single step
                                 active */

                TASK_SS_STEP,   /* tasks runs first pass of step, single step active */
                TASK_SS_RUN     /* task is running step, single step active */
} TASK_STATE;
```

'C' API calls are available to modify this structure in a safe manner:

createTask

Create a QS2 Task:

```
/*************************************************************************/
/*                                                                       */
/*  FUNCTION                                              RELEASE        */
/*                                                                       */
/*    createTask                                         PORTABLE C      */
/*                                                          1.1          */
/*  DESCRIPTION                                                          */
/*    Request that the Quickstep task be created and started on the      */
/*    next Quickstep execution loop.                                     */
/*                                                                       */
/*  INPUT                                                                */
/*    int stepNum - Step number to begin task execution on, 0 is first   */
/*                                                                       */
/*  OUTPUT                                                               */
/*    TASK * - NULL if no more tasks available, else pointer to TASK     */
/*             structure created, prior to execution.                    */
/*                                                                       */
/*  CALLS                                                                */
/*    Quickstep OS virtual table function pointer                        */
/*                                                                       */
/*  CALLED BY                                                            */
/*    As required by user code                                           */
/*                                                                       */
/*  RELEASE HISTORY                                                      */
/*                                                                       */
/*    DATE              NAME                      DESCRIPTION            */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
TASK *createTask(int stepNum)
{
       if (getCoreTable() != NULL)
       {
              if (coreTable->createTask != NULL)
              {
```

```
                        return((*coreTable->createTask)(stepNum));
                }
        }
        return(NULL);
}
```

getTaskFromHandle

Get a task pointer from a given task handle (task->taskHandle):

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    getTaskFromHandle                                      PORTABLE C      */
/*                                                              1.1          */
/*  DESCRIPTION                                                              */
/*    Each Task has a unique numeric handle that maps to a TASK *            */
/*    structure.  A call to this routine will return a that pointer or       */
/*    NULL if none is currently in existance.                                */
/*                                                                           */
/*  INPUT                                                                    */
/*    unsigned int handle - Unique task handle                               */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    TASK * - NULL if no task matches the passed handle, else TASK *        */
/*                                                                           */
/*  CALLS                                                                    */
/*    Quickstep OS virtual table function pointer                            */
/*                                                                           */
/*  CALLED BY                                                                */
/*    As required by user code                                               */
/*                                                                           */
/*  RELEASE HISTORY                                                          */
/*                                                                           */
/*    DATE              NAME                      DESCRIPTION              */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
TASK *getTaskFromHandle(unsigned int handle)
{
        if (getCoreTable() != NULL)
        {
                if (coreTable->getTaskFromHandle != NULL)
                {
                        return((*coreTable->getTaskFromHandle)(handle));
                }
        }
        return(NULL);
}
```

taskSetBranch

To cause a step to branch upon return from a resource filter:

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    taskSetBranch                                          PORTABLE C      */
/*                                                              1.1          */
/*  DESCRIPTION                                                              */
/*    Request that the Quickstep task that was running when this 'C' API     */
/*    was called branch to 'step' upon immediate return from the 'C' API.    */
```

```
/*    The branch will result in another task taking control prior to      */
/*    execution unless the 'taskStepLock' is invoked first.               */
/*                                                                        */
/*  INPUT                                                                 */
/*    void *task - TASK * passed to the resource filter when invoked.     */
/*    int step - Step number to branch to, 0 based, where 0 = first       */
/*                                                                        */
/*  OUTPUT                                                                */
/*    END_OF_STEP if branch is valid and successful                       */
/*    ERROR_NO_STEP if step does not exist                                */
/*    ERROR_TASK if task does not exist (null task pointer)               */
/*                                                                        */
/*  CALLS                                                                 */
/*    Quickstep OS virtual table function pointer                         */
/*                                                                        */
/*  CALLED BY                                                             */
/*    As required by user code                                            */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*                                                                        */
/**************************************************************************/
RETVAL taskSetBranch( void *task, int step);
```

taskStepLock

To lock a task such that it will let you execute the next step immediately after the current one is complete. This is valid for only a single extra step lock allowing atomicity to survive to the next step. It is assumed that this call occurs while this Task has ownership, if not the lock will have no effect:

```
/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    taskStepLock                                        PORTABLE C      */
/*                                                           1.1          */
/*  DESCRIPTION                                                           */
/*    Request that the Quickstep task that was running when this 'C' API  */
/*    was called run the next Quickstep step without switching control    */
/*    to another TASK, upon return.  Flag Lock cleared after next step    */
/*    is executed.                                                        */
/*                                                                        */
/*  INPUT                                                                 */
/*    void *task - TASK * passed to the resource filter when invoked.     */
/*                                                                        */
/*  OUTPUT                                                                */
/*    NONE                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*    Quickstep OS virtual table function pointer                         */
/*                                                                        */
/*  CALLED BY                                                             */
/*    As required by user code                                            */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*                                                                        */
/**************************************************************************/
void taskStepLock(void *task);
```

Writing any value to register 36996, (*Task Control Lock Register*), will have the same effect as a call to this function.

killTask

To kill a task:

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    killTask                                               PORTABLE C      */
/*                                                              1.1          */
/*  DESCRIPTION                                                              */
/*    Request that the Quickstep task be created and started on the          */
/*    next Quickstep execution loop.                                         */
/*                                                                           */
/*  INPUT                                                                    */
/*    void *task - TASK * Quickstep task control block to kill.              */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    SUCCESS - 0                                                            */
/*    ERROR_TASK if task does not exist (null task pointer)                  */
/*                                                                           */
/*  CALLS                                                                    */
/*    Quickstep OS virtual table function pointer                            */
/*                                                                           */
/*  CALLED BY                                                                */
/*    As required by user code                                               */
/*                                                                           */
/*  RELEASE HISTORY                                                          */
/*                                                                           */
/*    DATE              NAME                      DESCRIPTION              */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
RETVAL killTask(TASK *task)
```

Script Execution

Script commands may be executed by using the 'C' API. This is done by first allocating a Command Parser Environment (CTC_allocateCommandParser) and then using that environment to execute commands (CTC_runCommandParser). The environment structure appears as follows:

```
/*************************************************************
*  struct cmdParser - Run Script Command Environment Structure  *
*                                                               *
*  This structure is returned by the allocateCommandParser      *
*  function call and may be re-used for running Script commands.*
*************************************************************/
typedef struct cmdParser
{
        char *aszParameters;          // Line to process.
        char *m_byteArrayData;        // Scratch array to use.
        void *ftp;                    // Where to send results.
        void *lhFile;                 // File handle.
        int mode;                     // Mode of execution.
        int scriptIndex;              // Which instance we are.
        int lineCtr;                  // Line of execution in script.
        long position;                // Position of start of next line.
        char *labelName;              // Used when branch occurs,
```

```
                                        // pointer to branch label want.
        void *task;                      // typically NULL
} CMDPARSER;
```

CTC_allocateCommandParser

```
/*************************************************************************/
/*                                                                       */
/*  FUNCTION                                              RELEASE        */
/*                                                                       */
/*    CTC_allocateCommandParser                          PORTABLE C      */
/*                                                          1.0          */
/*  DESCRIPTION                                                          */
/*    This function is used to allocate an environment for running       */
/*    script commands.  It may be used for multiple commands.  Since     */
/*    a malloc call is involved it is best to reserve a pool than call   */
/*    and release after every use.                                       */
/*                                                                       */
/*  INPUT                                                                */
/*    int maxbuffer - Maximum amount of message response dumped to the   */
/*    telnet screen to copy to an internal buffer.  For example if you   */
/*    wanted an entire directory this might have to be 2048 whereas      */
/*    simple commands may only be 256.  Typically 512 is adequate and    */
/*    regardless of output will not exceed this value (buffer truncated).*/
/*    A value of 0 will allow commands to run but no result string is    */
/*    buffered.                                                          */
/*                                                                       */
/*  OUTPUT                                                               */
/*    RETVAL -                                                           */
/*      CMDPARSER * = Environment handle/pointer for additional calls to */
/*                other Script command functions.  Actually CMDPARSER    */
/*                structure pointer should you attempt to access directly*/
/*                                                                       */
/*  CALLS                                                                */
/*    Quickstep OS virtual table function pointer                        */
/*                                                                       */
/*  CALLED BY                                                            */
/*    As required by user code                                           */
/*                                                                       */
/*  RELEASE HISTORY                                                      */
/*                                                                       */
/*    DATE              NAME                      DESCRIPTION            */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
CMDPARSER *CTC_allocateCommandParser(int maxbuffer)
{
        if (getCoreTable() != NULL)
        {
                if (coreTable->allocateCommandParser != NULL)
                {
                        return((*coreTable->allocateCommandParser)(maxbuffer));
                }
        }
        return NULL;
}
```

CTC_releaseCommandParser

```
/*************************************************************************/
/*                                                                       */
/*  FUNCTION                                              RELEASE        */
/*                                                                       */
/*    CTC_releaseCommandParser                           PORTABLE C      */
```

```
/*                                                              1.0          */
/*  DESCRIPTION                                                              */
/*    This function is used to return the allocated Script Command           */
/*    environment, freeing memory usage.                                     */
/*                                                                           */
/*  INPUT                                                                    */
/*    CMDPARSER *pCmdP - Pointer returned by the CTC_allocateCommandPaser    */
/*    function call.                                                         */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    None                                                                   */
/*                                                                           */
/*  CALLS                                                                    */
/*    Quickstep OS virtual table function pointer                            */
/*                                                                           */
/*  CALLED BY                                                                */
/*    As required by user code                                               */
/*                                                                           */
/*  RELEASE HISTORY                                                          */
/*                                                                           */
/*    DATE              NAME                      DESCRIPTION              */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
void CTC_releaseCommandParser(CMDPARSER *pCmdP)
{
        if (getCoreTable() != NULL)
        {
                if (coreTable->releaseCommandParser != NULL)
                {
                        (*coreTable->releaseCommandParser)(pCmdP);
                }
        }
}
```

CTC_runCommandParser

```
/*****************************************************************************/
/*                                                                           */
/*  FUNCTION                                                  RELEASE        */
/*                                                                           */
/*    CTC_runCommandParser                                  PORTABLE C       */
/*                                                             1.0           */
/*  DESCRIPTION                                                              */
/*    This function is used to execute any desired script command.           */
/*    All responses returned to the telnet screen will be written            */
/*    directly to a memory buffer for later analysis by the caller,          */
/*    This includes all CRLF combinations, etc.  Screen output is simply     */
/*    redirected to a buffer instead of a telnet or serial diagnostic        */
/*    screen.  Most commands begin with ERROR or SUCCESS with a possible     */
/*    leading white space or CRLF combination.                               */
/*  INPUT                                                                    */
/*    CMDPARSER *pCmdP - Pointer returned by the CTC_allocateCommandPaser    */
/*    function call.                                                         */
/*                                                                           */
/*    char *cmd - Null terminated string of command to execute.              */
/*                                                                           */
/*  OUTPUT                                                                   */
/*    long - 0, if no error, else Script Error Mask (defined in              */
/*           CoreFunc.c                                                      */
/*                                                                           */
/*  CALLS                                                                    */
/*    Quickstep OS virtual table function pointer                            */
/*                                                                           */
/*  CALLED BY                                                                */
/*    As required by user code                                               */
/*                                                                           */
/*  RELEASE HISTORY                                                          */
```

```
/*                                                                          */
/*    DATE              NAME                        DESCRIPTION             */
/*                                                                          */
/*                                                                          */
/****************************************************************************/
long CTC_runCommandParser(CMDPARSER *pCmdP, char *cmd)
{
        if (getCoreTable() != NULL)
        {
                if (coreTable->runCommandParser != NULL)
                {
                        return((*coreTable->runCommandParser)(pCmdP, cmd));
                }
        }
        return SCRIPT_ERRMASK_FATAL;
}
```

CTC_getCommandParser

```
/****************************************************************************/
/*                                                                          */
/*  FUNCTION                                                 RELEASE        */
/*                                                                          */
/*    CTC_getCommandResponse                                PORTABLE C      */
/*                                                             1.0          */
/*  DESCRIPTION                                                             */
/*    This function is used to copy the command result response string      */
/*    to another buffer, length of that copied is returned.  Alternatively*/
/*    you may access the CMDPARSER structure itself.                        */
/*                                                                          */
/*  INPUT                                                                   */
/*    CMDPARSER *pCmdP - Pointer returned by the CTC_allocateCommandPaser   */
/*    function call.                                                        */
/*                                                                          */
/*    char *buf - User buffer to copy result message to, null terminated.   */
/*                                                                          */
/*  OUTPUT                                                                  */
/*    int - Number of characters copied to the user buffer, buf.            */
/*                                                                          */
/*  CALLS                                                                   */
/*    Quickstep OS virtual table function pointer                           */
/*                                                                          */
/*  CALLED BY                                                               */
/*    As required by user code                                              */
/*                                                                          */
/*  RELEASE HISTORY                                                         */
/*                                                                          */
/*    DATE              NAME                        DESCRIPTION             */
/*                                                                          */
/*                                                                          */
/****************************************************************************/
int CTC_getCommandResponse(CMDPARSER *pCmdP,char *buf)
{
        if (getCoreTable() != NULL)
        {
                if (coreTable->getCommandResponse != NULL)
                {
                        return((*coreTable->getCommandResponse)(pCmdP, buf));
                }
        }
        return 0;
}
```

Variant Internal Memory Storage (Reference Only)

Although not visible to the User unless accessed from a 'C' API level, below details how Variants are currently stored and variant.h include file definitions:

```
#define VARIANT_MAX_STRING 223        // maximum size of user string storage (add 1 for
                                      // array null)

// Bit identifiers for settings and last written variant type
#define VARIANT_INTEGER BIT0
#define VARIANT_UINTEGER BIT1
#define VARIANT_STRING BIT2
#define VARIANT_FLOAT BIT3
#define VARIANT_DOUBLE BIT4
#define VARIANT_STRING_PTR BIT5
#define VARIANT_INDIRECTION_FLAG BIT6  // Setting this bit means that the integer portion
                                       // of this is to be used to access another
                                       // register's value
#define VARIANT_NONVOLATILE BIT7   // storage structure is nonvolatile memory

// Requests for Variant access types in addition to VARIANT_INTEGER to VARIANT_DOUBLE
#define VARIANT_BASE_PROPERTIES_TYPE BIT10  // Get properties only
#define VARIANT_STORAGE_TYPE BIT11  // This is for requests of read/write not settings
#define VARIANT_CURRENT_TYPE -1       // Used as flag with VARIANT_STORAGE_TYPE to get
                                      // current Variant in its native (current) storage
                                      // format/type

// COMMANDS
#define VARIANT_CMD_NULL 0
#define VARIANT_CMD_INDIRECTION 1     // Assumes that a VARIANT_INTEGER will be written
                                      // such that a nonzero value sets the indirection
                                      // bit
#define VARIANT_CMD_CLEAR 2   // This command clears the variant to type integer,
                              // no array, value 0.

// Default precision for floats to string conversion
#define DEFAULT_VARIANT_FLOAT_PRECISION 6

// Maximum array depth allowed
#define VARIANT_ARRAY_MAX_COL_ELEMENTS 32768
#define VARIANT_ARRAY_MAX_ROW_ELEMENTS 32768

// Mode of operation
#define VARIANT_READ 0
#define VARIANT_WRITE 1

// Local registers
#define VARIANT_REG_NUMBER_LOCAL   100
#define VARIANT_REG_LOCAL_FIRST         RegisterNumber( 36001)
#define VARIANT_REG_LOCAL_LAST          RegisterNumber( 36100)

// Public Registers
#define VARIANT_REG_NUMBER_VOLATILE    600
#define VARIANT_REG_VPUBLIC_FIRST         RegisterNumber( 36101)
#define VARIANT_REG_VPUBLIC_LAST          RegisterNumber( 36700)

// NonVolatile registers
#define VARIANT_REG_NUMBER_NVOLATILE 100
#define VARIANT_REG_NVPUBLIC_FIRST         RegisterNumber( 36701)
#define VARIANT_REG_NVPUBLIC_LAST          RegisterNumber( 36800)


/*
Variant property selection registers

All Variant registers may become arrays by setting the 'Size' property a particular
```

```
variant. Variant register property access is via a Selection register, local to all
tasks.

Variant Selection Register - Register 36804.  Pointer to a Variant register (36001 to
36800)whose property to access.
Variant Index Register -  Register 36807.  Index value of Variant selected, base is 0.
Variant Size Register -   Register 36808.  Size of Variant array, write changes size,
default is 1.
Variant Increment Register - Register 36809.  0 to disable, 1 to increment, and -1 to
decrement.  Index cleared to 0 on write.  Each read or write to the actual Variant
Register will increment or decrement the index until it can access no more at which point
a fault will occur if bounds exceeded.
Variant Deletion Register -  Register 36810.  Allows the specified Variant to be deleted
NonVolatile Close Register - Register 36811.  Closes a nonvolatile variant file.
*/

#define REG_LOCAL_VARIANT_SELECTION          RegisterNumber( 36804)        // read/write
#define REG_LOCAL_VARIANT_ARRAYSIZE_COL      RegisterNumber( 36805)        // read/write
#define REG_LOCAL_VARIANT_ARRAYSIZE_ROW      RegisterNumber( 36806)        // read/write
#define REG_LOCAL_VARIANT_INDEX_COL          RegisterNumber( 36807)        // read/write
#define REG_LOCAL_VARIANT_INDEX_ROW          RegisterNumber( 36808)        // read/write
#define REG_LOCAL_VARIANT_INDIRECTION        RegisterNumber( 36809)        // read/write
#define REG_VARIANT_DELETION                 RegisterNumber( 36810)        // write
#define REG_NVARIANT_CLOSE                   RegisterNumber( 36811)        // write

/*
Task Control Lock Register

Writing any value to this register will cause the calling TASK to maintain control for
an additional step.
*/
#define REG_LOCAL_TASK_CONTROL_LOCK   RegisterNumber (36996)

/*
QS4 General resource filter use:
*/
#define REG_QS4_WRITE_FUNCTION_CALL   RegisterNumber (36990)
#define REG_QS4_READ_FUNCTION_CALL    RegisterNumber (36991)

typedef struct varHeader
{
        struct varHeader *nextCol;
        struct varHeader *nextRow;
        int iValue;
        float fValue;
        double dValue;
        unsigned char stringLen;  // length of string stored
        unsigned char settings;   // Bit 0 - int valid, 1 - string, 2 - float, 4 - double,
                                                  // 6 - indirection, 7 - nonvolatile
        unsigned char lastWrite;  // Bit 0 - int, 1 - string, 2 - float, 4 - double
        unsigned char precision;  // double to string conversion precision %.6f default
        short mallocLen;  // Length of space reserved for the string
        unsigned short indexCol;        // Col dimension index reference
        unsigned short indexRow;        // Row dimension index reference
        unsigned short arraysizeCol;    // depth of array in Col dimension, default is 1.
        unsigned short arraysizeRow;    // depth of array in Row dimension, default is 1.
} VAR_HEADER;


// Volatile Variant register
typedef struct
{
        VAR_HEADER header;
        char *sValue;
} VOL_VARIANT_REGISTER;

// Variant properties
typedef struct
{
        unsigned short indexCol;        // Col dimension index reference
        unsigned short indexRow;        // Row dimension index reference
```

```
        unsigned short arraysizeCol;  // depth of array in Col dimension, default is 1.
        unsigned short arraysizeRow;  // depth of array in Row dimension, default is 1.
} VARIANT_BASE_PROPERTIES;

// Non-volatile Variant register
typedef struct
{
        VAR_HEADER header;
        unsigned long sValue;         // File offset to string to use
        short openIdx;                // Open file index using
        unsigned short nvIndex;       // Which NV register this is, 0 base
        unsigned int cksum;           // Checksum for storage verification, make
                                      // int for alignment purposes
} NV_VARIANT_REGISTER;        // Stored in battery backed RAM.

// Universal variant storage structure
typedef struct
{
        int type;      // type of storage being used or requested
                       // If -1 on read then return current, else set to type want.
                       // On write must set to type that is stored within this structure

        // double to string conversion precision %.6f default
        // On read is what is presently set, write what want.
        unsigned char precision;

        // special flags for processing so far only
        // VARIANT_INDIRECTION_FLAG used, can be used to set property
        // in ->settings on write operation, no effect on read.  Written
        // value becomes register to reference for further operations.
        unsigned char flags;

        // 00, no operation other than read/write specified, else do defined
        // operation.  Currently have write for properties access to 'settings'
        // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
        // write value ignored.
        unsigned char cmd;

        // task number (offset in task array + 1, where 0 is 1) or handle thus usable from
        // remote or 'C' API, 4096 to 65535, set to 0 for public reg.  Task number is QS2
        // style reference, valid for 1 to N tasks (typically 96 max).  Use 'get tasks to
        // retrieve a tasks handle.
        unsigned short taskHandle;

        // this is reserved for later use and possible string length if want unsigned
        // char, 0 - 255 values, VARIANT_BYTE, future type
        unsigned short slength;

        unsigned int indexCol;   // Col dimension index reference
        unsigned int indexRow;   // Row dimension index reference
        union                  // Data that was read or has been written of 'type'
        {
                int iValue;
                unsigned int uiValue;
                float fValue;
                double dValue;
                char sValue[VARIANT_MAX_STRING+1];
        } data;
} VARIANT_STORAGE;

// TASK local register storage
typedef struct
{
        // local variant registers
        VOL_VARIANT_REGISTER *vol_variantreg[VARIANT_REG_NUMBER_LOCAL];
        // local control registers
        int variantSelectionRegister;
        int variantArraySizeRegister;
} TASK_LOCAL_REGS;
```

Note that the maximum string length is 223 bytes. Storage is allocated on an as needed basis. The last stored data type to a Variant register becomes the base upon which to convert all other accesses of a different type. For example if you store an ‘int’ and attempt to read a string then a conversion to string will result from the ‘int’. If the last storage was a double, the double would be used for any conversions.

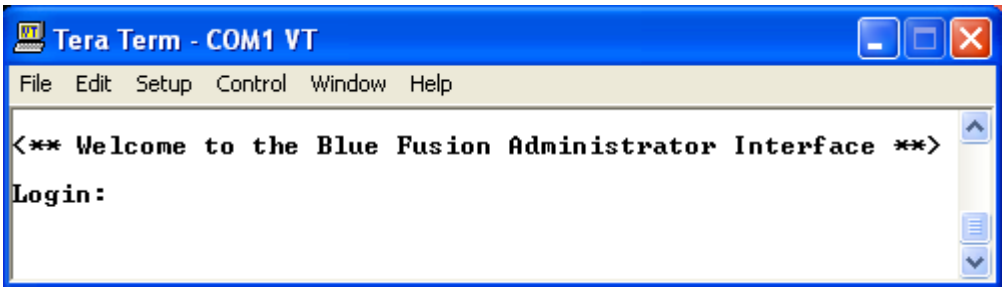
CHAPTER 7

Serial Port Administrative Functions

All serial ports monitor for a special character sequence while idle and in the default protocol mode (CT Binary). This screen is identical to that used by telnet. This section will only detail the differences.

Serial Admin Access

Connection: When connected to a serial port you may enter the telnet command mode by sending a ^A ESC ESC (0x01, 0x1b, 0x1b). The normal login request will then appear as does over telnet. All screen prompts and commands are then as in telnet/tcp. Send a ^A or a 'quit' command to exit and restore the normal use of the serial port (typically binary protocol by default). The default communications port setting, as shipped from the factory, is 19.2K baud, 8 data bits, 1 stop bit, no parity, no flow control.



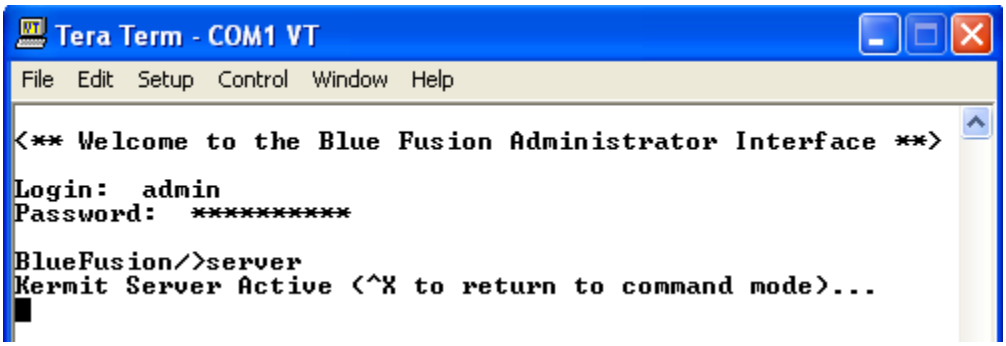
Reference the "Remote Administrative Guide" for detailed operation.

File Transfers

Kermit is an industry standard file transfer protocol supported on virtually all computers and operation systems. A public domain download is available from Columbia University:

http://www.columbia.edu/kermit/

The controller supports the Kermit file transfer protocol and can assume the role of a kermit server. The terminal session command 'server' is used to enter Kermit server mode, ^X to abort and restore terminal session.



The following Kermit commands are supported:

1. Get a file.
2. Send a file.
3. remote delete [path/file]
4. remote cd [path]
5. remote pwd (return current path)
6. remote who (return 5300 model, serial number, and ip address).
7. remote mkdir [path]
8. remote rmdir [path]
9. remote host EXISTS [path/file]
10. remote host FILESIZE [path/file]

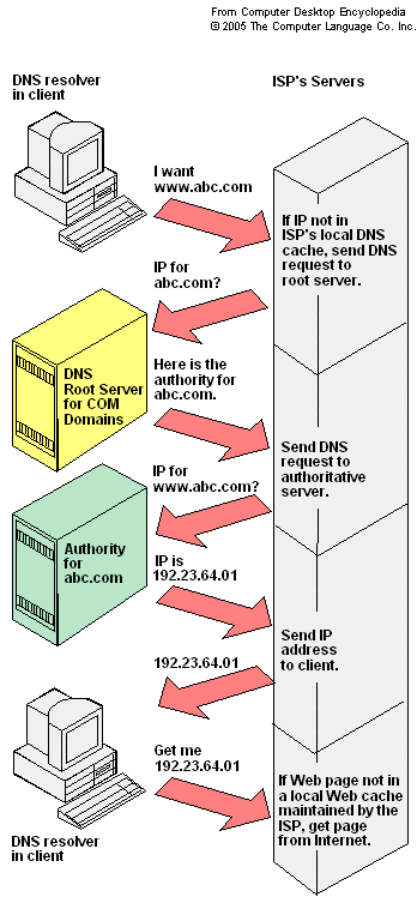
Most of the above commands, except 1 & 2 are available from the normal telnet terminal session using standard script commands. Up to 1K packet size is supported.

As with 'ftp', dropping a file on the root directory for automatic execution/loading is fully supported.

CHAPTER 8

DNS Support

(DNS - **D**omain **N**ame **S**ystem) A system for converting host names and domain names into IP addresses on the Internet or on local networks that use the TCP/IP protocol. For example, when a Web site address is given to the DNS either by typing a URL in a browser or behind the scenes from one application to another, DNS servers return the IP address of the server associated with that name.



DNS & 5300

The 5300 controller is capable of using DNS to resolve names used in place of IP addresses. The actual DNS IP address used will be that returned by the DHCP server, or when using static IP Addresses, 20128 to 20131 allow the setting of a static DNS server reference.

DNS IP resolution is supported anywhere IP addresses are used within scripts. Both names and IP '.' nomenclature may be intermixed within commands.

Script commands which support DNS:

ftpconnect <host name or IP Address>
set quicklink connections <host name or IP Address>, …
dnslookup <host name>
dnsRlookup
dnsRlookup <Reg #> <host name>
dnsRlookup 5 www.ctc-control.com
Assuming this resolved to 12.40.53.10, this would store the following:
Register 5 = 12
Register 6 = 40
Register 7 = 53
Register 8 = 10

CHAPTER 9

CTNet Variant Interface

The CTNet 32 bit communications DLL has been enhanced to support access to variant registers. Updates are available on the CTC web site, www.ctc-control.com . The DLL may be invoked from numerous Windows based programming languages. Discussion here is limited to its use with Variants and Visual Basic 6®. Reference the included Ctccomm32v2.bas file for most of the definitions described below.

Interface Definitions

VB6 DLL Function Definitions

Visual Basic may be used to interface with Variants using the standard CTC 32 bit Communications DLL. The definitions file has been expanded as is detailed below.

Constants

CT_VARIANT_??? defines the 'type' of data contained within the CT_VARIANT structure that is to be returned or is being written with the function call.

```
'VB6 Variant types to return and write
Global Const CT_VARIANT_INTEGER As Integer = 1
Global Const CT_VARIANT_UINTEGER As Integer = 2
Global Const CT_VARIANT_STRING As Integer = 4
Global Const CT_VARIANT_FLOAT As Integer = 8
Global Const CT_VARIANT_DOUBLE As Integer = 16
Global Const CT_MAX_STRING_SIZE As Integer = 223

// 'C' Variant types
#define CT_VARIANT_INTEGER 1
#define CT_VARIANT_UINTEGER 2
#define CT_VARIANT_STRING 4
#define CT_VARIANT_FLOAT 8
#define CT_VARIANT_DOUBLE 16
```

```
#define CT_MAX_STRING_SIZE 223  // Maximum string size
```

Structure – CT_VARIANT

The CT_VARIANT user type structure is a universal interface used to pass data back and forth to variants. It allows data to be represented as a Long (32 bit int), Float (32 bits), Double (64 bits), and a byte array of ascii characters of length slength.

```
' VB6 Register Structure definition
Type CT_VARIANT
    vRegister As Long          'Register desired
    type As Long               'Format want results returned in
    precision As Long          'Precision desired for floating point conversions
    flags As Long              'Defined flags, 0 for normal, (indirection, etc.)
    cmd As Long                'Special commands, 0 for normal operation
    taskHandle As Long         'Alternate task handle for local task register
                               'access, 0 = default public
    slength As Long            'Length of bytes returned in stringVar, not
                               'including null
    indexCol As Long            'Column selection, base 0
    indexRow As Long            'Row  selection base 0
    LongIntVar As Long         '32 bit signed integer storage
    FloatVar As Single         '32 bit float
    DoubleVar As Double        '64 bit double in Microsoft format
    stringVar(1 To 224) As Byte 'null terminated ASCII string of bytes
End Type

'C' Structure definition:
typedef struct
{
       long lRegister;//Register desired
       long type;     // Format want results returned in
       long precision;// Precision desired for floating point conversions
       long flags;    // Defined flags, 0 for normal, (indirection, etc.)
       long cmd;      // Special commands, 0 for normal operation
       long taskHandle; // task number or handle, 0 for public registers
       long slength;  // Length of bytes returned in stringVar, not include
                     // null
       long indexCol; // Column selection, base 0
       long indexRow; // Row selection base 0
       long LongVar;  // 32 bit signed integer storage
       float FloatVar; // 32 bit float
       double DoubleVar; // 64 bit double in Microsoft format

      // null terminated ASCII string of bytes
       unsigned char StringArray[VARIANT_MAX_STRING+1];
} DLL_VARIANT_STORAGE;
```

Structure – CT_VARIANT_BLOCK…

The CT_VARIANT_BLOCK… user type structure is a universal interface used to quickly read arrays of integers, floats and double in a single call. When using Ethernet up to 346 integer/floats or 173 doubles may be read per packet request, 115 if random read (50 integer/floats for a serial port connection, 25 doubles, 16 if random read). Upon return 'slength' will contain the number actually returned in case it was less that what requested. Note that VB6 does not support a Float type, only Double.

```
' VB6 Register Structure definition
Type CT_VARIANT_BLOCK_INTEGER
    vRegister As Long      'Register desired
    type As Long           'Format want results returned in
```

```
    precision As Long      'Precision desired for floating point conversions
    flags As Long          'Defined flags, 0 for normal, (indirection, etc.)
    cmd As Long            'Special commands, 0 for normal operation
    taskHandle As Long     'Alternate task handle for local task register
                               'access, 0 = default public
    slength As Long        'Length of bytes returned in stringVar, not
                               'including null
    indexCol As Long       'Column selection, base 0
    indexRow As Long       'Row  selection base 0
    numAccess As Long      'Number of items to read
    rowInc As Long         'Amount to increment row by, 0 is just read the columns
    colInc As Long         'Amount to increment column by, 0 is just read first
    arraysizeCols As Long  'Used on write operation, -1 do not expand existing
                           'columns, else columns desired.  Rows will
                           ' automatically grow as needed
    ibValue(1 To 346) As Long
End Type

Type CT_VARIANT_BLOCK_DOUBLE
    vRegister As Long      'Register desired
    type As Long           'Format want results returned in
    precision As Long      'Precision desired for floating point conversions
    flags As Long          'Defined flags, 0 for normal, (indirection, etc.)
    cmd As Long            'Special commands, 0 for normal operation
    taskHandle As Long     'Alternate task handle for local task register
                               'access, 0 = default public
    slength As Long        'Length of bytes returned in stringVar, not
                           'including null
    indexCol As Long       'Column selection, base 0
    indexRow As Long       'Row  selection base 0
    numAccess As Long      'Number of items to read
    rowInc As Long         'Amount to increment row by, 0 is just read the columns
    colInc As Long         'Amount to increment column by, 0 is just read first
    arraysizeCols As Long  'Used on write operation, -1 do not expand existing
                           'columns, else columns desired.  Rows will
                           ' automatically grow as needed
    dbValue(1 To 173) As Double
End Type


'C' Structure definition for CtGetVRegisterBlock():

// Limitations for UDP/TCP Ethernet access
#define MAX_VARIANT_BLOCK_32BITS      346            // 346 integers
#define MAX_VARIANT_BLOCK_64BITS      173    // 173 doubles
#define MAX_VARIANT_RANDOM_BLOCK      (MAX_VARIANT_BLOCK_32BITS/3)  // 115 items
// Limitations for serial port access
#define MAX_VARIANT_BLOCK_32BITS_SERIAL 50
#define MAX_VARIANT_BLOCK_64BITS_SERIAL (MAX_VARIANT_BLOCK_32BITS_SERIAL/2) //25
#define MAX_VARIANT_RANDOM_BLOCK_SERIAL (MAX_VARIANT_BLOCK_32BITS_SERIAL/3) //16

typedef struct
{
       long reg;   // may at some point reserve the upper 16 bits of this integer
                   // for 'type' req.
       long row;
       long col;
} VARIANT_ACCESS_REQUEST;

typedef struct
{
 long lRegister;   // Desired register
 long type;       // Type of storage being used or requested
 long precision;   // Double to string conversion precision %.6f default
                        // On read is what is presently set, write what want.
 long flags;   // Special flags for processing so far only
               // VARIANT_INDIRECTION_FLAG used, can be used to set property
               // in ->settings on write operation, no effect on read.  Written
               // value becomes register to reference for further operations.
 long cmd; // 00, no operation other than read/write specified, else do defined
       // operation.  Currently have write for properties access to 'settings'
```

```
        // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
                        // write value ignored.
      long taskHandle;  // task number or handle, 0 for public registers
      long slength;   // string length
      long indexCol;  // Column dimension index reference, unsigned short on controller
      long indexRow;  // Row dim index reference, unsigned short on controller
      long numAccess; // Number of items to read
      long rowInc;  // row incrementor
      long colInc;  // column incrementor
      long arraysizeCols;   // Used on write operation, -1 do not expand existing
                        // columns, else columns desired.  Rows will automatically
                        // grow as needed
      // Results type requested returned in relevant storage below:
      union
      {
            int ibValue[MAX_VARIANT_BLOCK_32BITS];          // 346 items
            float fbValue[MAX_VARIANT_BLOCK_32BITS];   // 346 items
            double dbValue[MAX_VARIANT_BLOCK_64BITS];  // 173 items
            union
            {
                  int ibValue;
                  float fbValue;
                  double dbValue;
            } random[MAX_VARIANT_RANDOM_BLOCK];
            VARIANT_ACCESS_REQUEST request[MAX_VARIANT_RANDOM_BLOCK];
      } block;
     } DLL_VARIANT_STORAGE_BLOCK;
```

DLL Function Declarations

For available function declarations reference the Ctccom32v2.bas file for VB6 or the Ctccom32v2.h file for 'C', available with CTCCom32 communications dll:

http://www.ctc-control.com/customer/downloads/Ctccom32/Ctccom32v2WinPCap.zip

DLL Functions

CtGetVProperties

This function is used to retrieve the size of a variant as well as the current default precision (double to string conversion accuracy).

```
//  FUNCTION: CtGetVProperties(ULONG lConnectID, ULONG lIndex, ULONG *pSize_Column,
//            ULONG *pSize_Row, ULONG *pPrecision)
//
//  PURPOSE:
//    Get the properties of a variant register.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    ULONG lIndex - Desired register whose properties to get
//    ULONG *pSize_Column - Number of columns found will be stored here
//    ULONG *pSize_Row - Number of rows found will be stored here
//    ULONG *pPrecision - Floating point precision currently active, default
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVProperties(ULONG lConnectID, ULONG lIndex,
                        ULONG *pSize_Column, ULONG *pSize_Row, ULONG *pPrecision);
```

Example:

```
Dim rows As Long
Dim columns As Long
Dim precision As Long
Dim reg As Long

        reg = 36101  'Get this variant register
        If CtGetVProperties(ConnectID, reg, columns, rows, precision) = SUCCESS Then
           ‘ have properties of register 36101 now
        Else
           ‘Check the error
           If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
                ‘ display error code…
           Else
                ‘unknown error
            End If
        End If
```

CtGetVRegister

This function can be used to read any register in the format desired, both variant and normal integer registers. Only variant registers may be referenced as an array.

```
//  FUNCTION: CtGetVRegister(ULONG lConnectID, DLL_VARIANT_STORAGE *variant)
//
//  PURPOSE:
//    Get the variant register contents.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE *variant - Structure to get information from and forward to
//                                   controller as well as return results.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegister(ULONG lConnectID, ULONG lIndex,
                        DLL_VARIANT_STORAGE *variant);
```

Example:

```
Dim Var As CT_VARIANT

        Var.vRegister = 36101       ‘set register want in structure to 36101
        ‘ Clear out the Variant storage structure.
        Var.cmd = 0
        Var.flags = 0
        Var.taskHandle = 0
        Var.sLength = 0
        Var.indexCol = 0        ‘set want first element [0][0], [row][column]
        Var.indexRow = 0
        Var.precision = 6      ‘this is only useful for string operations but set anyways
        Var.type = -1           ‘for first call just get whatever last type returned was, default

        If CtGetVRegister(ConnectID, Var) = SUCCESS Then
           ‘Got default type of variant, ‘type’ is set to what was there…
        Else
           ‘Check the error
           If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
                ‘ display error code…
           Else
                ‘unknown error
           End If
        End If
        Var.type = CT_VARIANT_INTEGER         ‘Request Integer type returned
        If CtGetVRegister(ConnectID, Var) = SUCCESS Then
           ‘Var.LongIntVar is now has valid data in it
```

```
Else
    'Check the error
    If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
            ' display error code…
    Else
            'unknown error
    End If
End If

Var.type = CT_VARIANT_DOUBLE        'Request Double type returned
If CtGetVRegister(ConnectID, Var) = SUCCESS Then
   'Var.DoubleVar is now has valid data in it
Else
    'Check the error
    If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
            ' display error code…
    Else
            'unknown error
    End If
End If

Var.type = CT_VARIANT_FLOAT         'Request 32 bit Float returned
If CtGetVRegister(ConnectID, Var) = SUCCESS Then
   'Var.FloatVar is now has valid data in it
Else
    'Check the error
    If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
            ' display error code…
    Else
            'unknown error
    End If
End If
Var.type = CT_VARIANT_STRING        'Request ASCII Bytes returned
If CtGetVRegister(ConnectID, Var) = SUCCESS Then
   'Var.stringVar now has valid data of length Var.slength (bytes)
Else
    'Check the error
    If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
            ' display error code…
    Else
            'unknown error
    End If
End If
```

CtGetVRegisters

This function can be used to read any number of registers in a single call. An array of preconfigured DLL_VARIANT_STORAGE blocks is passed and updated by the DLL. The interaction with the controller is identical to the single CtGetVRegister call. A unique request is sent to the controller for each array entry.

```
//  FUNCTION: CtGetVRegisters(ULONG lConnectID, ULONG numEntries,
//            DLL_VARIANT_STORAGE *variantList)
//
//  PURPOSE:
//    Get the variant registers listed in the array, they may be all one variant or
different variants.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    ULONG numEntries - Number of items in variant array
//    ULONG *doneEntries - Number of entries updated, sequentially, useful to determine
//                         which failed on.
//    DLL_VARIANT_STORAGE *variant[] - Array Structure of numEntries size to return
//                       results to, all should be initialized for proper indexX/indexY,
//                       register desired, type, etc...
//
```

```
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegisters(ULONG lConnectID, ULONG numEntries,
                       ULONG *doneEntries, DLL_VARIANT_STORAGE *variant);
```

CtGetVRegisterBlock…

This function can be used to read a block of variant registers that are stored as VARIANT_INTEGER, VARIANT_DOUBLE, or VARIANT_FLOAT data types. All cells read must be of that type or the read will abort.

```
//  FUNCTION: CtGetVRegisterBlock(ULONG lConnectID, DLL_VARIANT_STORAGE_BLOCK *variant)
//
//  PURPOSE:
//    Get the variant register contents as an Array of desired type
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK *variant - Structure to request and return integer/double
//              float union array data from controller.  Typically for 'C' and C++
//              that can handle unions
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegisterBlock(ULONG lConnectID, DLL_VARIANT_STORAGE_BLOCK
        *variant);

//  FUNCTION: CtGetVRegisterBlockInteger(ULONG lConnectID,
//              DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant)
//
//  PURPOSE:
//    Get the variant register contents as an Integer Array
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant - Structure to request and return
//              integer array data from controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegisterBlockInteger(ULONG lConnectID,
        DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant);

//  FUNCTION: CtGetVRegisterBlockFloat(ULONG lConnectID,
//                    DLL_VARIANT_STORAGE_BLOCK_FlOAT *variant)
//
//  PURPOSE:
//    Get the variant register contents as a Float Array
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_FLOAT *variant - Structure to request and return float
//              array data from controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegisterBlockFloat(ULONG lConnectID,
       DLL_VARIANT_STORAGE_BLOCK_FLOAT *variant);

//  FUNCTION: CtGetVRegisterBlockDouble(ULONG lConnectID,
//              DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant)
//
//  PURPOSE:
//    Get the variant register contents as a Double Array
```

```
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant - Structure to request and return double
//              array data from controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtGetVRegisterBlockDouble(ULONG lConnectID,
        DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant);

//FUNCTION CtGetVRegisterRandomBlock(ULONG lConnectID,DLL_VARIANT_STORAGE_BLOCK *variant)
//
//  PURPOSE:
//    Get the variant register contents.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK *variant - Structure to request and return random
//           integer/double union array data from controller.  Typically for 'C' and C++
//           that can handle unions.  'numAccess' defines the number of entries,'request'
//           array should have an entry for each item to access.  'type' is the type of
//           variant desired
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG CtGetVRegisterRandomBlock(ULONG lConnectID,
       DLL_VARIANT_STORAGE_BLOCK *variant);

//  FUNCTION: CtGetVRegisterRandomBlockDouble(ULONG lConnectID,
//                                  DLL_VARIANT_STORAGE_BLOCK_DOUBLE_RANDOM *variant)
//
//  PURPOSE:
//    Get the random variant register contents.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK *variant - Structure to request and return random
//           integer/double union array data from controller.  Typically for 'C' and C++
//           that can handle unions.  'numAccess' defines the number of entries,'request'
//           array should have an entry for each item to access.  All data is returned as
//           double.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//

ULONG _stdcall CtGetVRegisterRandomBlockDouble(ULONG lConnectID,
       DLL_VARIANT_STORAGE_BLOCK_DOUBLE_RANDOM *variant)
```

Example: Read random variants; [0][0], 36301[0][1], 36301[0][2], 36301[1][0], and 36302[0][0] as doubles. On return VarRB.slength contains the actual number read.

```
Global Const CT_VARIANT_DOUBLE As Integer = 16
Dim VarRB As CT_VARIANT_BLOCK_RANDOMDOUBLE

‘ Some general clearing of the structure, needed so not using any extended features
 VarRB.cmd = 0
 VarRB.flags = 0
 VarRB.indexCol = 0
 VarRB.indexRow = 0
 VarRB.colInc = 0
 VarRB.rowInc = 0
 VarRB.arraysizeCols = 0
 VarRB.taskHandle = 0
```

```
VarRB.precision = 6  'float/double precision desired
VarRB.slength = 0  'clear number of variants read so know on return
VarRB.type = CT_VARIANT_DOUBLE  'Desired type to read
VarRB.numAccess = 5    'number of variants to read

' Set to read 36301[0][0], 36301[0][1], 36301[0][2], 36301[1][0], and 36302[0][0]
VarRB.request(1).register = 36301
VarRB.request(1).col = 0
VarRB.request(1).row = 0

VarRB.request(2).register = 36301
VarRB.request(2).col = 1
VarRB.request(2).row = 0

VarRB.request(3).register = 36301
VarRB.request(3).col = 2
VarRB.request(3).row = 0

VarRB.request(4).register = 36301
VarRB.request(4).col = 0
VarRB.request(4).row = 1

VarRB.request(5).register = 36302
VarRB.request(5).col = 0
VarRB.request(5).row = 0

If CtGetVRegisterRandomBlockDouble(ConnectID, VarRB) = SUCCESS Then
   'was successful, VarRB.dbValue(…) array contains values read
   'VarRB.slength is the number of items read
   'Any undefined variant will return a 0
   'Use only the row index for a vector, single dimension array, leaving col at 0
Else
   'failed
End If
```

```
//  FUNCTION: CtSetVRegisterBlock(ULONG lConnectID, DLL_VARIANT_STORAGE_BLOCK *variant)
//
//  PURPOSE:
//    Set the variant register contents as an Array of desired type
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK *variant - Structure to setinteger/double/float
//              union array data in controller.  Typically for 'C' and C++
//              that can handle unions
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegisterBlock(ULONG lConnectID, DLL_VARIANT_STORAGE_BLOCK
        *variant);

//  FUNCTION: CtSetVRegisterBlockInteger(ULONG lConnectID,
//              DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant)
//
//  PURPOSE:
//    Set the variant register contents as an Integer Array
//
//  PARAMETERS:
```

```
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant - Structure to set
//              integer array data in controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegisterBlockInteger(ULONG lConnectID,
        DLL_VARIANT_STORAGE_BLOCK_INTEGER *variant);

//  FUNCTION: CtSetVRegisterBlockFloat(ULONG lConnectID,
//                    DLL_VARIANT_STORAGE_BLOCK_FlOAT *variant)
//
//  PURPOSE:
//    Set the variant register contents as a Float Array
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_FLOAT *variant - Structure to set float
//              array data in controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegisterBlockFloat(ULONG lConnectID,
       DLL_VARIANT_STORAGE_BLOCK_FLOAT *variant);

//  FUNCTION: CtSetVRegisterBlockDouble(ULONG lConnectID,
//              DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant)
//
//  PURPOSE:
//    Set the variant register contents as a Double Array
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant - Structure to set double
//              array data in controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegisterBlockDouble(ULONG lConnectID,
        DLL_VARIANT_STORAGE_BLOCK_DOUBLE *variant);
```

Example: Read 101 VARIANT_DOUBLE from register 36201. On return VarB.slength contains the actual number read.

Dim VarB As CT_VARIANT_BLOCK_DOUBLE

```
VarB.vRegister = 36201
 VarB.cmd = 0
 VarB.flags = 0
 VarB.indexX = 0
 VarB.indexY = 0
 VarB.precision = 6
 VarB.slength = 0
 VarB.type = -1
 VarB.taskHandle = 0
 VarB.numAccess = 101
 VarB.colInc = 1
 VarB.rowInc = 1
 VarB.dbValue(1) = 0

  If CtGetVRegisterBlockDouble(ConnectID, VarB) = SUCCESS Then
        'VarB.dbVAlue(1…) now has valid data in it of VarB.slength items.
 Else
       'Error occured
 End If
```

CtSetVRegister

This function can be used to write any register in the format desired, both variant and normal integer registers. Only variant registers may be referenced as an array.

```
//  FUNCTION: CtSetVRegister(ULONG lConnectID, DLL_VARIANT_STORAGE *variant)
//
//  PURPOSE:
//    Set the variant register contents.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    DLL_VARIANT_STORAGE *variant - Structure to get information from and forward to
//                         controller.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegister(ULONG lConnectID, DLL_VARIANT_STORAGE *variant);
```

Example:

```
Dim Var As CT_VARIANT
Dim reg As Long

        reg = 36101             'Read Variant register 36101
        Var.vRegister = reg
        ' Clear out the Variant storage structure.
        Var.cmd = 0
        Var.taskHandle = 0
        Var.sLength = 0
        Var.flags = 0
        Var.indexCol = 0
        Var.indexRow = 0
        Var.precision = 6

        Var.DoubleVar = 45.6745
        Var.type = CT_VARIANT_DOUBLE
        If CtSetVRegister(ConnectID, Var) = SUCCESS Then
           'it worked…
        Else
           'Check the error
           If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
                   ' display error code…
           Else
                   'unknown error
           End If
        End If
        Var.DoubleVar = 0
        'Read back
        If CtGetVRegister(ConnectID, Var) = SUCCESS Then
           'Var.DoubleVar should have value wrote in it
        Else
           'Check the error
           If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
                   ' display error code…
           Else
                   'unknown error
           End If
        End If
```

CtSetVRegisters

This function can be used to write any number of registers in a single call. An array of preconfigured DLL_VARIANT_STORAGE blocks is passed and updated by the DLL. The interaction with the controller is identical to the single CtSetVRegister call. A unique request is sent to the controller for each array entry.

```
//  FUNCTION: CtSetVRegisters(ULONG lConnectID, ULONG numEntries, ULONG *doneEntries,
//            DLL_VARIANT_STORAGE *variant)
//
//  PURPOSE:
//    Set the variant register contents.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    ULONG numEntries - Number of items in variant array
//    ULONG *doneEntries - Number of entries updated, sequencially, useful
//                         to determine which failed on
//    DLL_VARIANT_STORAGE *variant - Structure to get information from and
//                        forward to controller as well as return results.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtSetVRegisters(ULONG lConnectID,  ULONG numEntries,
                       ULONG *doneEntries, DLL_VARIANT_STORAGE *variant);
```

CtRunCommand

This function can be used to issue script commands to the controller. An array of ASCII bytes (cmd) is passed to the function of length, cmdLength. The results character stream, as it would appear to telnet (including CR/LF) will appear in the 'result' buffer of length, resultLength. No terminating NULL is supplied.

```
//  FUNCTION: CtRunCommand(ULONG lConnectID, char *cmd, ULONG cmdLength, char *result,
ULONG *resultLength)
//
//  PURPOSE:
//    Run the desired script command, returning results.
//
//  PARAMETERS:
//    ULONG lConnectID - Connection ID from a previous open call.
//    BYTE *cmd - command string to execute.
//    ULONG cmdLength - length of command string.
//    BYTE *result - location to store message results.
//    ULONG *resultLength - location to store returned message length.
//
//  RETURN VALUE:
//    Returns SUCCESS or FAILURE
//
DLLEXPORT ULONG WINAPI CtRunCommand(ULONG lConnectID, BYTE *cmd, ULONG cmdLength,
                       BYTE *result, ULONG *resultLength);
```

Example:

```
Dim cmdBytes() As Byte
Dim cmdLength As Long
Dim cmdResults() As Byte
Dim resultLength As Long
Dim myCmd As String
Dim resultString As String
Dim cterr As CT_ERR_INFO

    resultLength = 0
    'Setup the script command desired
    myCmd = "get versions"
    'Convert to ASCII bytes
    cmdBytes = StrConv(myCmd, vbFromUnicode)
    cmdLength = UBound(cmdBytes) + 1
    ReDim cmdResults((CT_MAX_STRING_SIZE) - 1)

    If CtRunCommand(ConnectID, cmdBytes(0), cmdLength, cmdResults(0), resultLength) = SUCCESS Then
       'Adjust byte buffer so can do conversion to string
```

```
    'Convert to a String
    resultString = StrConv(cmdResults, vbUnicode)
    resultString = Left(resultString, resultLength)

    'Have result in “resultString” now so analyze it as desired…
Else
    If CtGetErrorInfo(ConnectID, cterr) = SUCCESS Then
        ‘ display error code…
    Else
       ‘unknown error
    End If
End
```

Blank

CHAPTER

10

CTNet Binary Protocol (Server Interface)

This section discusses the CTNet Binary Protocol, at the packet level, as is supported by the controller. The CTNet binary protocol is a high-speed, protocol that has checksum and error reporting capabilities. It is used in cases where data integrity, response time, and processing time are the major criteria. Data transmission is fast for the following reasons:

- Both the commands and data are represented in binary form instead of ASCII.
- The information density is higher and fewer characters are transmitted during large data transfers.
- The controller can use the data "as is" and does not have to perform binary to ASCII conversion.

Therefore use of CTNet results in very short execution times. Note that CTNet use to be non-routable (2700 with 2217 Ethernet controllers). Non-routable protocols do not contain a networking layer (IP stack), so they cannot cross a router and are limited to local subnets or intranets.

Non-routable CTNet uses a node number in place of an IP address. This node number is defined by writing to Register 20000. You can also determine the node number by reading the value in Register 20000. Set this value within the *_startup.ini* file by defining the CTNET_DEVICENODE parameter.

Provisions have been made to extend the CTNet protocol by encapsulating it in a UDP/TCP packet. In this case the IP address becomes the destination and Register 20000 is ignored. Port 3000 is for UDP and port 6000 for TCP connections. UDP/TCP is fully routable. Refer to the last section of this chapter for how to encapsulate. In short the discussion that follows fully applies to the encapsulated packet. Serial port communications is also supported for all CTNet packets, again, Register 20000 does not apply in that case either since only point to point communications is supported.

Binary Protocol

The CTC Binary Protocol may be used to communicate with the 5200 controller via serial ports or a network connection. Regardless of the mode used the basic message layer is the same. On a network the serial port data is simply encapsulated as required. Most users will not require this section and should only refer to the DLL available for use under Windows 2000/XP. This DLL is discussed in detail within the *"CTC 32-bit Communications Functions Reference Guide"*, available at www.ctc-control.com for download. The CTC Binary Protocol is somewhat more difficult to use than something like the ASCII Protocol, but it can significantly reduce the time required to transfer large blocks of data between a computer and controller and is useful in more demanding applications. The protocol is more efficient, because:

- Both the commands and data are represented in binary form instead of ASCII. The information density is higher and, for large data transfers, fewer characters need to be transmitted.
- The controller does not have to convert the data from ASCII to binary before using it. This results in shorter execution times. Since the computer does not have to convert the data to ASCII, there also may be a significant time savings in the execution of the computer program (the time savings varies between different computer languages).

Protocol Framing

To select the CTC Binary Protocol, the first character of the command must be a binary 1 (Ø1H). The controller interprets the rest of the command according to the binary protocol. Use of an ASCII character, on the serial port, will result in the ASCII Protocol being used.

The protocol uses the following format to send messages to and from the controller:

<(Ø1H)> Specifies CTC binary protocol.
<length (1 byte)> Specifies packet length to follow. Packet length is defined as n data bytes + 2 (checksum and 0xff).
<data (n bytes)> Consists of function (command) code(s) plus relevant data. For function code and data descriptions, see the section on Binary Protocol Commands.
<checksum> Consists of the complement of the modulo-256 sum of data bytes. This value, when added to the modulo-256 sum of the data packet bytes, equals ØFFH. You can calculate the checksum by adding the data packet bytes and complementing the resulting sum.

```
//
// Generate a checksum for a packet
// Parameters:  p – pointer to start of data section
//              len – length of data only section (not length, checksum or 0xff)
// Returns:  <checksum>
```

```
unsigned char  Packet_Check(unsigned char * p, int len)
{
  unsigned int c = 0;
  int i;

        for( i=0; i<len; i++)
                c = (c + *(p + i)) & 255;
        return( (char)~c);
}
```

<FFH> Required by binary protocol; last byte of packet must be ØFFH. When the controller receives a binary packet, it counts out the number of bytes specified by the packet length. If the last byte is not ØFFH, it returns an error message.

Return communications from the controller to the computer use the same general format, with one exception. The controller does not transmit a leading (Ø1H) byte, since the original message was transmitted using the CTC binary protocol. If the command sent to the controller does not require data from the controller in the return message, the controller sends an acknowledge message like the one shown below:

<03H)> Specifies packet length to follow. Packet length is defined as n data bytes + 2.
<(64H)> Contains the acknowledge code; equal to decimal 100.
<9BH> Is the value of the checksum of the acknowledge code.
<FFH> Required by binary protocol; last byte of packet must be ØFFH.

When the packet sent to the controller is not correct, it transmits a not acknowledged code. This may happen when the checksum does not calculate correctly or when the last byte of the packet is not ØFFH. A message containing a not acknowledged code is similar to the one shown below:

<03H)> Specifies packet length to follow. Packet length is defined as n data bytes + 2.
<(65H)> Contains the not acknowledged code; equal to decimal 101.
<9AH> Is the value of the checksum of the not acknowledged code.
<FFH> Required by binary protocol; last byte of packet must be ØFFH.

When the format of the message is correct, but the controller cannot execute the command, it sends other error codes. For error code descriptions, see the section on *Binary Protocol Commands*. The following example shows how to create a command in correct format for the CTC binary protocol. It sets flag 4 in the controller.

1. *Send the following command:*

   Ø1H,Ø5H,13H,Ø3H,FFH,EAH,FFH
   Where:
   Ø1H Is the first byte and identifies the packet as using the CTC binary protocol.
   Ø5H Is the second byte and represents the length of the packet.

13H Is the third byte and contains the function code for a change flag command.
Ø3H Is the fourth byte and specifies flag 4. Flags 1 through 32 are represented as ØØH through 1FH, and Ø3H specifies flag 4.
FFH Is the fifth byte and specifies the new state of the flag. FFH represents SET and ØØH represents CLEAR.
EAH Is the sixth byte and contains the checksum value.
ØFFH Is the seventh and last byte of the packet and signals the end of the message.

2. *To acknowledge the message, the controller sends the following response:*

Ø3H,64H,9BH,FFH
Where:
Ø3H Is the first byte and specifies the packet length
64H Is the second byte and contains the acknowledge code (decimal 100)
9BH Is the third byte and contains the checksum value of third byte
FFH Is the fourth and last byte and signals the end of the message.

Binary Protocol Error Responses

When the controller cannot execute the data transmission from the computer, the controller responds with an error code indicating the nature of the fault. The error code is transmitted using the following format:

Ø3H Packet length.
Error code Error code, see list below.
Checksum The checksum is the complement of the previous byte.
FFH Last byte in packet; signals the end of the message.

Possible error codes are:

64H No error (acknowledgment of transmission
65H Checksum error, or end of packet <> FFH
66H Illegal register number specified
65H Value out of range, for example, input number not present in controller

Binary Protocol Commands

Each CTC binary protocol command has specific format. This section lists the commands and describes their format. The command descriptions also list the following information:

- The type of command
- Format of command sent to the controller
- Format of the controller's response

Not all Control Technology controllers support all of these commands. Contact Control Tech customer support if you have any questions about which of these commands you can use, or if you have any difficulty implementing a command. The following table lists the commands and the controllers which support the command.

Binary Protocol Commands (controller response is command + 1)	
Register and Flag Access Commands	
9	Read a register
11	Change a register
17	Read a Flag
19	Change a Flag
75	Read a bank of 50 registers
77	Read a bank of 16 registers
87	Request random registers from list (CTServer)
Variant Commands	
91	Get properties
93	Read a variant
95	Change a variant
109	Read a variant array block (consecutive)
111	Write a variant array block (consecutive)
113	Read a block of variants, randomly
Input/Output Access Commands	
15	Read a bank of 8 inputs
21	Read a bank of 8 outputs
25	Selectively modify first 128 outputs
29	Read an analog input
31	Read an analog output
33	Change an analog output
71	Get 32 analog inputs
73	Get 32 analog outputs
79	Read a bank of 128 inputs
85	Change multiple analog outputs
91	Read a bank of 128 outputs
Servo Access Commands	
23	Read a servo position
27	Read a servo's dedicated inputs
47	Read a servo error
Data Table Access Commands	
49	Read a data table's dimensions
51	Change a data table's dimensions
53	Read a data table value
55	Change a data table value
57	Read a row of data table values
59	Change a row of data table values

System and Controller Status Access Commands	
13	List counts of inputs, outputs, stepping and servo motors
35	Read controller step
61	Read controller status
63	Change controller status
65	Read system configuration
67	Change system configuration
69	List counts of miscellaneous I/O
105	Shutdown system
107	Get Controller Task Status

The following commands allow you to read and write values to registers and flags. You can read and write values for registers 1 through 65535. Some of the registers in this range are special function registers and you may not be able to read or write to them. Other registers do not exist on certain models and revision levels. Consult *Model 5300 Quick Reference Register Guide (951-530006)* for register specifics.

Variant Packets

A number of commands are available to interface with variant storage within the 5300. When communicating with the controller a packed data structure is used. Two separate structures are used, that for individual read/writes, VARIANT_STORAGE, or for block access VARIANT_STORAGE_BLOCK (VARIANT_STORAGE_BLOCK_SERIAL if serial port). When using block transfers the total size (number of elements) is dependent upon whether Ethernet or serial communications is being used. Ethernet allows for a larger packet and when using UDP and TCP the packet itself provides a CRC thus the checksum field is not really needed and not used on the larger block transfers.

When using variants the packet structure is identical except that the data portion is the packed variant structure:

<(Ø1H)> Specifies CTC binary protocol.
<length (1 byte)> Specifies packet length to follow. Packet length is defined as n data bytes + 2 (checksum and 0xff). Checksum is not used on packet type 109/110, 111/112, and 113/114 when using Ethernet communications (length set to 5 on request, response length is 3), it is used on serial since a reduce packet size is used.
<Command/Response Code>
<LSB Register #> Register of interest low byte unless random read in which case ignored.
<MSB Register #> Register of interest high byte unless random read in which case ignored.
<packed variant structure> Valid structures:

VARIANT_STORAGE
VARIANT_STORAGE_BLOCK
VARIANT_STORAGE_BLOCK_SERIAL.

<checksum> Consists of the complement of the modulo-256 sum of data bytes. This value, when added to the modulo-256 sum of the data packet bytes, equals ØFFH. You can calculate the checksum by adding the data packet bytes and complementing the resulting sum.

Register and Flag Access Command/Response definitions

// GET is request, GOT is controller response
// binary protocol message types

```
#define MSG_LOAD_PROGRAM_PACKET          ((BYTE) 0)
#define MSG_ENTER_PROGRAM_MODE           ((BYTE) 1)
#define MSG_LEAVE_PROGRAM_MODE           ((BYTE) 2)
#define MSG_UNLOAD_PROGRAM_PACKET        ((BYTE) 3)
#define MSG_PROGRAM_PACKET               ((BYTE) 4)
#define MSG_GET_ID_CODES                 ((BYTE) 5)
#define MSG_GOT_ID_CODES                 ((BYTE) 6)
#define MSG_OLD_GET_STATUS               ((BYTE) 7)
#define MSG_OLD_GOT_STATUS               ((BYTE) 8)
#define MSG_GET_REGISTER                 ((BYTE) 9)
#define MSG_GOT_REGISTER                 ((BYTE) 10)
#define MSG_SET_REGISTER                 ((BYTE) 11)
#define MSG_12                           ((BYTE) 12)
#define MSG_GET_IO_COUNTS                ((BYTE) 13)
#define MSG_GOT_IO_COUNTS                ((BYTE) 14)
#define MSG_GET_INPUTS                   ((BYTE) 15)
#define MSG_GOT_INPUTS                   ((BYTE) 16)
#define MSG_GET_FLAG                     ((BYTE) 17)
#define MSG_GOT_FLAG                     ((BYTE) 18)
#define MSG_SET_FLAG                     ((BYTE) 19)
#define MSG_20                           ((BYTE) 20)
#define MSG_GET_OUTPUTS                  ((BYTE) 21)
#define MSG_GOT_OUTPUTS                  ((BYTE) 22)
#define MSG_GET_SERVO_POSITION           ((BYTE) 23)
#define MSG_GOT_SERVO_POSITION           ((BYTE) 24)
#define MSG_SET_OUTPUTS                  ((BYTE) 25)
#define MSG_26                           ((BYTE) 26)
#define MSG_GET_SERVO_INPUT              ((BYTE) 27)
#define MSG_GOT_SERVO_INPUT              ((BYTE) 28)
#define MSG_GET_ANALOG_INPUT             ((BYTE) 29)
#define MSG_GOT_ANALOG_INPUT             ((BYTE) 30)
#define MSG_GET_ANALOG_OUTPUT            ((BYTE) 31)
#define MSG_GOT_ANALOG_OUTPUT            ((BYTE) 32)
#define MSG_SET_ANALOG_OUTPUT            ((BYTE) 33)
```

```
#define MSG_34                          ((BYTE) 34)
#define MSG_GET_STATUS                  ((BYTE) 35)
#define MSG_GOT_STATUS_1of4             ((BYTE) 36)
#define MSG_GOT_STATUS_2of4             ((BYTE) 37)
#define MSG_GOT_STATUS_3of4             ((BYTE) 38)
#define MSG_GOT_STATUS_4of4             ((BYTE) 39)
#define MSG_SET_EA_OUTPUT               ((BYTE) 40)
#define MSG_LOAD_EA_PROGRAM_PACKET      ((BYTE) 41)
#define MSG_42                          ((BYTE) 42)
#define MSG_UNLOAD_EA_PROGRAM_PACKET    ((BYTE) 43)
#define MSG_EA_PROGRAM_PACKET           ((BYTE) 44)
#define MSG_DUMP_USER_MEMORY            ((BYTE) 45)
#define MSG_USER_MEMORY                 ((BYTE) 46)
#define MSG_GET_SERVO_ERROR             ((BYTE) 47)
#define MSG_GOT_SERVO_ERROR             ((BYTE) 48)
#define MSG_GET_DATA_TABLE_SIZE         ((BYTE) 49)
#define MSG_GOT_DATA_TABLE_SIZE         ((BYTE) 50)
#define MSG_SET_DATA_TABLE_SIZE         ((BYTE) 51)
#define MSG_52                          ((BYTE) 52)
#define MSG_GET_DATA_TABLE_ELEMENT      ((BYTE) 53)
#define MSG_GOT_DATA_TABLE_ELEMENT      ((BYTE) 54)
#define MSG_SET_DATA_TABLE_ELEMENT      ((BYTE) 55)
#define MSG_56                          ((BYTE) 56)
#define MSG_GET_DATA_TABLE_ROW          ((BYTE) 57)
#define MSG_GOT_DATA_TABLE_ROW          ((BYTE) 58)
#define MSG_SET_DATA_TABLE_ROW          ((BYTE) 59)
#define MSG_60                          ((BYTE) 60)
#define MSG_GET_CONTROLLER_STATE        ((BYTE) 61)
#define MSG_GOT_CONTROLLER_STATE        ((BYTE) 62)
#define MSG_SET_CONTROLLER_STATE        ((BYTE) 63)
#define MSG_64                          ((BYTE) 64)
#define MSG_GET_SYSCONFIG_BYTE          ((BYTE) 65)
#define MSG_GOT_SYSCONFIG_BYTE          ((BYTE) 66)
#define MSG_SET_SYSCONFIG_BYTE          ((BYTE) 67)
#define MSG_68                          ((BYTE) 68)
#define MSG_GET_OTHER_IO_COUNTS         ((BYTE) 69)
#define MSG_GOT_OTHER_IO_COUNTS         ((BYTE) 70)
#define MSG_GET_32_ANALOG_INS           ((BYTE) 71)
#define MSG_GOT_32_ANALOG_INS           ((BYTE) 72)
#define MSG_GET_32_ANALOG_OUTS          ((BYTE) 73)
#define MSG_GOT_32_ANALOG_OUTS          ((BYTE) 74)
#define MSG_GET_50_REGISTERS            ((BYTE) 75)
#define MSG_GOT_50_REGISTERS            ((BYTE) 76)
#define MSG_GET_16_REGISTERS            ((BYTE) 77)
#define MSG_GOT_16_REGISTERS            ((BYTE) 78)
#define MSG_GET_128_INPUTS              ((BYTE) 79)
```

```
#define MSG_GOT_128_INPUTS                  ((BYTE) 80)
#define MSG_GET_128_OUTPUTS                 ((BYTE) 81)
#define MSG_GOT_128_OUTPUTS                 ((BYTE) 82)
#define MSG_SET_64_ANALOG_OUTS              ((BYTE) 85)
#define MSG_GET_N_REGISTERS                 ((BYTE) 87)
#define MSG_GOT_N_REGISTERS                 ((BYTE) 88)

// special message for 2217 v3.8 data structure
#define MSG_GET_2217_DATA                   ((BYTE) 83)
#define MSG_GOT_2217_DATA                   ((BYTE) 84)

// Variant data types
#define MSG_GET_VREGISTERROW                ((BYTE) 89)
#define MSG_GOT_VREGISTERROW                ((BYTE) 90)
#define MSG_GET_VPROPERTIES                 ((BYTE) 91)
#define MSG_GOT_VPROPERTIES                 ((BYTE) 92)
#define MSG_GET_VREGISTER                   ((BYTE) 93)
#define MSG_GOT_VREGISTER                   ((BYTE) 94)
#define MSG_SET_VREGISTER                   ((BYTE) 95)
#define MSG_GET_RUNCOMMAND                  ((BYTE) 97)
#define MSG_GOT_RUNCOMMAND                  ((BYTE) 98)
#define MSG_GET_VREGISTER_BLOCK             ((BYTE) 109)
#define MSG_GOT_VREGISTER_BLOCK             ((BYTE) 110)
#define MSG_SET_VREGISTER_BLOCK             ((BYTE) 111)

#define MSG_GET_VREGISTER_RANDOM_BLOCK      ((BYTE) 113)
#define MSG_GOT_VREGISTER_RANDOM_BLOCK      ((BYTE) 114)

// devicenet and/or distributed io messages
#define MSG_UNLOAD_REMOTE_DATA              ((BYTE) 101)
#define MSG_REMOTE_DATA_PACKET              ((BYTE) 102)
#define MSG_LOAD_REMOTE_DATA                ((BYTE) 103)
#define MSG_104                             ((BYTE) 104)
#define MSG_GET_REMOTE_IO                   ((BYTE) 105)
#define MSG_GOT_REMOTE_IO                   ((BYTE) 106)
```

Variant Structures

The distribution file Ctccom32v2.h is available on the Control Technology web site and contains the definitions for the structures used with the CTC communications DLL. The DLL conforms to the packet structure discussed within this document. In summary below are the definitions. Note the structures are packed, aligned on a byte boundary:

```
#define BIT0 0x0001
#define BIT1 0x0002
#define BIT2 0x0004
#define BIT3 0x0008
#define BIT4 0x0010

#define VARIANT_MAX_STRING 223
#define VARIANT_INTEGER BIT0
#define VARIANT_UINTEGER BIT1
#define VARIANT_STRING BIT2
#define VARIANT_FLOAT BIT3
#define VARIANT_DOUBLE BIT4

typedef struct
{
        int numAccess; // number of items to access
        int rowInc;            // row increment, if 0 just read columns based upon colInc.
        int colInc;            // col increment, if 0 just increment rows.
        int arraysizeCols;     // Used on write operation, -1 do not expand existing
                               // columns, else columns desired.  Rows will automatically
                               // grow as needed
} BLOCKACCESS;

typedef struct
{
   int type;   // type of storage being used or requested
                       // If -1 on read then return current, else set to type want.
                       // On write must set to type that is stored within this structure
   unsigned char precision;  // double to string conversion precision %.6f default
                              // On read is what is presently set, write what want.
   unsigned char flags;// special flags for processing so far only
                       // VARIANT_INDIRECTION_FLAG used, can be used to set property
                       // in ->settings on write operation, no effect on read.  Written
                       // value becomes register to reference for further operations.
   unsigned char cmd;// 00, no operation other than read/write specified, else do defined
               // operation.  Currently have write for properties access to 'settings'
                       // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
                       // write value ignored.
   unsigned char pad;
   unsigned short taskHandle; // task number (offset in task array + 1, where 0 is 1) or
                              / handle thus usable from remote or 'C' API, 4096 to
                              // 65535, set to 0 for public reg.
   unsigned short slength;    // this is reserved for later use and possible string
                              // length if want unsigned char, 0 - 255 values,
                              // VARIANT_BYTE, future type

   unsigned int indexCol;       // Column dimension index reference
   unsigned int indexRow;       // Row dimension index reference
   union                      // Data that was read or has been written of 'type'
   {
        int iValue;
        unsigned int uiValue;
        float fValue;
        double dValue;
        unsigned int dSwap[2]; // used to swap doubles for PC access
        char sValue[VARIANT_MAX_STRING+1];
   } data;
} VARIANT_STORAGE;

#define MAX_VARIANT_BLOCK_32BITS      346     // 346 integers
#define MAX_VARIANT_BLOCK_64BITS      173     // 173 doubles
#define MAX_VARIANT_RANDOM_BLOCK      (MAX_VARIANT_BLOCK_32BITS/3)  // 115 items

#define MAX_VARIANT_BLOCK_32BITS_SERIAL 50
#define MAX_VARIANT_BLOCK_64BITS_SERIAL (MAX_VARIANT_BLOCK_32BITS_SERIAL/2)// 25 doubles
#define MAX_VARIANT_RANDOM_BLOCK_SERIAL (MAX_VARIANT_BLOCK_32BITS_SERIAL/3)// 16 items


typedef struct
{
        int reg;  // may at some point reserve the upper 16 bits of this
```

```
                // integer for 'type' req.
        int row;
        int col;
} VARIANT_ACCESS_REQUEST;

// Allow for block reads
typedef struct
{
   int type;    // type of storage being used or requested
                // If -1 on read then return current, else set to type want.
                // On write must set to type that is stored within this structure
                // write not supported for block access
                // If block access type field will be 0 if error else type of first cell.
                // slength will be the number of elements returned within data.block.?[n]
   unsigned char precision;  // double to string conversion precision %.6f default
                             // On read is what is presently set, write what want.
   unsigned char flags;      // special flags for processing so far only
                      // VARIANT_INDIRECTION_FLAG used, can be used to set property
                      // in ->settings on write operation, no effect on read.  Written
                      // value becomes register to reference for further operations.
   unsigned char cmd;// 00, no operation other than read/write specified, else do defined
                // operation.  Currently have write for properties access to 'settings'
                // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
                // write value ignored.
   unsigned char pad;
   unsigned short taskHandle; // task number (offset in task array + 1, where 0 is 1) or
                              // handle thus usable from remote or 'C' API, 4096 to
                              // 65535, set to 0 for public reg.

   unsigned short slength;    // this is reserved for later use and possible string
                              // length if want unsigned char, 0 - 255 values,
                              // VARIANT_BYTE, future type
   unsigned int indexCol;     // Column dimension index reference
   unsigned int indexRow;     // Row dimension index reference
   union                      // Data that was read or has been written of 'type'
   {
        int iValue;
        unsigned int uiValue;
        float fValue;
        double dValue;
        unsigned int dSwap[2]; // used to swap doubles for PC access
        char *psValue;
        char sValue[VARIANT_MAX_STRING+1];
        // will be stored in same VARIANT_STORAGE upon return, thus
        // data.blockread.numAccess * sizeof(variant type)
        // if BLOCKACCESS then iValue[n], fValue[n], or dValue[n] up to
        // MAX_VARIANT_READBLOCK_SIZE

        struct
        {
                BLOCKACCESS blockaccess;      // Defines block read of variant cells, data
                                              // storage must be big enough since
                union
                {
                        int ibValue[MAX_VARIANT_BLOCK_32BITS];
                        float fbValue[MAX_VARIANT_BLOCK_32BITS];
                        double dbValue[MAX_VARIANT_BLOCK_64BITS];
                        union
                        {
                                int ibValue;
                                float fbValue;
                                double dbValue;
                        } random[MAX_VARIANT_RANDOM_BLOCK];
                        VARIANT_ACCESS_REQUEST request[MAX_VARIANT_RANDOM_BLOCK];
                };
        } block;
   } data;
} VARIANT_STORAGE_BLOCK;

typedef struct
{
```

```
   int type;    // type of storage being used or requested
                // If -1 on read then return current, else set to type want.
                // On write must set to type that is stored within this structure
                // write not supported for block access
                // If block access type field will be 0 if error else type of first cell.
                // slength will be the number of elements returned within data.block.?[n]
   unsigned char precision;  // double to string conversion precision %.6f default
                             // On read is what is presently set, write what want.
   unsigned char flags;      // special flags for processing so far only
                        // VARIANT_INDIRECTION_FLAG used, can be used to set property
                        // in ->settings on write operation, no effect on read.  Written
                        // value becomes register to reference for further operations.
   unsigned char cmd;// 00, no operation other than read/write specified, else do defined
                // operation.  Currently have write for properties access to 'settings'
                // VARIANT_CMD_SET_INDIRECTION and VARIANT_CMD_CLEAR_INDIRECTION,
                // write value ignored.
   unsigned char pad;
   unsigned short taskHandle; // task number (offset in task array + 1, where 0 is 1) or
                              // handle thus usable from remote or 'C' API, 4096 to
                              // 65535, set to 0 for public reg.

   unsigned short slength;    // this is reserved for later use and possible string
                              // length if want unsigned char, 0 - 255 values,
                              // VARIANT_BYTE, future type
   unsigned int indexCol;     // Column dimension index reference
   unsigned int indexRow;     // Row dimension index reference
   union                      // Data that was read or has been written of 'type'
   {
        int iValue;
        unsigned int uiValue;
        float fValue;
        double dValue;
        unsigned int dSwap[2]; // used to swap doubles for PC access
        char *psValue;
        char sValue[VARIANT_MAX_STRING+1];
        // will be stored in same VARIANT_STORAGE upon return, thus
        // data.blockread.numAccess * sizeof(variant type)
        // if BLOCKACCESS then iValue[n], fValue[n], or dValue[n] up to
        // MAX_VARIANT_READBLOCK_SIZE

        struct
        {
                BLOCKACCESS blockaccess;      // Defines block read of variant cells, data
                                              // storage must be big enough since
                union
                {
                        int ibValue[MAX_VARIANT_BLOCK_32BITS_SERIAL];
                        float fbValue[MAX_VARIANT_BLOCK_32BITS_SERIAL];
                        double dbValue[MAX_VARIANT_BLOCK_64BITS_SERIAL];
                        union
                        {
                                int ibValue;
                                float fbValue;
                                double dbValue;
                        } random[MAX_VARIANT_RANDOM_BLOCK_SERIAL];
                        VARIANT_ACCESS_REQUEST request[MAX_VARIANT_RANDOM_BLOCK_SERIAL];
                };
        } block;
   } data;
} VARIANT_STORAGE_BLOCK_SERIAL;
```

Variant Access Commands

Get Properties - Command 91

Command 91 reads the current properties of a variant which includes its number of rows and columns as well as default floating point precision (typically 6).

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
05H Specifies the packet length
5BH Get Properties function code
LSB - MSB Specifies the variant register number whose properties desire. Specified with the least significant byte first.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response

ØAH Specifies the packet length.
5CH Get Properties response code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
LSB- MSB Number of columns.
LSB- MSB Number of rows.
<Precision Byte> - Floating point precision currently set.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Read a Variant - Command 93

Command 93 reads a Variant cell. If the Variant is not an array simply set the row and column to 0 in the structure.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
<sizeof(VARIANT_STORAGE) + 5> Specifies the packet length
5DH Read a Variant function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE structure> Variant storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response

<sizeof(VARIANT_STORAGE) + 3> Specifies the packet length
5EH Read a Variant response code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE structure> Variant storage area.

Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Example Structure initialization:
Read 36201[2][5] as a double – (36201 is the LSB/MSB in the message sent)

```
VARIANT_STORAGE v;
memset((void *)&v,0,sizeof(VARIANT_STORAGE));
v.indexCol = 5;
v.indexRow = 2;
v.precision = 6;
v.type = VARIANT_DOUBLE;
```

Depending upon which type you are accessing the returned Variant will be accessed as follows where 'rp' is a pointer to the receive buffer.

```
// Got the data
memcpy((void *)&v,rp+4,sizeof(VARIANT_STORAGE));
switch(v.type)
{
        case VARIANT_FLOAT:
                variant->FloatVar = v.data.fValue;
                break;
        case VARIANT_DOUBLE:
                variant->DoubleVar = v.data.dValue;
                break;
        case VARIANT_STRING:
                variant->slength = v.slength;
                if (variant->slength > VARIANT_MAX_STRING)
                {
                        // too big
                        return FAILURE;
                }
                memcpy(variant->StringArray, v.data.sValue, variant->slength);
                // null terminate
                variant->StringArray[variant->slength] = 0x00;
                break;
        case VARIANT_INTEGER:
                variant->LongVar = v.data.iValue;
                break;
        default:
                return FAILURE;  // unknown type
}
return SUCCESS;
```

Change a Variant - Command 95

Command 95 writes a Variant cell. If the Variant is not an array simply set the row and column to 0 in the structure.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol

<sizeof(VARIANT_STORAGE) + 5> Specifies the packet length
5FH Change a Variant function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE structure> Variant storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
Checksum Contains the complement of the previous byte
FFH Signals the end of the message

Example Structure initialization:
Write 36201[2][5]– (36201 is the LSB/MSB in the message sent)

```
VARIANT_STORAGE v;
memset((void *)&v,0,sizeof(VARIANT_STORAGE));
v.indexCol = 5;
v.indexRow = 2;
v.precision = 6;
```

For each type of data writing where 'variant' is user structure (reference previous section):

```
switch(variant->type)
{
        case VARIANT_FLOAT:
                v.data.fValue = variant->FloatVar;
                break;
        case VARIANT_DOUBLE:
                v.data.dValue = variant->DoubleVar;
                break;
        case VARIANT_STRING:
                if (variant->slength > VARIANT_MAX_STRING)
                {
                        // too big
                        return FAILURE;
                }
                memcpy(v.data.sValue, variant->StringArray,variant->slength);
                // null terminate
                v.data.sValue[variant->slength] = 0x00;
                break;
        case VARIANT_INTEGER:
                v.data.iValue = variant->LongVar;
                break;
        default:
                return FAILURE;
}
```

… Send data packet and await ACK …

Read a Variant Array Block - Command 109

Command 109 performs a read starting at a specific row/column position in a Variant array and reads the requested number of cells sequentially or until there are no more.

Format of Message Sent to Controller
UDP/TCP

Ø1H Identifies the packet as using the CTC binary protocol
< 5> Specifies the packet length for, without Variant area since exceeds byte storage size..
6DH Read a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field, **not used**.
FFH Signals the end of the message.

Serial Port:

Ø1H Identifies the packet as using the CTC binary protocol
<sizeof(VARIANT_STORAGE_BLOCK_SERIAL) + 5> Specifies the packet length.
6DH Read a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK_SERIAL structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response
UDP/TCP

< 3> Specifies the packet length for, without Variant area since exceeds byte storage size..
6EH Read a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK structure> Variant storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field, **not used**.
FFH Signals the end of the message.

Serial Port:

<sizeof(VARIANT_STORAGE_BLOCK_SERIAL) + 3> Specifies the packet length.
6EH Read a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK_SERIAL structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Example Structure initialization:
Read 36201[0][0] as a integer, 5 consecutive cells – (36201 is the LSB/MSB in the message sent). There are 35 columns in each row.

```
VARIANT_STORAGE_BLOCK v;
if (ctc->connType == SERIAL)
{
        sz = sizeof(VARIANT_STORAGE_BLOCK_SERIAL);
        length = sz+5;      // packet length

}
else
{
        sz = sizeof(VARIANT_STORAGE_BLOCK);
        length = 5;
}
// initialize the variant structure
memset((void *)&v.type,0,sz);
v.indexCol = 0;
v.indexRow = 0;
v.precision = 6;
v.type = VARIANT_INTEGER;  // String not supported
v.data.block.blockaccess.colInc = 1;
v.data.block.blockaccess.rowInc = 1;
v.data.block.blockaccess.numAccess = 5;
v.data.block.blockaccess.arraysizeCols = 35;  // tells when to increment row number
```

Depending upon which type you are accessing the returned Variant will be accessed as follows where 'rp' is a pointer to the receive buffer. 'sz' is the size of the structure used, that of VARIANT_STORAGE_BLOCK or VARIANT_STORAGE_BLOCK_SERIAL.

```
// Got the data
memcpy((void *)&v,rp+4,sz);
// move the data into the vb structure
variant->type = v.type;                         // type of data read
variant->slength = v.slength;          // number read
switch(variant->type)
{
        case VARIANT_FLOAT:
                memcpy(variant->block.fbValue, v.data.block.fbValue,
                        sizeof(float) * variant->slength);
```

```
                    break;
                case VARIANT_DOUBLE:
                    memcpy(variant->block.dbValue, v.data.block.dbValue,
                        sizeof(double) * variant->slength);
                    break;
                case VARIANT_INTEGER:
                case VARIANT_UINTEGER:
                    memcpy(variant->block.ibValue, v.data.block.ibValue,
                        sizeof(int) * variant->slength);
                    break;
                default:
                    return FAILURE;  // unknown type
        }
        return SUCCESS;
```

Write a Variant Array Block - Command 111

Command 111 performs a write starting at a specific row/column position in a Variant array and writes the requested number of cells sequentially or until there are no more.

Format of Message Sent to Controller
UDP/TCP

Ø1H Identifies the packet as using the CTC binary protocol
< 5> Specifies the packet length for, without Variant area since exceeds byte storage size..
6FH Write a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field, **not used**.
FFH Signals the end of the message.

Serial Port:

Ø1H Identifies the packet as using the CTC binary protocol
<sizeof(VARIANT_STORAGE_BLOCK_SERIAL) + 5> Specifies the packet length.
6DH Write a Variant Block function code
LSB - MSB Specifies the variant register number. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK_SERIAL structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.

64H Contains the acknowledge function code (decimal 100)
Checksum Contains the complement of the previous byte
FFH Signals the end of the message

Example Structure initialization:
Write 36201[0][0] as a floats, 5 consecutive cells – (36201 is the LSB/MSB in the message sent). There are 35 columns in each row.

```
VARIANT_STORAGE_BLOCK v;
if (ctc->connType == SERIAL)
{
        sz = sizeof(VARIANT_STORAGE_BLOCK_SERIAL);
        length = sz+5;      // packet length

}
else
{
        sz = sizeof(VARIANT_STORAGE_BLOCK);
        length = 5;
}
// initialize the variant structure
memset((void *)&v.type,0,sz);
v.indexCol = 0;
v.indexRow = 0;
v.precision = 6;
v.type = VARIANT_FLOAT;  // String not supported
v.data.block.blockaccess.colInc = 1;
v.data.block.blockaccess.rowInc = 1;
v.data.block.blockaccess.numAccess = 5;  // assume variant->numAccess is 5
v.data.block.blockaccess.arraysizeCols = 35;
// move the data in now where 'variant' is a user structure of choice
memcpy((void *)&v.data.block.fbValue, variant->fbValue, sizeof(float) * variant->numAccess);
```

… Send data packet and await ACK …

Read a Block of Variants Randomly - Command 113

Command 113 read variants in any desired order of any cell or different variant. All will be returned of the same type, integer, float, or double. String is not supported.

Format of Message Sent to Controller
UDP/TCP

Ø1H Identifies the packet as using the CTC binary protocol
< 5> Specifies the packet length for, without Variant area since exceeds byte storage size..
71H Read a random Variant Block function code
LSB - MSB Specifies the variant register number, may be any value such as 0x0000. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field, **not used**.
FFH Signals the end of the message.

Serial Port:

Ø1H Identifies the packet as using the CTC binary protocol
<sizeof(VARIANT_STORAGE_BLOCK_SERIAL) + 5> Specifies the packet length.
71H Read a random Variant Block function code
LSB - MSB Specifies the variant register number, may be any value such as 0x0000. Specified with the least significant byte first.
<VARIANT_STORAGE_BLOCK_SERIAL structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Format of Controller Response
UDP/TCP

< 3> Specifies the packet length for, without Variant area since exceeds byte storage size..
72H Read a random Variant Block response code
LSB - MSB returns what sent.
<VARIANT_STORAGE_BLOCK structure> Variant storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field, **not used**.
FFH Signals the end of the message.

Serial Port:

<sizeof(VARIANT_STORAGE_BLOCK_SERIAL) + 3> Specifies the packet length.
72H Read a randomVariant Block response code
LSB - MSB returns what sent.
<VARIANT_STORAGE_BLOCK_SERIAL structure> Variant block storage area.
Checksum Contains the complement of the modulo-256 sum of all bytes after the length field
FFH Signals the end of the message.

Example Structure initialization:
Read 36301[0][0], 36301[0][1], 36301[0][2], 36301[1][0], 36302[0][0] as doubles.

```
VARIANT_STORAGE_BLOCK v;
// initialize the variant structure
if (ctc->connType == SERIAL)
{
        sz = sizeof(VARIANT_STORAGE_BLOCK_SERIAL);
        length = sz+5;      // packet length

}
```

```
else
{
        sz = sizeof(VARIANT_STORAGE_BLOCK);
        length = 5;
}
// initialize the variant structure
memset((void *)&v.type,0,sz);
v.precision = 6;
v.type = VARIANT_DOUBLE;
v.data.block.blockaccess.numAccess = 5;
// 36301[0][0]
v.data.block.request[0].reg = 36301;
v.data.block.request[0].row = 0;
v.data.block.request[0].col = 0;
// 36301[0][0]
v.data.block.request[1].reg = 36301;
v.data.block.request[1].row = 0;
v.data.block.request[1].col = 1;
// 36301[0][0]
v.data.block.request[2].reg = 36301;
v.data.block.request[2].row = 0;
v.data.block.request[2].col = 2;
// 36301[0][0]
v.data.block.request[3].reg = 36301;
v.data.block.request[3].row = 1;
v.data.block.request[3].col = 0;
// 36301[0][0]
v.data.block.request[4].reg = 36302;
v.data.block.request[4].row = 0;
v.data.block.request[4].col = 0;
```

Depending upon which type you are accessing the returned Variant will be accessed as follows where 'rp' is a pointer to the receive buffer. 'sz' is the size of the structure used, that of VARIANT_STORAGE_BLOCK or VARIANT_STORAGE_BLOCK_SERIAL.

```
// Got the data
memcpy((void *)&v,rp+4,sz);
// move the data into the vb structure
variant->type = v.type;                         // type of data read
variant->slength = v.slength;          // number read
memcpy(&variant->block.random[0], &v.data.block.random[0],
        sizeof(v.data.block.random) * variant->slength);
return SUCCESS;
```

Register and Flag Access Commands

Binary Protocol Conventions

The binary protocol uses specific conventions for specifying register and flag numbers and values and for checksum error detection.

- When specifying a register number, it is expressed as ØØØ1H through ØFFFFH, corresponding to registers 1 through 65535. For example, register 10 is expressed as ØØØAH.

- You must specify register numbers with the least significant byte first.
- When specifying a flag number, it is expressed as ØØH through ØFH for flags, corresponding to flags 1 through 32. For example, flag 5 is expressed as Ø4H.
- The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description.
- When the controller responds with a register value, it is always a four-byte representation of the register data expressed in 2's complement binary, with the least significant byte transmitted first.

Reading a Numeric Register - Command 9

Command 9 reads the value in any register that allows read access.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø5H Specifies the packet length
Ø9H Indicates the read register function code
LSB - MSB Specifies the register number, ØØØ1H - ØFFFFH. Specified with the least significant byte first.
Checksum Contains the complement of the modulo-256 sum of the previous 3 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø7H Specifies the packet length.
ØAH Indicates the register contents function code
LSB, 3SB, Four-byte representation of register data, expressed in 2's
2SB, MSB complement binary, with the least significant byte transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 5 bytes
FFH Signals the end of the message.

Reading a Bank of 16 Registers - Command 77

Command 77 reads the values in a bank of 16 consecutive registers.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø5H Specifies the packet length
4DH Indicates 16 register group read function code
LSB - MSB Specifies bank of registers to read, ØØØØH - Ø3D9H
Checksum Contains the complement of the modulo-256 sum of the previous 3 bytes
FFH Signals the end of the message

Format of Controller Response

45H Specifies the packet length
E4H Indicates the register contents function code
LSB - MSB Indicates bank of registers, ØØØØH - Ø3D9H
LSB, 3SB, Contains the value of the first register in the group. For a
2SB, MSB description of register data. See the description for single register read.
LSB, 3SB, Contains the value of the second register in the
2SB, MSB group and continues for all 16 registers in the group.
Checksum Contains the complement of the modulo-256 sum of the previous 67 bytes.
FFH Signals the end of the message.

Reading a Bank of 50 Registers - Command 75

Command 75 reads the values in a bank of 50 consecutive registers, limited from 1 to 1000.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
4BH Indicates 50 register group read function code
ØØH - 13H Specifies the bank of 50 registers to be read, , ØØH - 13H
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Format of Controller Response

CCH Specifies the packet length
4CH Indicates the register contents function code
ØØH - 13H Indicates the bank of 50 register to follow, ØØH - 13H
LSB, 3SB, Contains the value of the first register in the group. For a
2SB, MSB description of register data. See the description for single register read.
LSB, 3SB, Contains the value of the second register in the
2SB, MSB group and continues for all 50 registers in the group
Checksum Contains the complement of the modulo-256 sum of the previous 202 bytes.
FFH Signals the end of the message.

Request Random Registers from List - Command 87

Command 87 reads the values of up to 50 random registers from a list.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
??H Specifies the packet length, all following bytes, including checksum but not ending FFH.
57H Indicates Random register read function code

NUMREGS – Single byte from 1 to 50 representing number of following random registers to read. Registers are listed as 2 byte shorts (16 bits), lsb/msb, results are returned as 32 bit integers.
LSB – MSB1 First register number to read, 16 bits
LSB – MSB2 Second register number to read, 16 bits
…
LSB – MSBN Last register number to read, 16 bits
Checksum Contains the complement of the modulo-256 sum of all the bytes after the packet length bytes.
FFH Signals the end of the message

Format of Controller Response

??H Specifies the packet length
58H Indicates the register contents function code
NUMREGS – Single byte from 1 to 50 representing number of following random registers results which are being returned. Registers results are returned as 32 bit integers, lsb to msb.
LSB, 3SB, Contains the value of the first register in the group. For a
2SB, MSB description of register data. See the description for single register read.
LSB, 3SB, Contains the value of the second register in the
2SB, MSB group and continues for all NUMREGS registers in the group.
…
Checksum Contains the complement of the modulo-256 sum of the previous bytes, excluding packet length.
FFH Signals the end of the message.

Changing a Register Value - Command 11

Command 11 changes the value in any register that allows write access.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø9H Specifies the packet length
ØBH Indicates the change register value function code
LSB - MSB Specifies the register number, ØØØ1H - ØFFFFH. Specified with the least significant byte first.
LSB, 3SB, Four-byte representation of register data, expressed in 2's
2SB, MSB complement binary, with the least significant byte transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 7 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)

Checksum Contains the complement of the previous byte
FFH Signals the end of the message.

Reading a Flag's State - Command 17

Command 17 reads the state of any flag.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
11H Indicates the read flag state function code
Flag Number Specifies the flag number, ØØH - 1FH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø4H Specifies the packet length.
12H Indicates the flag state function code
ØØH or FFH Indicates the flag's status. ØØH if flag is clear and FHH if set. Any other value means that the results are indeterminate.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Changing a Flag's State - Command 19

Command 19 changes the state of any flag.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø5H Specifies the packet length
13H Indicates the change flag state function code
Flag Number Specifies the flag to be changed, ØØH - 1FH
ØØH or FFH Specifies the new state of the flag. ØØH represents CLEAR and FFH represents SET.
Checksum Contains the complement of the previous 3 bytes
ØFFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
Checksum Contains the complement of the previous byte
FFH Signals the end of the message

Digital Input/Output Access Commands

The following commands allow you to read digital input and output states and turn a digital output on or off. Input and output states are read as a group of either 8 or 128.

Binary Protocol Conventions

The binary protocol uses specific conventions for specifying groups of inputs and outputs, their states and for checksum error detection.

- When specifying a bank of inputs or outputs as a group of 8, the first bank of inputs or outputs are specified as ØØH, corresponding to 1 through 8. The second bank is specified as Ø12H, corresponding to 9 through 16, and so on up to 7FH for the 16th bank, corresponding to 121 through 128.
- The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description.
- When the controller responds with a the data for a group of 8 inputs or outputs, the lowest input number is represented by the least significant, the next the 7th least significant bit, and so on.
- For input states, a 1 represents a grounded (on) input.
- For output states, a 1 represents an output that is turned on.

Reading a Bank of 8 Inputs - Command 15

Command 15 reads the state of a group of eight digital inputs. The read inputs function code (ØFH) allows you to read a group of 8 inputs. Inputs are grouped so that the first group of inputs is 1 to 8; the second is 9 to 16, up to 121 to 128 for the 16th and last group.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
ØFH Indicates the read input state function code
Bank Specifies the bank of inputs, ØØH - 7FH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø4H Specifies the packet length.
1ØH Indicates the input data function code
ØØH - FFH Contains the data for the eight inputs, The lowest input number is represented by the least significant bit. A 1 indicates a grounded (on) input.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Reading a Bank of 128 Inputs - Command 79

Command 79 reads a bank of 128 inputs.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
4FH Indicates the read 128 inputs request function code
Bank Specifies the input bank to read, ØØH - 7FH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø4H Specifies the packet length.
50H Indicates the input values function code
Bank Input bank to follow, ØØH - 7FH
Inps1-8 Contains the data for the eight inputs, with the lowest input number is represented by the least significant bit. A value of 1 indicates a grounded (on) input.
Inps9-16 Contains the data for the next eight inputs. This continues for a total of 128 inputs.
Checksum Contains the complement of the modulo-256 sum of the previous 18 bytes
FFH Signals the end of the message

NOTE: The controller returns a value of zero (0) for nonexistent inputs with in a bank.

Reading a Bank of 8 Outputs - Command 21

Command 21 reads the state of a group of eight digital outputs. Outputs are grouped in the same manner as inputs.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
15H Indicates the read output state function code
Bank Specifies the bank of outputs, ØØH - 7FH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø4H Specifies the packet length.
16H Indicates the output status function code
ØØH - FFH Contains the data for the eight outputs with the lowest output number represented by the least significant bit. A 1 indicates a that an output is on.

Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Reading a Bank of 128 Outputs - Command 91

Command 91 reads a bank of 128 digital outputs. The outputs are grouped in the same manner as inputs.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
51H Indicates the read 128 outputs request function code
Bank Specifies the bank of outputs, ØØH - 7FH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

14H Specifies the packet length.
52H Indicates the output values function code
Bank Specifies the bank of outputs, ØØH - 7FH
Outs1-8 Contains the data for the eight outputs, with the lowest output number is represented by the least significant bit. A value of 1 indicates an output is on.
Outs9-16 Contains the data for the next eight output. This continues for a total of 128 output.
Checksum Contains the complement of the modulo-256 sum of the previous 18 bytes
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

NOTE: The controller reports nonexistent outputs within a bank as off, value is 0.

Selectively Changing the First 128 Outputs - Command 25

Command 25 selectively changes the state of a group of 128 digital outputs. This command uses separate on and off masks so you can change specific outputs. For example, an off-mask-Ø of Ø6H (ØØØØ Ø11Ø in binary) would turn off outputs one along with four through eight and outputs two and would remain in their previous state. A subsequent on-mask-Ø of CØH (11ØØ ØØØØ in binary) turns on outputs seven and eight.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol

23H Specifies the packet length
19H Indicates the modify outputs function code
off-mask-Ø to off-mask-15 Specifies a series of 16 eight-bit masks used to selectively turn off any or all of the controller's first 128 outputs. The masks are applied to successive banks of 8 outputs, with the least significant bit of the mask being applied to the lowest numbered output in the bank. A mask value of Ø turns the associated output off. A value of 1 does not change the output.
on-mask-Ø to on-mask-15 Specifies a series of 16 eight-bit masks used to selectively turn on any or all of the controller's first 128 outputs. The masks are applied to successive banks of 8 outputs, with the least significant bit of the mask being applied to the lowest numbered output in the bank. A mask value of 1 turns the associated output on. A value of Ø does not change the output.
Checksum Contains the complement of the modulo-256 sum of the previous 33 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
Checksum Contains the complement of the previous byte
FFH Signals the end of the message

Analog Input and Output Access Commands

The following commands allow you to read analog input and output states and change the value of an analog output. Input and output states are read individually.

Binary Protocol Conventions
The binary protocol uses specific conventions for specifying analog inputs and outputs, their values and for checksum error detection.

- When specifying an input or output the first input or outputs are specified as ØØH. The last input or output you can specify is 64. Its number is 3FH.
- The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description.

Reading an Analog Input - Command 29

Command 29 reads the value of any of the analog inputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
1DH Indicates the read analog input function code
Analog Input Specifies the input to be read, ØØH - FFH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes

FFH Signals the end of the message.

Format of Controller Response

Ø5H Specifies the packet length.
1EH Indicates the analog input value function code
LSB - MSB Contains the two-byte representation of the analog value, expressed as a number in the range of 0 - 10,000 decimal (ØØØØH - 271ØH), with the least significant byte transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 3 bytes
FFH Signals the end of the message

Reading an Analog Output - Command 31

Command 31 reads the value of any of the analog outputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
1FH Indicates the read analog output function code
Analog Output Specifies the output to be read, ØØH - FFH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø5H Specifies the packet length.
1EH Indicates the analog output value function code
LSB - MSB Contains the two-byte representation of the analog value, expressed as a number in the range of 0 - 10,000 decimal (ØØØØH - 271ØH), with the least significant byte transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 3 bytes
FFH Signals the end of the message

Changing an Analog Output - Command 33

Command 33 changes the value of any of the analog outputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø6H Specifies the packet length
21H Indicates the read analog output function code
Analog Output Specifies the output to be changed, ØØH - FFH

LSB - MSB Contains the two-byte representation of the analog value, expressed as a number in the range of 0 - 10,000 decimal (ØØØØH - 271ØH), with the least significant byte transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 4 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø5H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
9BH Checksum value. Contains the complement of the previous byte
FFH Signals the end of the message

Change Multiple Analog Outputs - Command 85

Command 85 changes the value of up to 64 sequential analog outputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø6H Specifies the packet length
55H Indicates the write multiple analog output function code
Analog Output Start Specifies the first output to be changed, Ø1H - FFH
Length Specifies the number of sequential analog outputs to change 01H - 40H
LSB – MSB First Contains the two-byte representation of the analog value, expressed as a number in the range of 0 - 10,000 decimal (ØØØØH - 271ØH), with the least significant byte transmitted first.
…
LSB-MSB Last
Checksum Contains the complement of the modulo-256 sum of the previous bytes
FFH Signals the end of the message.

Format of Controller Response

Ø5H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
9BH Checksum value. Contains the complement of the previous byte
FFH Signals the end of the message

Servo Access Commands

The following commands allow you to read a servo's position, error and auxiliary inputs.

Binary Protocol Conventions

The binary protocol uses specific conventions for specifying servo axes, their position and error, the state of a servo's auxiliary inputs, and for checksum error detection. You can perform these operations for servos axes 1 - 16.

- When specifying a servo, the first servo axis is specified as ØØH and the 16th specified as ØFH.
- The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description.

Reading a Servo's Position - Command 23

Command 23 reads the position of a servo.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
17H Indicates the read servo position function code
Servo Number Specifies the servo axis to be read , ØØH - ØFH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø7H Specifies the packet length.
18H Indicates the servo position function code
LSB, 3SB, Contains the four byte representation of the servos
2SB, MSB position. The value is expressed in 2's complement binary, with the least significant bye transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 5 bytes
FFH Signals the end of the message

Reading a Servo's Error - Command 47

Command 47 reads a servo's error.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
2FH Indicates the read servo error function code
Servo Number Specifies the servo axis to be read , ØØH - ØFH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø7H Specifies the packet length.
3ØH Indicates the servo position function code
LSB, 3SB, Contains the four byte representation of the servo's error.
2SB, MSB The value is expressed in 2's complement binary, with the least significant bye transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 5 bytes
FFH Signals the end of the message

Reading a Servo's Dedicated Inputs - Command 27

Command 27 reads the status of a servo's dedicated inputs. The controller returns the status of the dedicated input using a one bite code.

- Bit Ø, indeterminate
- Bit 1, Home input
- Bit 2, Start input
- Bit 3, Local/remote input
- Bit 4, Reverse limit input
- Bit 5, Forward limit input
- Bit 6, indeterminate
- Bit 7, indeterminate

Bit Ø is the least significant bit.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
1BH Indicates the read dedicated input status function code
Servo Number Specifies the servo axis to be read, ØØH - ØFH
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø7H Specifies the packet length.
1CH Indicates the servo dedicated input status function code
Status Contains a one byte of the servo's auxiliary input status.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Data Table Access Commands

The following commands allow you to read and change a data table's dimensions; read and change the value of a data table element; read the values in a data table row; and change the values in a data table row.

Binary Protocol Conventions
The binary protocol uses specific conventions for specifying rows and columns of a data table. The manner in which the row or column is specified varies with the command. The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description. The controller may return an error code under the following circumstances:

- The requested data table size is tool large for the controller.
- The requested data table size does not fit in the memory available when stored along with the Quickstep program.
- The command contains a data table column number greater than 32.

Reading a Data Table's Dimensions - Command 49

Command 49 reads the dimensions of a data table. The number of data table columns is ØØH to 2ØH.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø3H Specifies the packet length
31H Indicates the read data table dimensions function code
CEH Contains the checksum of the previous byte
FFH Signals the end of the message.

Format of Controller Response

Ø6H Specifies the packet length.
32H Indicates the data table dimensions function code
LSB, MSB Contains the number of data table rows in the current program, with the least significant bye transmitted first.
cols Contains the number of data table columns.
Checksum Contains the complement of the modulo-256 sum of the previous 4 bytes
FFH Signals the end of the message

Changing a Data Table's Dimensions - Command 51

Command 51 changes a data table's dimensions.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol

Ø6H Specifies the packet length
33H Indicates the change data table dimensions function code
LSB, MSB Contains the new number of data table rows, with the least significant bye transmitted first.
columns Contains the new number of data table columns.
Checksum Contains the complement of the modulo-256 sum of the previous 4 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
9BH Contains the checksum, complement of the previous byte
FFH Signals the end of the message

Reading a Data Table Value - Command 53

Command 53 reads the value of a specific data table element by specifying its row and column number.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø6H Specifies the packet length
35H Indicates the read data table location function code
LSB, MSB Contains the row number of data table element, with the least significant bye transmitted first.
columns Contains the column number of data table element.
Checksum Contains the complement of modulo-256 sum of the previous 4 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø5H Specifies the packet length.
36H Indicates the data table data function code
LSB, MSB Contains the data from the data table, expressed as a positive integer. The range is from 0 to 65,535 (decimal) with the least significant bye transmitted first.
Checksum Contains the complement of the modulo-256 sum of the previous 3 bytes
FFH Signals the end of the message

Changing a Data Table Value - Command 55

Command 55 changes the value of a specific data table element by specifying its row and column number.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø8H Specifies the packet length
37H Indicates the change data table location function code
LSB, MSB Contains the row number of data table element, with the least significant bye transmitted first.
columns Contains the column number of data table element.
LSB, MSB Contains the new value for the specified data table element. The new value can range from 0 to 65,535 (decimal) with the least significant bye transmitted first.
Checksum Contains the complement of modulo-256 sum of the previous 6 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
9BH Contains the checksum, complement of the previous byte
FFH Signals the end of the message

Reading a Data Table Row - Command 57

Command 57 reads the values in specific data table row and columns by specifying its row and column number.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø7H Specifies the packet length
39H Indicates the read data table row function code
LSB, MSB Contains the row number, with the least significant byte transmitted first.
First col Indicates the first data table column to read
Quantity Specifies the number of data table columns to read (n); <= 27 columns
Checksum Contains the complement of modulo-256 sum of the previous 5 bytes
FFH Signals the end of the message.

Format of Controller Response

Length Specifies the packet length, (n * 2) + 4, where n = number of columns read.
3AH Indicates the data table row data function code
Quant Specifies the number of data table columns read (n); <= 27 columns
For each of n locations
LSB, MSB Contains the data from the data table, expressed as a positive integer. The range is from 0 to 65,535 (decimal) with the least significant bye transmitted first.
End of location data

Checksum Contains the complement of the modulo-256 sum of the previous (n * 2) + 2 bytes
FFH Signals the end of the message

NOTE: If the number of data table columns specified extends beyond the actual number of columns the controller's response only contains data for the existing columns and the response will be shorter than expected.

Changing a Data Table Row - Command 59

Command 57 changes the values in specific data table row and columns by specifying its row and column number.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
length Specifies the packet length, (n * 2) + 4, where n = number of columns to be changed.
3AH Indicates the change data table row function code
LSB, MSB Contains the row number, with the least significant byte transmitted first.
First col Indicates the first data table column to change
Quantity Specifies the number of data table columns to change (n); <= 27 columns
For each of n locations
LSB, MSB Contains the data from the data table, expressed as a positive integer. The range is from 0 to 65,535 (decimal) with the least significant bye transmitted first.
End of location data
Checksum Contains the complement of modulo-256 sum of the previous (n * 2) + 5 bytes
FFH Signals the end of the message.

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
9BH Contains the checksum, complement of the previous byte
FFH Signals the end of the message

System and Controller Status Access Commands

The following commands allow you to read the status of a controller; start, stop or reset a controller; read or change the configuration of the controller's dedicated inputs; and obtain information about the number and type of controller resources in a particular controller.

Binary Protocol Conventions
The binary protocol uses specific bits for controller status and system configuration information. See the command descriptions for information on how to send and read this information. The checksum value is the complement of the previous byte(s). Some commands use the complement of the modulo-256 sum of the previous bytes; see the command description.

Reading a Controller's Current Status - Command 61

Command 61 reads a controller's status and reports if it is running, stopped, has a software fault, or is in programming mode.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø3H Specifies the packet length
3DH Indicates the read status byte function code
CEH Contains the checksum of the previous byte
FFH Signals the end of the message.

Format of Controller Response
Ø4H Specifies the packet length.
3EH Indicates the status byte function code
status Indicates the status of the controller, where:
Bit Ø = Ø if running and = 1 if stopped
Bit 1 = Ø in normal mode and = 1 in programming mode
Bit 2 = Ø if status OK and = 1 if there is a software fault
Bit 3 = Ø if in mid-program and =1 if fresh reset.
Bit Ø is the least significant bit and bits 4 through 7 are undefined.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Changing a Controller's Status - Command 63

Command 63 changes a controller's status.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
3FH Indicates the change controller status byte function code
status Indicates the status of the controller, where:
Bit Ø = Ø to start the controller and = 1 to stop it
Bit 3 = 1 to reset the controller and = Ø to continue
Bit Ø is the least significant bit and will always start or stop the controller. All unspecified and undefined bits should be set to Ø.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes

FFH Signals the end of the message

Format of Controller Response

Ø3H Specifies the packet length.
64H Contains the acknowledge function code (decimal 100)
Checksum Contains the complement of the previous byte
FFH Signals the end of the message.

Reading a Controller's System Configuration - Command 65

Command 65 reads the configuration of the controller's dedicated inputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø3H Specifies the packet length
41H Indicates the read system configuration function code
BEH Contains the checksum of the previous byte
FFH Signals the end of the message.

Format of Controller Response

Ø4H Specifies the packet length.
42H Indicates the system configuration function code
config Indicates the configuration of the controller, where:
Bit Ø = 1 if using input 1 for the start function
Bit 1 = 1 if using input 2 for the stop function
Bit 2 = 1 if using input 3 for the reset function
Bit 3 = 1 if using input 4 for the step function
Bit Ø is the least significant bit and bits 4 through 7 are undefined.
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Changing a Controller's System Configuration - Command 67

Command 67 changes the configuration of the controller's dedicated inputs.
Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
43H Indicates the change system configuration function code
config Indicates the new configuration of the controller, where:
Bit Ø = 1 to use input 1 for the start function
Bit 1 = 1 to use input 2 for the stop function
Bit 2 = 1 to use input 3 for the reset function
Bit 3 = 1 to use input 4 for the step function.

Bit Ø is the least significant bit and bits 4 through 7 are undefined.

Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes

FFH Signals the end of the message

Format of Controller Response

Ø3H Specifies the packet length.

64H Contains the acknowledge function code (decimal 100)

Checksum Contains the complement of the previous byte

FFH Signals the end of the message.

Listing Counts of Inputs, Outputs, Motion - Command 13

Command 13 obtains information about the number and type of controller resources and reports the information.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol

Ø3H Specifies the packet length

ØDH Indicates the I/O count request function code

F2H Contains the checksum of the previous byte

FFH Signals the end of the message.

Format of Controller Response

ØCH Specifies the packet length

ØEH Indicates the I/O count function code

flags Indicates the number of flags, typically 20H

inputs LSB Indicates the number of inputs, LSB: ØØH to F8H

inputs MSB MSB: ØØH to Ø4H

outputs LSB Indicates the number of outputs, LSB: ØØH to F8H

outputs MSB MSB: ØØH to Ø4H

stepping mtrs Indicates the number of stepping motor axes, ØØH to 1ØH

servos Indicates the number of servo axes, ØØH to 1ØH

analog inputs Indicates the number of analog inputs, ØØH to FFH

analog outputs Indicates the number of analog outputs, ØØH to FFH

Checksum Contains the complement of the modulo-256 sum of the previous 10 bytes

FFH Signals the end of the message

Listing Counts of Miscellaneous I/O - Command 69

Command 69 obtains information about the number and type of various controller resources, such as prototyping boards, high-speed counting boards, thumbwheel arrays, and numeric displays and reports it.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø3H Specifies the packet length
45H Indicates the miscellaneous I/O count request function code
BAH Contains the checksum of the previous byte
FFH Signals the end of the message.

Format of Controller Response

Ø7H Specifies the packet length
46H Indicates the I/O count function code
protos Indicates the number of flags, typically 2ØH
h s counters Indicates the number of high-speed counters
twhls Indicates the number of 4-digits thumbwheel arrays
disps Indicates the number of 4-digit numeric displays
Checksum Contains the complement of the modulo-256 sum of the previous 5 bytes
FFH Signals the end of the message

Reading Controller Step Status - Command 35

Command 35 reads the status of tasks in the controller. By executing this command four times, once for each group of eight tasks, you may obtain all the information necessary to reconstruct the hierarchy and status of the controller's tasks. In addition, if software fault has halted execution of your program, the controller's response indicates the type of the fault, the step where it occurred, and any relevant parametric data. As it starts each new task, your Quickstep program assigns a task number from 1 to 32. The main program is always task number one. Each of the 32 tasks, whether they are currently being used or not, reports back a step number along with a 32-bit mask word. If the program is currently using a task number, the mask shows whether the task in currently suspended, waiting for one or more sub-tasks to finish. This is shown by a 1 bit in the bit position of the mask word corresponding to the task for which the current task is waiting. For example, if the main program, task one, called up three sub-tasks, tasks two, three and four, the mask word for task one would be as follows:

ØØØØØØØØ ØØØØØØØØ ØØØØØØØØ ØØØØ111Ø MSB LSB

To extract the hierarchy of tasks being executed:

1. Start with task one and read its mask word to determine its sub-tasks.
2. Read the mask word of each sub-task, these indicate if any tasks are being executed a the next level down the hierarchy.
3. As you follow the hierarchy of tasks under execution, you may determine the current step being executed by each via the step number data provided. Step number are offset by -1

NOTE: Do not assume that Quickstep allocates task numbers in the order of task hierarchy. The starting and stopping of task numbers in a complex program may result in a scattering of active tasks through out the 32 possible task numbers. The only way to determine the active tasks is to follow the task hierarchy as outlined above. When a controller is stopped because of a software fault the message returned by the controller will contain a software fault code. A list of all fault codes can be found in the Fault Task Handler chapter.

Format of Message Sent to Controller

Ø1H Identifies the packet as using the CTC binary protocol
Ø4H Specifies the packet length
23H Indicates the status request function code
task range Bank of 8 tasks to be read, ØØH to Ø3H, where:
ØØH = tasks 1 through 8
Ø1H = tasks 9 through 16
Ø2H = tasks 17 through 24
Ø3H = tasks 25 through 32
Checksum Contains the complement of the modulo-256 sum of the previous 2 bytes
FFH Signals the end of the message

Format of Controller Response

39H Specifies the packet length
24H - 27H Indicates the controller status function code
Status - If the controller is stopped, it returns a value of ØFFH indicating true. Contains a value of ØØH indicating the controller is running
Fault type - Contains the type code for a software fault, if any are present. If the value is ØØH then no software fault is present.
NOTE: See the table on the previous page for a list of software fault codes.
Fault step – LSB, MSB, 16 bit, where where ØØØØH = step 1, ØØØ1H = step 2
LSB, 3SB, Data relating to software fault if any; otherwise
2SB, MSB unspecified. 48 bytes follow and provide the following data for each of the eight tasks:
LSB, MSB Step number currently being executed by this task, where ØØØØH = step 1, ØØØ1H = step 2, and so forth.
LSB, 3SB, 32 bit mask, indicating with a 1 or Ø for each of the 32
2SB, MSB possible tasks whether this task is waiting for the completion of each task or not. Lowest order bit of LSB represents task 1, etc.
Checksum Contains the complement of the modulo-256 sum of the previous 55 bytes
FFH Signals the end of the message

IP Encapsulation

An option exists which allows the CTC Binary Protocol to be sent over UDP and/or TCP, allowing it to be routed. All Blue Fusion controllers support the raw, low level, non-routable binary protocol, as well as run background servers listening for UDP and TCP connections which support "IP Encapsulation". Simply put, a header is added on to the current serial protocol. The controller listens for UDP requests on IP port 3000 and TCP on port 6000.

```
#define MAXPKTDATALEN 216
#pragma pack(1)
typedef struct ctcIPPacket_s
{

   // Used to validate proper CTC packet versions.
   //
   BYTE version_major;
   BYTE version_minor;

   // Identifier for each packet sent.  Used to validate incoming packets.
   //
   UINT16 transaction_id;

   // Required within packet.  Only the sender knows for sure the
   // type of the request.  The spare aligns data along word
   // boundaries.
   //
   BYTE type;
   BYTE spare;

   // Number of octets in the CTC binary.
   //
   UINT16 data_size;

   // Up to 216 (maximum in octets) of data.  Note : current maximum
   // packet size is 216 octets + 8 octets or 224 octets or bytes.
   //
   BYTE data[MAXPKTDATALEN];
} CTCPACKET;
#pragma pack()
```



The above structure is aligned on a 1 byte boundry. (#pragma pack(1)).

version_major/version_minor

These two byte fields represent the major and minor software revision of the initiator. The controller side simply returns whatever was received by the host making the request. Typically "version_major" = 0x04 and "version_minor" = 0x00.

transaction_id

The transaction_id is a two byte, little endian format (lsb/msb) field which contains an incrementing number, starting at 0x0001, to track the transaction request by. The controller will return the packet setting the transaction ID to that received, including the response information in the "data" field. Do not use a transaction id of 0x0000.

Type

0x14 – Request
0x15 – Reply

spare

Not used. Alignment purposes only. Set to 0x00.

data_size

This contains the length of the "data" field stored lsb/msb.
The maximum size of the "data" field is 216 bytes.

data

This is the binary protocol transaction which has been encapsulated. Refer to the standard CTC Binary Protocol Documentation. Messages from the host begin with 0x01, that from the controller are the length byte. Both message end with a checksum and 0xff byte. Only the number of bytes defined within "data_size" are contained within "data", not the full maximum of 216 bytes.

Example register read request of register 0x0002 with transaction ID 0x0001:

```
|---------------------- Header ------------------------|------------ Binary Protocol Msg -----------|
0x04  0x00  0x01  0x00  0x14  0x00  0x07  0x00  0x01  0x05  0x09  0x02  0x00  0xf4  0xff
```

checksum = ~(0x09 + 0x02 + 0x00) = 0xf4

Reply from controller:

```
|---------------------- Header ------------------------|--------------- Binary Protocol Msg ---------------|
0x04  0x00  0x01  0x00  0x15  0x00  0x08  0x00  0x07  0x0a  0x00  0x00  0x00  0x00  0xf5  0xff
```

Register contained 0x00000000. Note that little endian storage is used (lsb first).

CHAPTER

11

Quickstep 2 & QuickBuilder Symbols

This section discusses the symbol file generated by Quickstep 2 and the QuickBuilder tools. These symbols can be imported into HMI displays and used to symbolically monitor assigned registers. Additionally the recommended CTNet Binary Protocol commands for both legacy and newer controllers are discussed.

Quickstep 2 Symbol Table

Controllers: 2700, 5100, 5200, 5300.

The symbol format used by Quickstep 2 consists of an ASCII text file with tab delimited fields, each line representing a record entry. Each record is terminated by a 0x0d 0x0a combination. There are four fields:

TYPE – This field determines the resource type. It consists of a single bit set as follows:

1 – Constant
2 – Analog Input
4 – Analog Output
8 – Counter
16 – Data table column
32 - Display
64 – Flag
128 – Digital Input
256 - Stepper
1024 – Register
2048 – Servo
8192 – Step Name
65536 – Unknown Step name
131072 – Digital Output
524288 – Thumb Wheel

RESOURCE – Assigned resource number for access.

STATE – State references the active state, normally open or closed, only used for digital resources. If 0 then normally open and active state is a 1, if 1 then normally closed and active state is a 0.

NAME – Symbolic name.

Example:

```
1024    38      0       bZeroBatchCount
1024    39      0       bEditJob
1024    9       0       bTest
128     1       0       reot
128     12      0       buckleSensors
128     13      0       sawVFDStatus
131072          13      0       servoReset
131072          14      0       servoEnable
131072          15      0       runSaw
4       1       0       sawSpeed
4       2       0       cnvyrSpeed
```

Referencing the symbol 'sawSpeed', its TYPE is a 4, meaning Analog Output. The RESOURCE is '1', first analog output in the controller. STATE field is ignored since that is only for digital.

For HMI access purposes only the following TYPE fields are typically supported (32 bit integers), rest can be ignored:

2 – Analog Input (resource 1 to N)
4 – Analog Output (resource 1 to N)
64 – Flag (resource 1 to N)
128 – Digital Input (resource 1 to N)
1024 – Register (resource 1 to N)
131072 – Digital Output (resource 1 to N)

Quickstep 2 HMI Communications

Numerous commands are available within the CTNET Binary Protocol as described in the previous chapter.

Note: 2700, 5100, and 5200 controllers do not support Variants.

To simplify access all resources can be accessed by using registers, adding the resource number to the base value:

Register guide:
http://www.ctc-control.com/customer/techinfo/docs/5300_951/951-530006.pdf

Analog Input, base register 8500
Analog Output, base register 8000
Flag, base register 13200
Digital Input, base register 2000
Register, base register 0
Digital Output, base register 1000

For example, if the TYPE is a 2, designating an Analog Input, with a RESOURCE number of 5, reading register 8505 (8500 + 5) will result in the Analog Input value.

Referencing the 5300 Enhancements Overview, the primary commands of interest are:

Command	***Description***
9	Read a register
11	Change a register
75	Read a bank of 50 registers (limited from 1 to 1000)
77	Read a bank of 16 registers
87	Request random registers from list (not supported 2701E/2601)

QuickBuilder Symbol Table

Controllers: 5300. Two symbol table formats available, that described below as well as the Quickstep 2 table format for backward compatibility to tools like CTCMON. Note that the Quickstep 2 format has reduced symbolic information. The symbol file can be found in the project sub-directory with a '.sym' file extension. Two are created upon translation, that with the base name ending with '_QS2.sym' is in the Quickstep 2 format.

The symbol format used by QuickBuilder consists of an ASCII text file with fixed field sizes, padded with spaces, each line representing a record entry. Each record is terminated by a 0x0d 0x0a combination. There are eight fields:

SYMBOL – Symbolic name, starting in record position 1.

GROUP – Storage group, starting in record position 51. Available groups are:
double – 64 bit double in Microsoft format externally via CTC binary protocol, gcc internally (32 bit words swapped). Quickbuilder will reference this as a float (float in Quickbuilder world is actually 64 bits).
boolean – 32 bit integer with 0 or 1 value, false/true.
int – 32 bit integer.
string – Array of characters.

TYPE – Resource type, starting in record position 61. Available types are:
axis – Servo or stepper axis.

var – Variable, volatile.
nvar – Variable, non-volatile
din – Digital Input
dout – Digital Output
ain – Analog Input
aout – Analog Output
pid – PID Algorithm, where RESOURCE is input of PID and REGISTER is output of PID.

RESOURCE – Assigned resource number for access, starting in record position 71.

REGISTER – Assigned register number for access, starting in record position 81. Typically used instead of RESOURCE.

STORAGE – Storage type, starting in record position 91. Available types are:
scalar – Single item.
vector – One dimensional array.
table – Two dimensional array.
Note: Arrays are allocated dynamically thus size can change during runtime.

MODULE – Controller module model number referenced, for informational purposes only, starting in record position 101. All spaces if variable or pid.

SLOT – Controller slot module is expected in, for informational purposes only, starting in record position 111. All spaces if variable or pid.

Example:

```
buckleSensors                boolean   din      12       2012     scalar    M3-11     1
bZeroBatchCount              boolean   var      0        38       scalar
cnvyrSpeed                   int       aout     2        8002     scalar    M3-34     2
cnvyrSpeed                   int       ain      1        8501     scalar    M3-32     3
COFFEE_POT                   boolean   dout     3        1003     scalar    M3-10     4
conveyorVFDStatus            boolean   din      14       2014     scalar    M3-11     1
cSaw                         int       var      0        16       scalar
HEATER                       boolean   dout     2        1002     scalar    M3-10     4
nvar1                        double    nvar     0        36701    scalar
PID_PWM                      int       pid      8502     5903     scalar
PID1                         int       pid      8501     8001     scalar
PID2                         int       pid      8502     8017     scalar
PWM1                         boolean   dout     1        1001     scalar    M3-10     4
var1                         double    var      0        36101    scalar
```

QuickBuilder HMI Communications

Numerous commands are available within the CTNET Binary Protocol as described in the previous chapter.

Note: QuickBuilder makes extensive use of Variants which are only supported in the 5300 controller..

To simplify access all resources can be accessed by referencing the assigned REGISTER.

Referencing the 5300 Enhancements Overview, the primary commands of interest are:

Legacy Register Commands (scalar integers)

Command	***Description***
9	Read a register (integer only, else converts if a Variant)
11	Change a register (integer only)
75	Read a bank of 50 registers (integer only, 1 to 1000)
77	Read a bank of 16 registers (integer only)
87	Request random registers from list (integer only, else converts if a Variant)

Variant Commands (integers, floats, strings, scalar & arrays)

Command	***Description***
91	Get properties (only needed if dynamic array size needed)
93	Read a variant
95	Change a variant
109	Read a variant array block (consecutive)
113	Read a block of variants, randomly

Note: For optimized performance integer access should use the Legacy Register Commands. Variants can be of any type and have a greater overhead.