Clear
Sky
Institute

CSI Motion Controller

CSIMC Technical Reference Manual

Hardware Version C
Firmware Version 10311
Last changed August 19, 2000

This document prepared on Linux kernel version 2.2.12-20 with Applixware version 4.4.2.

Contents

1. Introduction 1
1.1. Requirements 1
1.2. Hardware Specifications 1
1.3. Software Specifications. 2
1.4. Example Configuration 2
1.4.1. Telescope Application: t-app 2
1.4.2. Focus Application: f-app 3
1.4.3. Dome Application: d-app 3
1.4.4. Monitors: csimc. 3
2. Circuit Board 4
2.1. Connections, Jumpers and Indicators. 4
2.1.1. Switches 4
2.1.2. Configuration Jumpers. 5
2.1.3. LEDs 5
2.1.4. Connectors 6
2.1.5. RS232 Port 7
2.1.6. Power Connections 8
2.1.7. Error in Rev C boards 8
2.2. Interface Circuitry 9
2.2.1. Typical Input circuit 9
2.2.2. Typical output circuit 9
2.2.3. Power supply 9
2.2.4. Encoder circuit 10
2.2.5. RS485 connector circuit. 10
2.2.6. Motor connector circuit. 11
3. Diagnostic Tools 12
3.1. Self-Test 12
3.1.1. Performing the Self-Test 12
3.1.2. Flashing Error Codes 13
3.2. Board Statistics Report 14
4. Network Application Protocol. 17
4.1. Shells and Threads 17
4.2. Shell Syntax 17
4.2.1. Comments 17
4.2.2. Data Types 17
4.2.3. Core Functions 18
4.2.4. Core Variables 18
4.2.4.1. Motion Variables 18
4.2.4.2. I/O Variables 20
4.2.4.3. System Variables 21
4.2.5. Local and Global Variables 21
4.2.6. Remote Variables 22
4.2.7. User Defined Functions 22
4.2.8. Expressions 23
4.2.9. Statements. 23
4.2.9.1. If caveat 24
5. Csimcd: Control Daemon 25
5.1. Usage 25
5.2. Configuration file 26
5.3. Log file 26

6. Csimc: Interactive Sessions 27
6.1. Usage 27
6.2. Commands 27
6.2.1. Command line editor 27
6.2.2. Downloading new firmware 28
6.2.3. Connection control 28
6.2.4. Serial port connection 28
6.2.5. History 29
6.2.6. Load script file 29
6.2.7. Session tracing 29
6.2.8. Help 29
6.2.9. Interruptions 29
6.2.10. Exiting 30
6.3. Configuration File 30
6.4. Sample Session 31
7. Csimcio: Script Interface 33
7.1. Usage 33
8. Programming Examples 34
9. Host API 39
10. Implementation Notes 41
10.1. Datalink Protocol 41
10.1.1. Data Packet Format 41
10.1.2. Topology 42
10.1.3. Operation 42
10.1.4. Token Packet Format 43
10.2. Power-On 43
10.2.1. LAN Firmware updates 43
10.3. Development Tools 44
10.3.1. Debugger Pod 44
10.3.2. Embedded C Compiler 45
10.3.3. Downloading a new CSIMC 45
10.4. Equation of Motion 45

1. Introduction

The motivation behind the CSIMC was to replace expensive bus-specific motion controller cards with a network of small generic circuit boards each capable of handling all aspects of one axis of motion. The full set of requirements is as follows. Clear Sky Institute, Inc., is proud to offer the CSIMC as it is our belief the CSIMC meets each of these requirements.

1.1. Requirements

- generate smooth step and direction signals suitable for steppers and digital servo controllers
- read any quadrature balanced or single-ended encoder
- provide several general purpose TTL input and output lines
- provide several channels of general purpose analog-to-digital inputs
- simple UTP cabling to connect together exactly the number of boards needed in an application
- one RS232 host connection for controlling the entire collection
- small size and low power so boards can be placed near each axis to simplify wiring and installation
- very powerful and general purpose programmability directly on board
- session-based communication between host and each board
- session-based communication directly among boards for coordinated actions
- off-load real-time demands from host
- on-board arbitrary trajectory pursuit and tracking algorithm
- host access from terminal, C program API or from scripting languages such as *perl*, *sh* or others
- provide for manufacturing test, installation and field troubleshooting without test equipment
- maximum use of surface mount devices for high producibility

1.2. Hardware Specifications

Follows are the hardware specifications of the CSIMC.

- step and direction square wave TTL outputs; programmable from 0.01 to 1,000,000Hz; position accuracy ±0 counts; velocity precision 0.01 Hz, absolute accuracy 100 PPM 0-70° C, relative accuracy the greater of 1 part per 16 million or 0.0037 Hz (crossover at 62KHz), less than 10 PPM drift in 30 days[1]; square wave duty cycle 50% ±5%.
- incremental quadrature encoder input; jumper selectable as either two balanced pairs (A+/A- and B+/B-) or two single-ended inputs (A+ and B+ referenced to common); full-time error detection; can accept voltages up to ±12V. Count rate to 4,000,000Hz.
- 8 misc input control lines, each opto isolated; accepts either external switch closure to ground or TTL input source.
- 8 misc output control lines, each opto isolated; jumper configurable as either shorting two isolated pins or supplying TTL output.
- 8 independent 8-bit Analog-to-Digital converters, each capable of sampling 0..5VDC input in 60µS.
- any one board connects to host computer using an RS232 connection running 38.4 kbits/s.

[1]from CST Reeves MX045 clock oscillator data sheet

- up to 31 additional boards may be connected using an isolated 2-wire balanced UTP RS485 bus operating in multidrop mode using a synchronous Token-ring datalink protocol running at 38.4 kbits/s. Max cable length is 1,000 ft.
- each board is assigned one of 32 unique network addresses from 0..31 using a 5-circuit DIP switch.
- board boots instantly from onboard FLASH.
- FLASH program can be upgraded in the field while on network
- 27 LED status indicators for easy in-field installation and diagnostics.
- power-on self-tests check all major subsystems.
- 2 separate isolated power systems, one for on-board logic and one to supply external I/O devices such as an encoder or home switches. These may be combined if desired to power the board from one source. On-board regulators allow powering from 7-12VDC unregulated or from regulated 5VDC. Logic draws 200 mA. External can supply regulated 5V up to 300ma.
- operating temp 0 .. 105 ° F still-air ambient at non-condensing humidity.
- dimensions 76 x 152mm (3 x 6 inches), height 15 mm (0.6 inch).

1.3. Software Specifications

- Synchronous token-ring datalink protocol.
- host may talk with any node, and nodes may talk directly with each other.
- multi-tasking operating system, supporting multiple simultaneous threads of execution.
- host support daemon process provides separate Internet socket to each thread on each board.
- fully user programmable in very-C-like language with on-board JIT compiler.
- nodes may get or test other node’s variables.
- step rate is recomputed once every 32 ms.
- arbitrary trajectory may be downloaded and tracked entirely on-board; defined as up to 200 positions *vs* time, cubic spline interpolated; may use encoder or motor steps.

1.4. Example Configuration

This section discusses a typical application. To avoid getting lost early in the details, we only describe the overall situation and how the CSIMC boards are used in a high-level way to give a flavor for the approach used in solving motion applications. After studying this manual, you will be equipped to configure and write the supporting software to effectively apply CSIMC boards to your exact application.

In this example, four CSIMC nodes are used. Two are connected to servo motors on the main axes of a telescope, with each including an encoder, two limit switches and one home switch. One node is connected to operate a focuser with a stepper motor and home switch. The fourth is operating a dome, with a bidirectional AC motor, encoder and a home switch.

The host computer is running several processes which are in communication with the nodes. The main application work is assigned to three processes, one each for the telescope, focuser and dome. Two copies of *csimc* are also shown to indicate that several instances of this tool may be running at one time connected to any set of nodes for monitoring purposes or to override control if desired.

1.4.1. Telescope Application: t-app

The telescope application uses nodes addressed at 1 and 2 to control the main axes. It would likely begin a session by finding the home positions of each axis, using a function saved in a configuration script. If this were an initial installation, the limit positions might then be found next and saved. To perform an observation, an ephemeris would be computed and a table of positions over time would be loaded into each node. The nodes would then acquire and track this path entirely on their own.

1.4.2. Focus Application: f-app

The focus application operates a focuser connected to a stepper motor on Node 3. It assumes the home switch is at one extreme end of permissible travel. This fact is used to find home from an unknown starting position upon start-up without the need for limit switches. From then on, the application issues the desired focus position as desired, such as when the temperature or a filter changes.

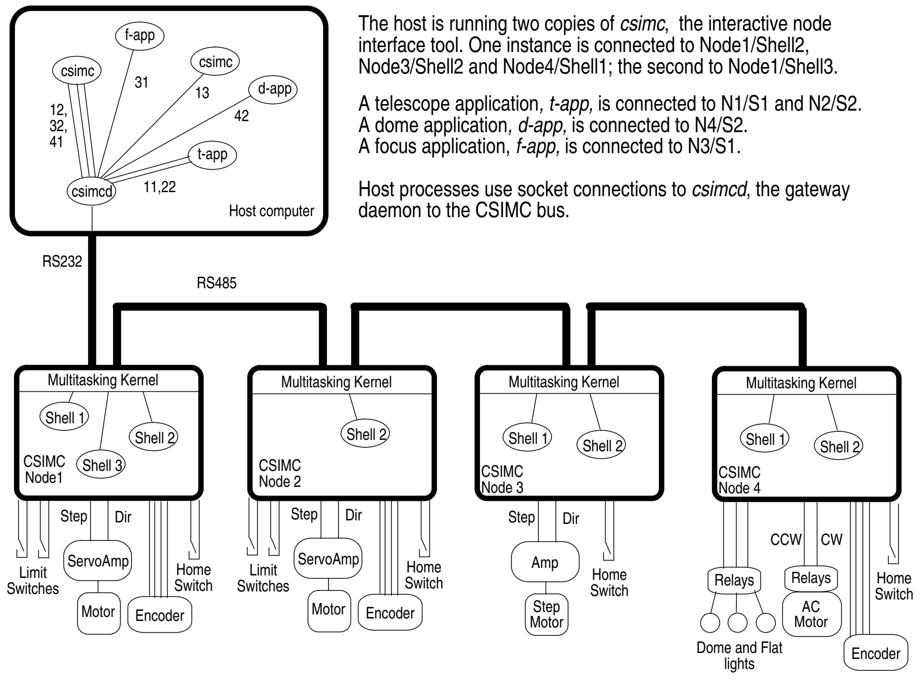
1.4.3. Dome Application: d-app

This dome application is charged with maintaining the dome opening in accord with the telescope pointing position. Node 4 is used. On powerup, the dome encoder would be calibrated with the home switch. The dome is rotated by commanding an AC motor to rotate clockwise or counterclockwise. These are issued to maintain the dome azimuth to within a specified degree of accuracy determined by slit opening and the geometry of the mount.

Also configured are a dome work light and two lights used for creating camera flats. These can be controlled from Node 4 also.

1.4.4. Monitors: csimc

Perhaps one is being used interactively by an operator to check on several nodes. Perhaps the other is running from a script to collect motor *vs.* encoder data to investigate drive slippage.



The host is running two copies of *csimc*, the interactive node interface tool. One instance is connected to Node1/Shell2, Node3/Shell2 and Node4/Shell1; the second to Node1/Shell3.

A telescope application, *t-app,* is connected to N1/S1 and N2/S2.
A dome application, *d-app,* is connected to N4/S2.
A focus application, *f-app,* is connected to N3/S1.

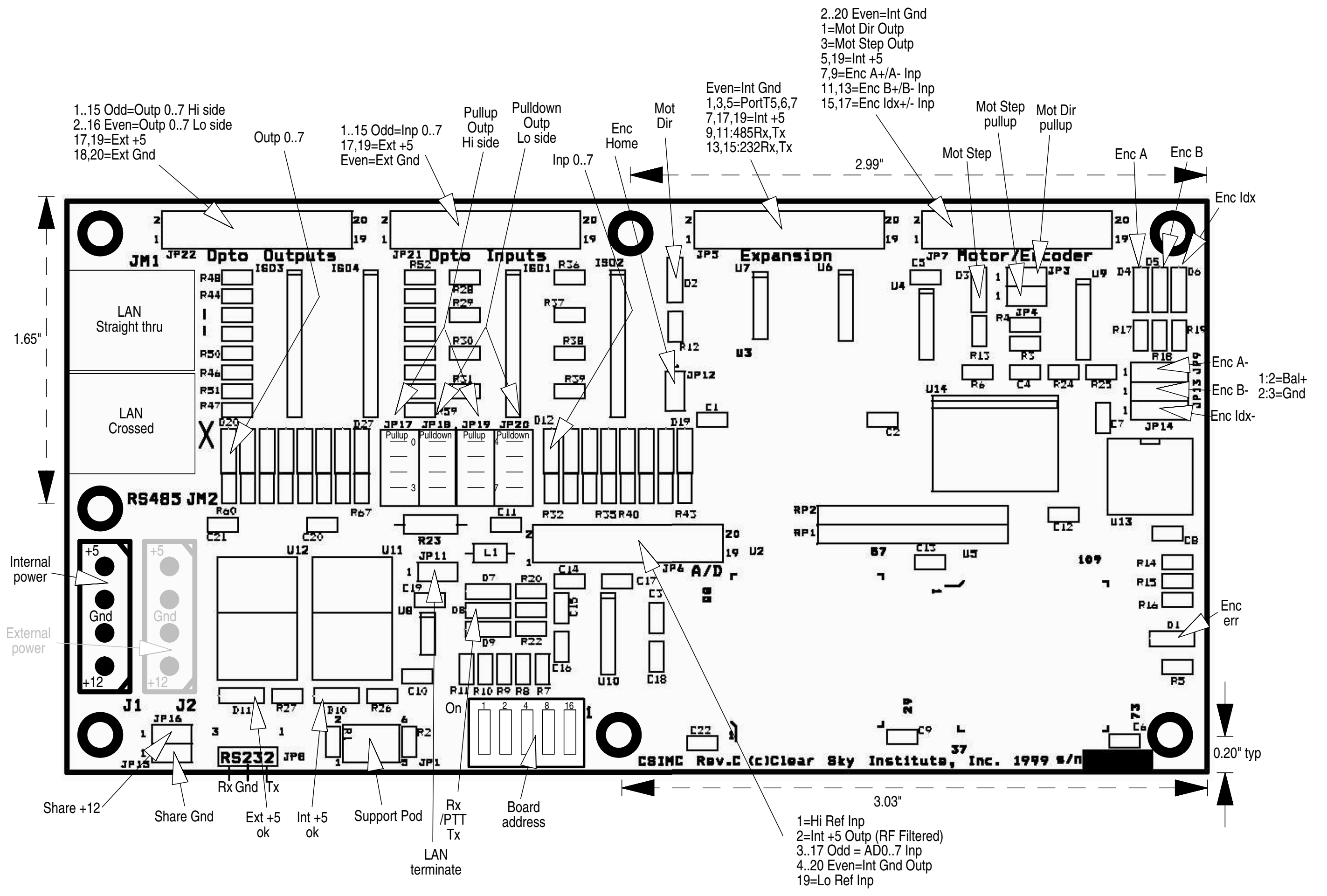
Host processes use socket connections to *csimcd*, the gateway daemon to the CSIMC bus.

Four CSIMC controllers are shown, addressed as 1, 2, 3 and 4. Nodes 1 and 2 are running servo motors with independent encoders and limit and home switches. Node 3 is running a stepper motor on a focuser. Node 4 is operating a rotating dome, including work and flat lights.

2. Circuit Board

This section describes the CSIMC circuit board.

2.1. Connections, Jumpers and Indicators



2.1.1. Switches

Switch	Description
S1 Board address	1..5 == 2^0.. 2^4, **LSB on left, "1" is up,** Open=0 Closed=1 Set to a unique value on each board from 0..31. 0 00000 8 00010 16 00001 24 00011 1 10000 9 10010 17 10001 25 10011 2 01000 10 01010 18 01001 26 01011 3 11000 11 11010 19 11001 27 11011 4 00100 12 00110 20 00101 28 00111 5 10100 13 10110 21 10101 29 10111 6 01100 14 01110 22 01101 30 01111 7 11100 15 11110 23 11101 31 11111

2.1.2. Configuration Jumpers

Jumper	Description, when installed
JP3	Add pullup to Motor Direction output
JP4	Add pullup to Motor Step output
JP6	RS232: 1=Tx 2=Gnd 2=Rx
JP9	Encoder A-: shunt 1-2 for balanced, 2-3 grounded
JP11	Terminate RS485 LAN .. use on first and last in chain.
JP12	Route encoder index to Input bit 0 for use as home signal
JP13	Encoder B-: shunt 1-2 for balanced, 2-3 grounded
JP14	Encoder Idx-: shunt 1-2 for balanced, 2-3 grounded
JP15	Share Int and Ext +12 busses
JP16	Share Int and Ext ground busses
JP17	Pull up Output Hi side to +5. Top..down = Outputs 0..3
JP18	Pull down Output Lo side to Gnd. Top..down = Outputs 0..3
JP19	Pull up Output Hi side to +5. Top..down = Outputs 4..7
JP20	Pull down Output Lo side to Gnd. Top..down = Outputs 4..7

2.1.3. LEDs

LED	Description, when lit
D1	Encoder error
D2	Motor direction is negative (JP7/1 is 0V)
D3	Motor step (JP7/3 is 0V)
D4	Encoder A
D5	Encoder B
D6	Encoder Index
D7	RS485 receive data
D8	RS485 PTT off
D9	RS485 transmit data
D10	Internal +5 bus voltage ok
D11	External +5 bus voltage ok
D12	Input 0 is @ 0V or closed to Ext ground
D13	Input 1 is @ 0V or closed to Ext ground
D14	Input 2 is @ 0V or closed to Ext ground
D15	Input 3 is @ 0V or closed to Ext ground
D16	Input 4 is @ 0V or closed to Ext ground
D17	Input 5 is @ 0V or closed to Ext ground
D18	Input 6 is @ 0V or closed to Ext ground
D19	Input 7 is @ 0V or closed to Ext ground
D20	Output 0 is +5V wrt Ext, or open circuit
D21	Output 1 is +5V wrt Ext, or open circuit
D22	Output 2 is +5V wrt Ext, or open circuit
D23	Output 3 is +5V wrt Ext, or open circuit
D24	Output 4 is +5V wrt Ext, or open circuit
D25	Output 5 is +5V wrt Ext, or open circuit
D26	Output 6 is +5V wrt Ext, or open circuit
D27	Output 7 is +5V wrt Ext, or open circuit

2.1.4. Connectors

Connector	Pin Assignments
J1 Internal power (PC compatible)	1: +12 2: Gnd 3: Gnd 4: +5V
J2 External power (PC comparable)	1: +12 2: Gnd 3: Gnd 4: +5V
JP1 BDM Pod (*for factory use only*)	1: BKGD mode 2: Logic ground 3: Logic +5 4: /RESET 5: Logic ground 6: Logic +5
JP5 Expansion	2..20 Even: Internal ground 1: PortT5. Shunt to 2 to run Selt-Test on powerup 3: PortT6. Shunt to 4 for RS232 aux on powerup 5: PortT7 7: Internal +5 9: RS485 Receive 11: TS485 Transmit 13: RS232 Receive 15: RS232 Transmit 17: Internal +5 19: Internal +5
JP6 A-to-D	4..20 Even: Internal ground output 1: Hi reference input (hint: connect to 2) 2: Internal +5V output, RF filtered 3: Channel 0 input 5: Channel 1 input 7: Channel 2 input 9: Channel 3 input 11: Channel 4 input 13: Channel 5 input 15: Channel 6 input 17: Channel 7 input 19: Lo reference input (hint: connect to 20)
JP7 Motor Encoder (see also JP9, JP12, JP13, JP14)	2..20 Even: Internal ground 1: Motor Direction output 3: Motor Step output 5: Internal +5 output 7: Enc A+ input (or single-ended A) 9: Enc A- input (not used with single-ended) 11: Enc B+ input (or single-ended B) 13: Enc B- input (not used with single-ended) 15: Enc Index+ input (or single-ended A) 17: Enc Index- input (not used with single-ended) 19: Internal +5 output
JP8 RS232 see §2.1.5.	1: TX 2: Logic ground 3: RX

Connector	Pin Assignments (cont.)
JP21 Opto-isolated Inputs	2..20 Even: External ground 1: Input Bit 0 3: Input Bit 1 5: Input Bit 2 7: Input Bit 3 9: Input Bit 4 11: Input Bit 5 13: Input Bit 6 15: Input Bit 7 17: Ext +5 19: Ext +5
JP22 Opto-isolated Outputs	1: Output Bit 0, HI side 2: Output Bit 0, Lo side 3: Output Bit 1, HI side 4: Output Bit 1, Lo side 5: Output Bit 2, HI side 6: Output Bit 2, Lo side 7: Output Bit 3, HI side 8: Output Bit 3, Lo side 9: Output Bit 4, HI side 10: Output Bit 4, Lo side 11: Output Bit 5, HI side 12: Output Bit 5, Lo side 13: Output Bit 6, HI side 14: Output Bit 6, Lo side 15: Output Bit 7, HI side 16: Output Bit 7, Lo side 17: Ext +5 18: Ext ground 19: Ext +5 20: Ext ground
JM1 RS485 straight	3: B 4: A 1,2,5,6: unused
JM2 RS485 crossed	3: A 4: B 1,2,5,6: unused

2.1.5. RS232 Port

The RS232 connector, JP8, serves two purposes. On exactly one board in a CSIMC network it will function as the gateway to the computer. On other boards it is available as a general purpose RS232 communication port from the API (see §9. and §6.2.4.). CSIMC tries to autosense which role the RS232 is playing but if the device connected starts transmitting immediately on powerup the role can be difficult to discern. In this case, insert a shunt in JP5 connecting pins 3-4 to force CSIMC to never consider the RS232 as a gateway to the host.

2.1.6. Power Connections

The CSIMC has on-board power regulators and filtering which allows the board to be powered by relatively unregulated 7-12 VDC. Or the regulators may be bypassed so the board may be powered directly using clean well-regulated 5VDC. In either case, the total current draw by the components on-board is approximately 200mA, for a total power dissipation of about 1 W.

The CSIMC is designed with separate power distribution buses. One supplies on-board circuitry and is referred to as the **Internal** power bus. The other is used to power off-board components and is referred to as the **External** bus. Separating the two power buses reduces the chances that the CSIMC will be damaged due to external voltage surges, wiring errors, or other unusual occurrences. There is a separate power connector and regulator for each bus. Jumpers are available to allow using one supply to power both buses, if desired.

2.1.7. Error in Rev C boards

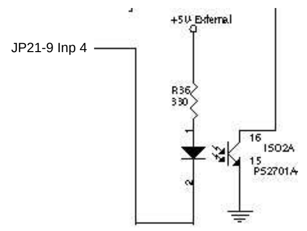
Unfortunately, due to a layout error in of Rev C of the board, only one power connector would fit, J1. This happens to be the Internal power bus. This results in the following rules:

1. To power the board with one +12V supply, install *both* jumpers JP15 and JP16. A small heatsink should be attached to each regulator.
2. To power the board with both +12V and +5 as from a typical PC power supply connector, install *both* jumpers JP15 and JP16. A heat sink will not be required.
3. To power the board with one +5V supply, solder a wire on the rear of the board connecting J1-1 and J2-1 (pin 1 is on the side nearest the jumpers JP15 and JP16) and install *only* jumper JP15.
4. To power the board with separate supplies of either voltage, the second (External bus) supply must be wired directly to the PC board in place of J2. In this case, proceed as if nothing were wrong and choose jumpers as desired as per §2.1.2. If powering from 12V a small heatsink should be attached to each regulator.

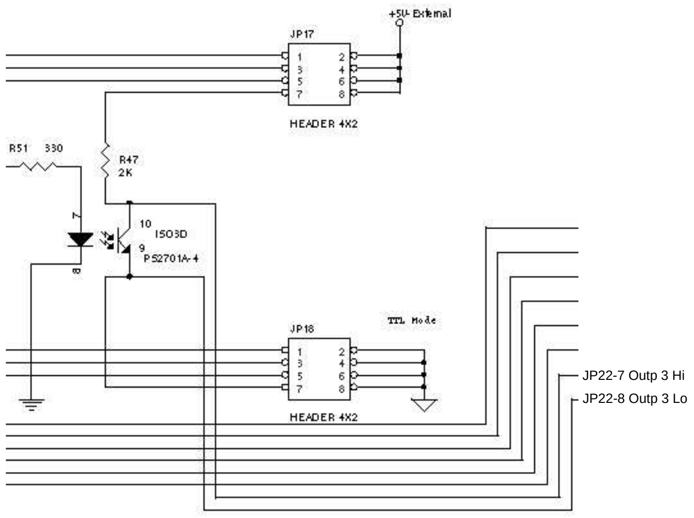
2.2. Interface Circuitry

The following portions from the schematic of the CSIMC precisely indicate the circumstances of each connector.

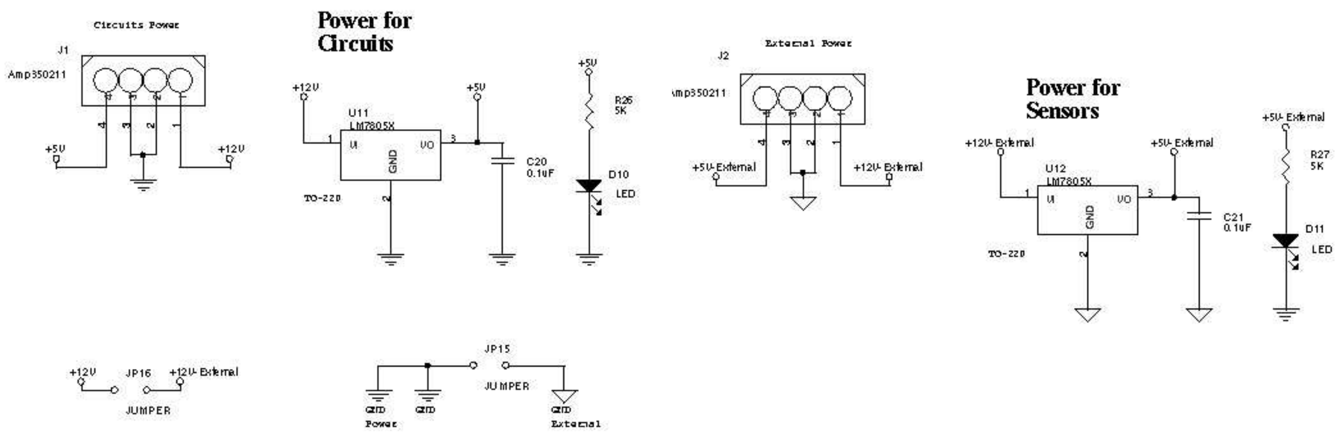
2.2.1. Typical Input circuit



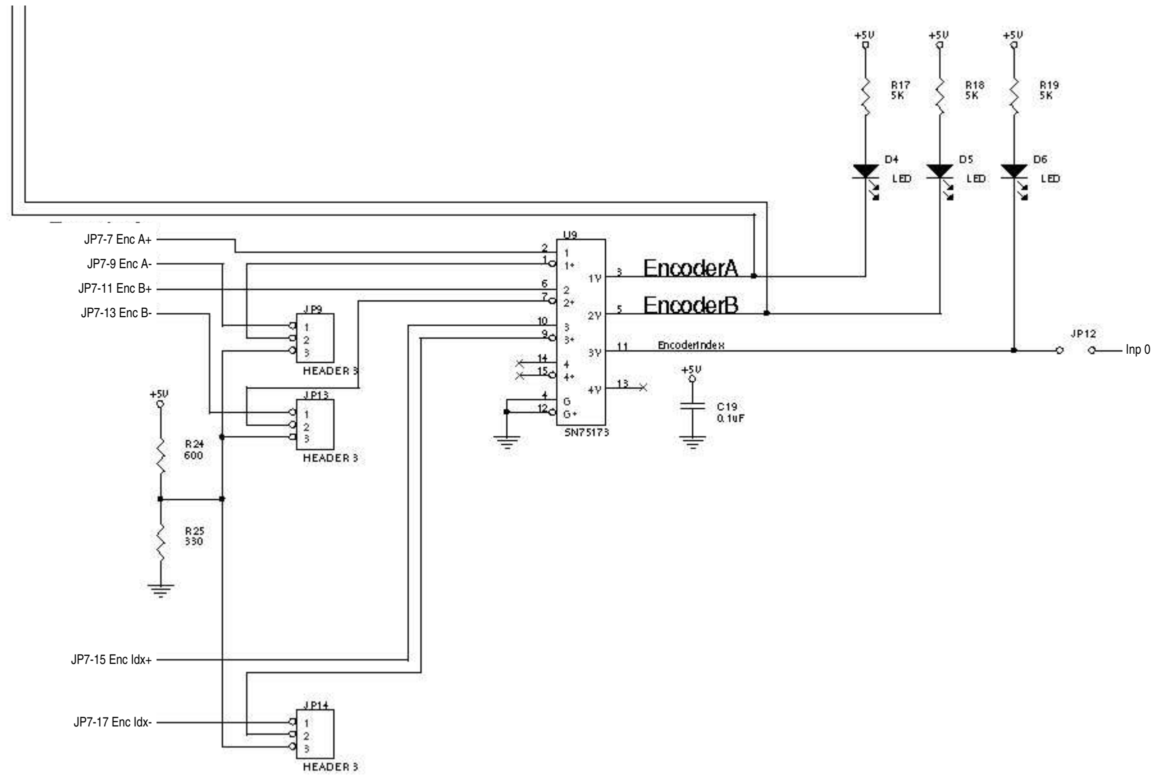
2.2.2. Typical output circuit



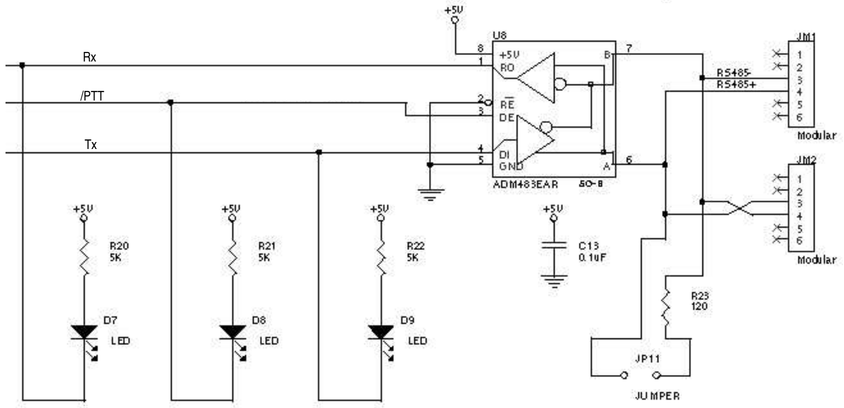
2.2.3. Power supply



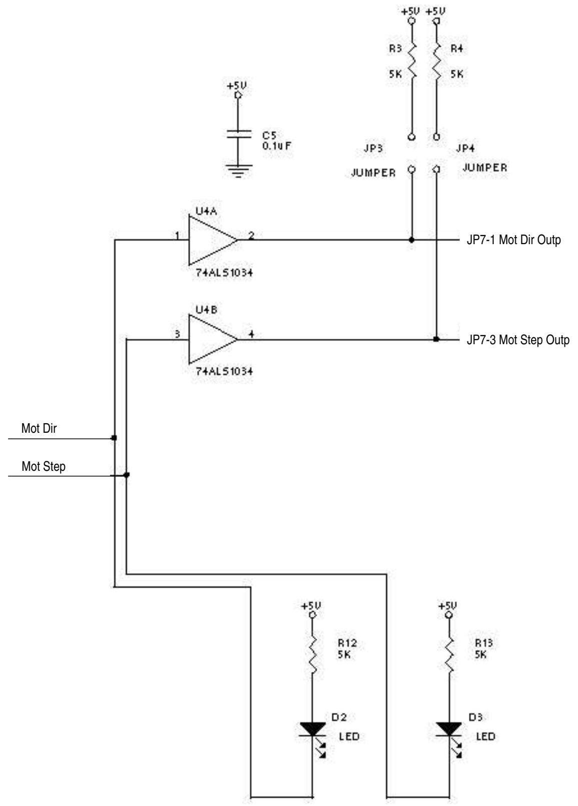
2.2.4. Encoder circuit



2.2.5. RS485 connector circuit



2.2.6. Motor connector circuit



3. Diagnostic Tools

The CSIMC comes equipped with a set of core diagnostics which operate without need for a host connection. Additional tests are possible if the unit is connected to a host computer. These tests are designed to ascertain the basic integrity of a CSIMC operating both stand-alone and while operating live in a networked application. What follows is a description of these capabilities.

3.1. Self-Test

Each CSIMC has the ability to test its own major subsystems without additional test equipment. All that is required is for power to be applied and one jumper be inserted temporarily to activate the tests. If an encoder is used with the board, it should also be installed during the test.

The test runs by itself but does require human monitoring since many of the checks indicate status by using the on-board LED indicators. The entire test repeats indefinitely until power is cycled. If an intermittent failure is suspected, those tests which can detect their own failure, such as bad memory, will stop at the point of failure and flash a pattern of lights forever. In this way, the tests may be left running unattended and checked as desired. A table of error patterns follows in § 3.1.2.

3.1.1. Performing the Self-Test

Proceed as follows to perform the self-test. Before you begin, have one spare .1" jumper handy. The instructions assume the board is oriented so the long side with the address selection DIP switch is nearest you.

1. Disconnect power.
2. Unplug all connectors, except the encoder if applicable.
3. Install the jumper on JP5 connecting pins 1 and 2.
4. Reconnect power.
5. Find D10 and D11, in the bottom left near the power connectors. D10 indicates internal power is OK and D11 indicates external power is OK. Both these LEDs should always be lit.
6. Find D3, in the top right. This is the Motor Step output monitor. It should be pulsing rapidly at about 10 flashes per second. If you have an oscilloscope handy you should measure a square wave on on JP7-3 with a period of 102.3 ms.
7. Find D2, at the top center. This is the Motor Direction output monitor. It should be flashing 16 times slower, one full cycle every 1.64 seconds.
8. Find D1, in the lower right. This is the Encoder Error monitor. It should be dark if there is a functioning encoder installed or lit if there has been an encoder error. Encoder error is determined by watching the 2 encoder quadrature input lines. Since these should only change in a 4-state gray-code transition sequence, any deviation from this sequence is an error.
9. Find D4-6, three LEDs in the top far right. If there is an encoder attached, rotate it slowly. You should see D4 and D5 flash in step with the quadrature pattern from the encoder signals. Depending on the precision of your encoder, only a slight rotation is needed to see the change. Don't try too hard to see the exact pattern, just see if they each seem to be following a repetitive alternating pattern; correct counting is confirmed in the next step. If your encoder also generates an index pulse and you can position it to trigger this output, D6 will light when the index is active. This is very hard to do.
10. Find D16-19, the right 4 of a group of 8 LEDs in the center of the board. As the encoder is rotated these 4 LEDs count the steps; the number should smoothly increase while rotating one way and decrease the other way. Note that the bits are reversed, *i.e.*, the Least Significant Bit (LSB) is on the left. If an encoder is not attached, D16-19 should remain dark at all times.
11. Find D12-15, the left 4 LEDs in this group. These are serving as a 4-bit binary counter and are continuously counting motor steps. The LSB is again on the left. The count will start all off at 0, and count

up to 31 (all on) in 1.5 seconds then count back down to 0 and repeat. Take careful note that all lights get lit in their turn and are following a correct binary sequence, up then down. Also note that while counting up, D2 should be Off.

12. Find D23-27, the right 5 LEDs in the other row of 8 toward left center. These are controlled by the board address DIP switch. The switches labeled 1-5 correspond to LEDs D23-27, in order. Toggle each switch and confirm that the corresponding LED is On when the switch is Closed.
13. Find D20-22, the left 3 LEDs in this group. These are reporting the progress of a continuous memory test. These LEDs should count up from 0 through 7 then repeat from 0. Note that the LSB is on the left. Each cycle requires approximately 3 seconds.
14. Find D7-9, just above the address DIP switch in the lower center. These indicate the status of the RS485 LAN test. The self-test repeatedly sends the character ’P’ (0x50) at 300 Baud, 8/N/1, and arranges for a loopback to test both the transmitter and receiver. During each test, D8 should be lit for about 1/10 second, then it should go out and D7 and D9 should flash briefly. This pattern should repeat every 2.6 seconds.
15. If you have a way to connect to the RS232 port with a simple terminal or terminal emulator program (such as *seyon* or *minicom* in Linux or *terminal* in Windows), set it up for 38.4KB, 8/N/1. You should see the character ’P’ about 2.5 times each second. If you send a character, you should see it echoed.

Remove the self-test activation jumper JP5-1/2 and disconnect power. This concludes the self-test. If all tests passed, the CSIMC is likely in good working order.

3.1.2. Flashing Error Codes

When an error occurs during a self test that can be detected automatically, LEDs D20-27 will flash an 8-bit binary number until JP5-1/2 is removed. Note that the LSB is on the *left* (D20). Some of these tests are also performed during normal operation in which case they only flash the code a few times then the CSIMC is automatically rebooted. If these ever occur, please contact CSI with the flash count and the circumstances by which it can be recreated. The codes are defined as follows:

Value	D20-27 Pattern	Description
1	10000000	Memory bit stuck on
2	01000000	Memory bit stuck off
3	11000000	Memory restore failed
4	00100000	RS232 transmitter stuck on
5	10100000	RS485 transmitter stuck on
6	01100000	RS485 loopback timed out
7	11100000	RS485 loopback received incorrect character back
8	00010000	FLEX failed to request program load
9	10010000	FLEX failed to accept program
10	01010000	Motor steps running but at incorrect rate
11	11010000	Motor steps are not running
12	00110000	Illegal instruction fault
13	10110000	Bad opcode
14	01110000	scheduler error
15	11110???	stack overflow. upper 3 bits is location code.
16	00001000	boot loader fails
17	10001000	scheduler fails
18	01001000	variable assertion fails
19	11001000	Boot loader fails writing EEPROM
20	00101000	Boot loader fails writing FLASH
21	10101000	Boot loader fails erasing FLASH

3.2. Board Statistics Report

One of the core functions (see § 4.2.3.) is **stats()**, which reports a variety of statistics pertaining to the operation of the CSIMC when it is up and running. What follows is a typical display and a table explaining each field. In general, be wary of high Rx and Tx error rates and low memory availability. Packet stats do not include Token packets (see §10.1.4.)

```
   Node: 10305 version 111 FLEX  00001=1 myaddr
   Time:    96 motlat   3% COP    0:01 dT    11:30 up
 Memory: 11918 + 1200 + 406 = 13524
    abs:    38 bytes     1 args      0 tmpv
    max:    35 bytes     2 args      0 tmpv
    min:    35 bytes     2 args      0 tmpv
   sign:    42 bytes     1 args      0 tmpv
  pause:    61 bytes     1 args      1 tmpv
  RS232:    11 sent     11 rcvd      0 noisy     0 frame     0 ovrrn     0 stray
  RS485: 32845 sent  23767 rcvd      0 noisy     0 frame     0 ovrrn     0 stray
Tx Pkts:    33 total    33 1stry     0 retry     0 fail
Rx Pkts:    65 total    65 good     30 mine      0 dups      0 rogue
Peer 35:     5 TId       3% CTStk   15% RTStk   66% CStk   0:00 up
Peer 34:     6 TId      12% CTStk   15% RTStk   71% CStk   0:01 up
Peer 33:     7 TId       0% CTStk    0% RTStk   18% CStk   0:00 up    (232)
```

Row	Field	Description	Nominal Operation
Node		Info about the node state	
	version	Software version number: Model Rev Version	1RRVV
	FLEX	Version of Altera FLEX microcode	>= 111
	myaddr	Board address, in decimal and binary	$11111_2 = 0..31_{10}$
	Gateway	Present only if node is the host gateway	optional
Time		Parameters related to CSIMC time and ID	
	***motlat**	maximum motion update loop latency; ms	< 35
	***COP**	Computer Operating Properly watchdog time allocation used, percent	< 75
	***dT**	time since last stats() report, HHHH:MM	desired stats range
	up	time since CSIMC was booted, HHHH:MM	large ☺
Memory		Info about memory	
	Heap	Sorted list of free memory blocks and sum, in bytes.	Best if first dominates sum.
Functions		One row for each User-defined function on this node,	if any
	bytes	bytes used to store this function definition	any
	args	number of arguments to this function	0..9
	tmpv	temporary variables ($0, ...) used by this function	0..9
RS232		RS232 host port stats	
	***sent**	characters transmitted	any
	***rcvd**	characters received	any
	noisy	noise on any start, data or stop bit	few if any
	frame	zero detected rather than stop bit	few if any
	ovrrn	new byte received before current was dispatched	few if any
	stray	unexpected interrupt	few if any
RS485		RS485 host port stats	
	***sent**	characters transmitted	any
	***rcvd**	characters received	any
	noisy	noise on any start, data or stop bit	few if any
	frame	zero detected rather than stop bit	few if any
	ovrrn	new byte received before current was dispatched	few if any
	stray	unexpected interrupt	few if any
Tx Pkts		Packets transmitted by this node to either port	
	*†**total**	packets sent from this node, sans ACKs	any
	*†**1stry**	packets acknowledged on first try	vast majority of total
	*†**retry**	packets acknowledged after some retries	few if any
	*†**fail**	packets that were never acknowledged	0
Rx Pkts		Packets received by this node from either port	
	*†**total**	total Sync bytes seen	any
	*†**good**	total packets seen with correct checksum	vast majority of total
	*†**mine**	packets seen addressed to this node, sans ACKs	any
	*†**dups**	packets seen more than once, ie, our ACKs not seen	few if any
	*†**rogue**	packets with proper checksums but bogus fields	0
Threads		One row for each Thread running on this node	
	Peer	Network address with which this thread talks	32..62
	TId	thread ID	0..24
	CTStk	Compile-Time stack, % used of total mem available	< 75%
	RTStk	Run-Time stack, % used of total mem available	< 75%
	CStk	C stack, % used of total mem available	< 75%
	up	time alive, *HHHH:MM*.	any
	(232)	Present only if this thread is handling RS232 traffic	optional

* These fields are reset after each report.
† These fields are clamped to a maximum value of 65535.

4. Network Application Protocol

Each CSIMC and the host computer are connected together on a logical bus. Each node is assigned a unique address in the range 0..31. The host acts as though it were up to 32 additional *virtual* nodes with addresses 32..63. Each node may communicate directly with any other node. The protocol is a connectionless datagram packet system which means no setup or takedown is required to communicate from one node to another.

4.1. Shells and Threads

Each node runs a multithreaded operating system. Each thread implements one instance of a **shell**. There is one shell thread for each other network address with which a given node is communicating. The combination of **From** and **To** network addresses uniquely identifies the sending and receiving shells. When a node receives a packet for itself (meaning the **To** field matches its own address) it checks the **From** field. If this is the first time this node has heard from the **From** address it starts a new shell for it and sends the data from the packet to it. If it has heard from the other node before then the same shell is used, maintaining context. Thus, by issuing messages consisting of shell commands one node may communicate with and control another.

Each shell can access all local hardware and shares a common pool of **global variables** among all shells on a node. Each shell also has its own set of **local variables**.

4.2. Shell Syntax

Shell syntax consists of ASCII text in a format very much like the C programming language[2]. There are basic built-in functions and variables to directly control the node hardware. It is also fully programmable to allow creating and executing arbitrary algorithms on the fly.

The shell supports functions, variables and expressions as well as conditional and looping statements. All statements must be terminated with a semicolon (;) (like C). Whitespace characters (space, tab, formfeed, newline, return) are ignored except as they serve to separate tokens (like C). The suffix convention for files containing code in this language is **.cmc**, an acronym for "<u>C</u>SI<u>M</u>C <u>C</u>ommand language".

4.2.1. Comments

Comments may be in any of three common styles as follows.

Comment Style	Rule
ANSI C	/* .. */
C++	// to end-of-line
shells	# to end-of-line

4.2.2. Data Types

All variables and function return values are 32 bit signed integers and can hold values -2,147,483,648 .. 2,147,483,647. Since all variables are the same type, declarations are neither necessary nor allowed (unlike C). Scoping rules rely on name spaces defined by naming conventions (unlike C). All names are case-sensitive (like C). Variables and functions persist on a node until it is powered down. Commands which produce motion return immediately; you can create your own synchronous versions by waiting for the **working** variable in your program.

[2]*The C Programming Language*, Second Edition, Brian W Kernighan and Dennis M. Ritchie, Prentice Hall ©1998

4.2.3. Core Functions

Core functions are built into each **shell**. They provide specialized hardware capabilities, provide a means for communicating to the host peer, and other services. Some core functions also use or set certain **core variables** as indicated. The failure of any core function causes the shell to die and any motion it was causing to stop.

Core Functions	
etrack (*t0*, *dt*, *p0*, *p1*, ...)	Track a path given as a list of encoder positions at *dt* ms intervals, where position *p0* occurs at **clock** time *t0*. Path consists of at least 2 up to a maximum of TBD positions. Cubic spline interpolation is used between positions. If *t0* is in the future, CSIMC will drive to *p0*, stop, and begin at *t0*. If *t0* is already past but not too late for some portion of the path, CSIMC will catch the path and track what there is left. If all of the path is late, CSIMC reports "Too late" and no motion occurs. While tracking, the values **maxacc** or **maxvel** are never exceeded, but if they ever would have been the variable **ontrack** is set to 0, otherwise it is set to 1. When the path is completed the motor stops at **maxacc**. Variable **toffset** may be set or changed at any time to apply an offset to the path.
mtrack (*t0*, *dt*, *p0*, *p1*, ...)	Same as **etrack** except positions are in motor steps.
printf (*fmt*, *exp1*, *exp2*, ...)	Send a formatted message to **peer**. The format string, *fmt*, works like C's printf() but only `%d`, `%x`, `%%`, `\"` and `\n` are supported.
stats()	Print statistics to **peer** pertaining to this node's operation. See § 3.2.
stop()	Ramps down motor steps to 0 at **limacc**.

4.2.4. Core Variables

Core variables get or set system state information, most with regards to some hardware facility. Most variables just read back the last value written, indicated by **RW** in the second column. Some have more involved behavior and are described with separate **R** and **W** rows. Some variables are read-only as indicated by just **R**; writing to such variables is silently ignored.

4.2.4.1. Motion Variables

Follows are core variables involved with motion, *i.e.*, step output and encoder input.

Core Motion Control Variables		
eerror	R	1 if an encoder error has been detected since last written, else 0. An error means one of the 4 illegal grey-code transitions has been detected from the possible 8. The complete absence of an encoder is not considered an error.
	W	Writing any value will reset the latched status value to 0.
epos	R	Current encoder count. Initialized to 0 on powerup.
	W	Writing any value to **epos** sets the value to 0.
esign	RW	Set to 1 if motor and encoder steps are of same sign, or -1 if opposite sign.

Core Motion Control Variables (cont.)		
esteps	RW	Encoder counts per revolution. This value is used in conjunction with **msteps** to set the nominal ratio of motor to encoder steps.
etpos	W†	Set the encoder target position, in encoder counts from home.
	R	Returns last value written when read, except when **e/mtrack()** are active then it returns the encoder position for which it is striving.
etrig	RW	Set to the value of **epos** when an edge-triggered input is triggered. See **iedge**.
etvel*	W†	Set the encoder target velocity, in encoder counts/sec*. Acceleration is **maxacc**. Command is ignored if greater than **maxvel**.
	R	Returns last value written when read, except during **e/mtrack()** when it returns the encoder velocity for which it is striving.
evel*	R	Current encoder velocity, in counts/sec*. Based on counts in last 0.50 second.
kv	RW	Velocity profile coefficient, x1000. The default is 2000, which gives slightly under damped behavior. See the discussion for the equation of motion, §10.4.
limacc*	RW	Deceleration used when hit a limit switch (see **ilimit**) or by the **stop()** core function, in motor steps/sec^2*. Must be greater than 0.
maxacc*	RW	Maximum acceleration, in motor steps/sec^{2*}. Must be greater than 0.
maxvel*	RW	Maximum velocity, in motor steps/sec*. Must be greater than 0.
mdir	R	1 if current or last motor motion was in positive direction, -1 if in negative direction.
mpos	R	Current motor step count. Initialized to 0 on powerup.
	W	Writing *any* value to **mpos** sets the value to 0.
msteps	RW	Motor steps per revolution. This value is used in conjunction with **esteps** to set the nominal ratio of motor to encoder steps.
mtpos	W†	Set the motor target position, in motor steps.
	R	Returns last value written when read, except when **e/mtrack()** are active then it returns the motor position for which it is striving.
mtrig	RW	Set to the value of **mpos** when an edge-triggered input is triggered. See **iedge**.
mtvel*	W†	Set the motor target velocity, in motor steps/sec*. Acceleration is **maxacc**. Command is ignored if greater than **maxvel**.
	R	Returns last value written when read, except during **e/mtrack()** when it returns the motor velocity for which it is striving.
mvel*	R	Current motor velocity, in steps/sec*.
ontrack	R	Set to 1 by the **e/mtrack()** functions as long as path is within **maxvel** and **maxacc** limits, else 0 when path is being restrained by either and hence is off track.
timeout	RW	Maximum time, in milliseconds, any one motion-causing command may take. If the time is exceeded, **working** is set to -1and the motor is stopped at **limacc**.
toffset	RW	Motor steps to add to path while running **mtrack()** , or encoder counts while running **etrack**.

Core Motion Control Variables (cont.)		
working	R	Indicates state of last motion-causing command: position or velocity is achieved or path is finished. The state is coded as follows: 1: command is underway 0: command is complete -1: last command took longer than **timeout**.
vascale*	RW	Sets an invisible divisor applied to all variables pertaining to velocity or acceleration *when they are used*. Purpose is to provide for 0.1 or 0.01 precision. May be set to 1, 10 or 100. The default is 1. See further explanation below*.

* All variables pertaining to velocities and accelerations are in units of steps/second or steps/second^2 divided by **vascale**. Legal values for **vascale** are 1, 10 and 100. The default is 1. For example, if **vascale** is 100 and **mtvel** is assigned 123456 this is actually commanding a velocity of 1234.56 steps per second. *Setting vascale does not change the stored numeric value of any variable, it only changes its interpretation when it is used.* Continuing the example, displaying the value of **mtvel** will always show 123456 even if **vascale** is changed in the mean time. It is not sensible to enter a value into one variable with **vascale** set to one value, change **vascale**, then set a different variable because the scales will not be commensurate. To avoid this, it is advised to set **vascale** once very early such as from a boot script (see § 6.3.) then leave it. If it is changed, all affected variables should be reassigned to be sure they are consistent. Although **vascale** can lead to a bit of confusion, the advantage of this is to provide a precision of 0.1 or 0.01 Hz without using precious onboard memory for floating point representations and formatted I/O. Variables pertaining to position are not effected by **vascale**; they are always in native encoder or motor steps.

† Writing to **mtpos**, **mtvel**, **etpos** or **etvel** is the basic method to cause motor steps. *Writing to these values causes motor steps to be generated immediately.* All such motion will adhere to **maxvel** and **maxacc**. Assigning values to these variables also marks the shell from which they are written as using the motor step generation facility of the hardware. Any shell which is still using the step generator when another shell on the same node issues a step command will be killed. In this way, new uses override current uses.

4.2.4.2. I/O Variables

Follows are the core variables involved with the A/D converters and general purpose input/output lines of the CSIMC. The I/O variables are bit-masks. Line *i* is at bit position 1<< *i*, where *i* is a number from 0 through 7 corresponding to the name used in the connector table (see § 2.1.4.).

Core I/O Facilities Control Variables		
ad0..7	R	Actually 8 different variables, one each to read the current value of the 8 Analog-to-Digital converters on JP6. Numeric value ranges from 0..255, corresponding to 0..5V, or about 20mV per step.
iedge	W	Bit mask to reset and arm the edge triggers of a set of input lines. Once triggered, a line remains latched until **iedge** is written again. The transition direction which causes the trigger is defined by **ipolar**. See also **mtrig** and **etrig**. For each bit .. 1 means reset then arm the edge-trigger. 0 has no effect
	R	Returns a bit mask of the state of each edge-triggered input latch. A value of 1 means the edge occurred; what caused it depends on **ipolar**.

Core I/O Facilities Control Variables (cont.)		
ilevel	R	Returns a bit mask of the current level of the 8 misc input lines. This is entirely independent of whether the line is also being used as edge-triggered. The polarity of each line is defined with **ipolar**.
ipolar	RW	Bit mask to set the polarity behavior of each input line. For each bit .. regards to edge-triggering: 1 means **iedge** is set to 1 and latched at the next rising edge (0 $\rightarrow$ 5V) 0 means **iedge** is set to 1 and latched at the next falling edge (5V $\rightarrow$ 0) regards to level-following: 1 means **ilevel** is set to 1 whenever the input is near 5V 0 means **ilevel** is set to 1 whenever the input is near 0V
olevel	RW	Bit mask to control the 8 misc output lines. Depending on JP17-20, for each bit .. 1 means short output or drive to 0v 0 means open output or drive to 5v
homebit	RW	A mask with one bit set to indicate this input line is used as home. Has no special properties but it seemed important enough to warrant a special place to save.
nlimbit	RW	A mask with one bit set to indicate this input line is connected to the limit switch for motion in the negative motor direction. As long as this bit is true in either **ilevel** or **iedge**, motion will not occur in the negative direction. Polarity may be set with **ipolar**, latching may be set with **iedge**. Latching is recommended to avoid missing a small switch at high speed, but beware starting on top of a limit. For convenience when using latched, CSIMC automatically resets a latched negative limit bit when motion in the positive direction is commanded.
plimbit	RW	Same as **nlimbit** but all signs are opposite for the positive-going motor axis.

4.2.4.3. System Variables

Follows are the core variables involved with overall CSIMC system status.

Core System Management Variables		
clock	R	ms since node was booted. 0 .. 2,147,483,647. Rolls over to 0 every 24.9 days.
	W	May be set to any non-negative value to synchronize with host. However, doing so effects the smooth flow of time so writing to **clock** *stops all motion*.
myaddr	R	The board address of this node.
peer	R	Network address with which this node communicates.
prompt	RW	If true, the shell issues a prompt when ready for a new command. The prompt consists of the node address (**myaddr**) followed by a greater-than (>) and a space. This variable is separate for each thread.
version	R	System version number in the form MMVVRR, where MM = model number; VV = software version; RR = software release.

4.2.5. Local and Global Variables

Each shell has access to 36 general purpose variables, divided into two categories.

There are 26 **global variables**, named simply a..z. These are shared among all shells running on a given node and so can be used for inter-thread communication.

There are also 10 **local variables**, named $0..$9. Each shell on a node has its own set of these (even though the names are the same in each).

4.2.6. Remote Variables

Any place a **core** or **global variable** is legal in the shell syntax, a suffix of the form @***n*** may be added to the name to indicate that the value is with respect to node ***n***. Here are some examples.

Set the motor step rate to 3000 on node 2:

```
mtvel@2 = 3000;
```

Set our motor step rate to twice that of the current rate on node 1 plus the value of x on node 3:

```
mtvel = 2*mtvel@1 + x@3;
```

Synchronize our clock to that on node 3:

```
clock = clock@3;
```

4.2.7. User Defined Functions

Up to TBD **user defined functions** may be created using the **define** statement. This has the following form:

```
define function_name (arg1, arg2, ...) {
     statement1;
     statement2;
     ...
}
```

Function names must begin with an alpha character then be followed by of any number of additional alphanumeric characters or underscore (_), although only the first 7 characters are retained as significant. If a user function is defined with the same name as a **core function**, the user function will take precedent.

Functions are global on a node. That is, once defined by any shell, the function may then be used by any other existing or new shell on the same node.

Functions may have up to 10 **arguments**. Argument names must begin with a dollar ($), followed by an alpha character or underscore, then any number of additional alphanumeric characters or underscore, although only the first 7 characters are retained as significant. Requiring a leading $ results in a separate name space allowing for future expansion of core variables.

Functions may use up to 10 temporary **stack variables**, named $0 .. $9. These are separate from the **local variables** of the same name. Stack variables exist only while a function is being executed and vanish when the function returns.

Functions may optionally **return** a value which may then be used in further expressions. If a function does not return a value, then the caller of the function should not use the functions's return value in an expression.

A user function may be deleted by using **undef** as follows:

```
undef function_name;
```

N.B. There is no check to prevent undefining a function that is used by other functions. Also, function references are resolved when the function is *defined*. This means, for example, if function A is used by function B and function A is redefined, function B should be redefined also. Failing to account for these situations will result in undefined behavior.

4.2.8. Expressions

Constants, variables and functions may be used to form expressions which compute and yield values. Operator precedence is as in C; use parentheses to override. All expressions are listed in the following table. Note that assignment is also an expression which yields a value (like C).

Expression Form	**Description**
Constants	Any signed 32 bit value, in decimal, octal (leading 0) or hex (leading 0x).
Core variables	*see separate table.*
Core functions	*see separate table.*
User Global Variables	One of [a-z].
User Functions	One of [a-zA-Z_], then any number of [a-zA-Z0-9_] (but only the first 7 are significant); then arguments in parens.
Function parameters	$, then one of [a-zA-Z_], then any number of [a-zA-Z0-9_] (but only the first 7, including the $, are significant). Available only within function definitions.
Stack variables	$ followed by one of [0-9]. Available only within function definitions.
Unary operators	As in C: ! ~ - + -- ++ unary + is allowed but ignored
Arithmetic operators	As in C: * / % + -
Boolean operators	As in C: < <= > >= == |= numerically 0 if false or 1 if true.
Bitwise operators	As in C: << >> & ^ |
Logical operators	As in C: && ||
Choice operator	As in C: ?:
Assignment	As in C: = *op*= *op* is any Arithmetic or Bitwise operator.

4.2.9. Statements

Expressions may be executed once, repeatedly or conditionally using flow control statements as follows. These are like C with the additions of a new syntax to create and destroy functions, and a shorthand for printing variables.

Statement	**Description**
while (*exp*) *statement*;	while *exp* is not 0 (‘‘true’’) do *statement* .
for (*exp1*; *exp2*; *exp3*) *statement*;	do *exp1*. then while *exp2* is not 0 do *statement* then *exp3.* All expressions are optional (like C)
do *statement;* while (*exp*);	do *statement* and repeat if *exp* is not 0 (‘‘true’’).

Statement	Description(cont.)
if (*exp*) *t-statement;* else *f-statement;*	if *exp* is not 0 do *t-statement* else do *f-statement.* The *else* portion is optional.
{ *statement1*; *statement2*; ... }	any number of statements may be grouped to act as one by surrounding them with curly braces {} (like C)
break;	jumps to the first statement after the smallest enclosing **while**, **for** or **do** statement.
continue;	jumps to the start of the next iteration of the smallest enclosing **while**, **for** or **do** statement.
return (*exp*);	exits the current function, supplying optional *expression* as value. Results are undefined if no expression is given and the caller uses the function's return value anyway (like C).
;	a null statement.
define *func* (*$arg0*, *$arg1*, ...) { *statement1*; *statement2*; ... }	defines a new user *function*. May have up to 10 *arguments*. The body of zero or more *statements* may use up to 10 temporary variables named $0..$9. If *func* is already defined it will be silently redefined.
undef *func*;	forget the definition of user *function*. Take care not to undefine a function used by other functions!
= *exp;*	an *expression* which is assigned to nothing is a shorthand equivalent to **printf ("%d\n", *exp*)**.

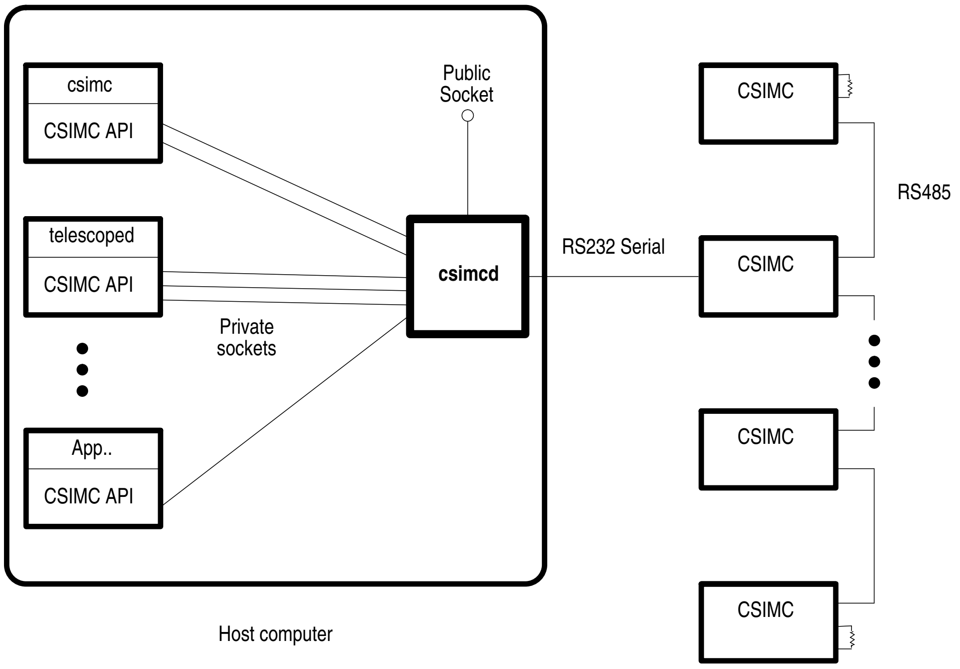
4.2.9.1. *If* caveat

The **if** statement is a bit odd when used interactively because the compiler can not look ahead and determine whether an optional **else** will follow. If no **else** statement is desired, use the **null statement** (see §4.2.9.). For example:

```
04> if (i<20) j = 30;;
04>
```

5. Csimcd: Control Daemon

Csimcd is a daemon process running on the host computer. It is required to operate the CSIMC nodes in a network. It serves as the gateway between the single physical RS232 link on the host computer to the CSIMC network and the multiple client processes on the host which wish to use the CSIMC nodes. The API (see §9.) is used by each application to contact *csimcd* one time for each CSIMC to which it wishes to communicate. The following diagram summarizes the overall arrangement of the CSIMC network.



To serve host processes, *csimcd* advertises connection service to other interested processes on a well-known TCP/IP port. By default this is 7623. It is this socket to which all applications that use the CSIMC API (see §9.), including the *csimc* interactive command tool (see §6.), first connect when wishing to contact a node. *Csimcd* accepts the request and tests the indicated address by sending a Ping packet. If it gets an ACK, it binds a unique address to a new socket for this connection, records it in a table so it can associate a socket address with a node connection, and returns the new socket to the calling client process. Subsequent traffic on this socket is then passed through to the connected node. If either side fails or closes its connection, the peer side is closed as well.

In order to implement a network wide panic mode, *csimcd* listens for a **Control-C** character to arrive from any client socket. If it does, it broadcasts a Reboot command packet to all nodes on the network.

Another duty for *csimcd* is to listen for logging packets on host virtual address 63. When any such packets arrive, they are saved in a log file.

5.1. Usage

The *csimcd* program supports the following command line options:

csimcd [-c *config_file*] [-i *port*] [-m] [-t *tty*] [-v]

-c: specifies an alternate configuration file (see § 6.3.). The default is **$TELHOME/archive/config/csimc.cfg**.

-i: specifies an alternate TCP/IP port on which to listen for client connections, including the *csimc* interactive tool (see § 6.). This option overrides the value of PORT in the configuration file, if any. The default with no configuration file is 7623.

-m: allow multiple instances of *csimc* to be running at the same time. This is required to support separate CSIMC LANs connected to the same computer at one time, and each is using a separate ***port*** and ***tty***.

-t: specifies the RS232 line to use for connection to the CSIMC network. This option overrides the value in the configuration file, if any. The default with no configuration file is **/dev/ttyS0**.

-v: each causes increasing amount of verbose detail to be logged, up to a maximum of 4.

5.2. Configuration file

Both *csimcd* and *csimc* share a common configuration file, **$TELHOME/archive/config/csimc.cfg**. Complete details may be found in §6.3.

5.3. Log file

Csimcd logs all unusual events to a log file located in **$TELHOME/archive/logs/csimcd.log.** Routine operation can also be monitored by using various levels of the **-v** option. The verbosity level can be increased after the program has already begun by sending it the SIGHUP signal. Each signal will increase the level by one until a maximum is reached when it will roll back to no verbosity. To send this signal, first find the process id of the *csimcd* program by inspecting the output of "`ps axf`" then use the value in the PID column as "`kill -HUP` *`pid`*". Or, if you are running OCAAS, you can also use its *killp* tool to send a signal to a process by name, as in "`killp -HUP csimcd`".

6. *Csimc*: Interactive Sessions

The program *csimc* provides a command line interface for up to one connection with one or more CSIMC nodes. To make more than one connection to the same node, run multiple instances of *csimc*, such as from separate *xterms*. Any number of instances of *csimc* may be running at the same time.

Csimc requires *csimcd* (see §5.) and starts it automatically if it is not already running. If *csimc* can not establish contact with *csimcd* it exits with value 1.

6.1. Usage

The *csimc* program has the following command line options:

csimc [-c *config_file*] [-i *host port*] [-l] [-n *addr*] [-r] [-t *addr baud*] [-v]

-c: specifies an alternate configuration file (see § 6.3.). The default is **$TELHOME/archive/config/csimc.cfg**.

-i: specifies an alternate TCP/IP *host* and *port* at which to contact a *csimcd*. The defaults come from the **HOST** and **PORT** entries of the **csimc.cfg** config file. If there is no config file, the default host is *localhost* at 127.0.0.1 and the default port is 7623.

-l: causes the *.cmc* scripts listed in the configuration file (see §6.3.) for each node to be loaded. If any node or script fails to load, *csimc* exits with value 3. Scripts may also be loaded later from the command prompt with the **load** command.

-n: requests an initial connection be created to the node with the given address, as if by issuing the "**!*addr***" command from the prompt (see § 6.2.). If the connection can not be made, *csimc* exits with value 2.

-r: reboot all nodes on the network.

-t: requests an initial connection be created to the serial port on the node with the given address at the given baud rate, as if by issuing the "**!serial *n b***" command from the prompt (see § 6.2.). If the connection can not be made, *csimc* exits with value 2.

-v: sets more verbose output to *stderr*.

If any other command line arguments are given, a usage summary is printed to *stderr* and the program exits with value 4. *Csimc* exits with value 0 when it receives EOF from its *stdin*. All connections are automatically closed whenever *csimc* exits for any reason.

6.2. Commands

Once *csimc* is running it puts the user in direct communication with any desired node. Almost everything from *stdin* is sent to the current node connection, and anything any connected node transmits is sent to *stdout*. (These are the users keyboard and screen unless redirected.)

But the program has a few built-in commands of its own. To use these type a bang (!) as the first thing on a line followed by the command. This allows you to temporarily talk directly to the *csimc* program and not to a node's shell. Follows is a description of these commands. Any portion of the command name may be entered but the first character is sufficient. While connected to a node, the prompt will consist of the node address followed by greater-than (>). Before connecting to any node the prompt will be xx>.

6.2.1. Command line editor

If *csimc* finds *stdin* is connected to a tty (that is, *isatty(3)* returns true) it implements a simple command line editor with history reminiscent of that provided by *tcsh*, *gnuplot*, *gdb* and other interactive command-line UNIX programs. Trying to move somewhere inappropriate causes a beep. When the line looks as desired, press Enter.

If *stdin* is not a tty, the line editor is not available and all input is simply processed sequentially immediately.

Key	Editing Action
Backspace or Delete	erase character to left of cursor
Left arrow	move left 1 space nondestructively
Right arrow	move right 1 space nondestructively
Up arrow	move to next older history entry
Down arrow	move to next newer history entry
Control-A	move to beginning of line
Control-E	move to end of line
Control-K	delete to end of line
Control-U	delete entire line

6.2.2. Downloading new firmware

New firmware for a node may be loaded using the following command:

```
! Firm node-address filename.cmf
```

Replace the firmware stored in FLASH on the given node with the new code from the given file. Any existing connections to the node will first be closed. Progress and data rate are displayed as the download takes place. Once begun this can not be interrupted. The **F** is capitalized to make it harder to invoke by accident but beware *there is no "are you sure?" message*. **Do not do this unless you know what you are doing... this command can render the CSIMC so useless it must be returned to the factory for refurbishment,**

6.2.3. Connection control

A connection is made to a given node by typing ! followed by the node address, as follows:

```
! node-address
```

Create a new connection to the node with the given address and start on it a new shell. Command mode is automatically ended and subsequent input goes to the shell on the given node. When a new node connection is created other existing connections are not broken. If the given node has already been connected to before, the same shell is reconnected; a new shell is *not* created.

A connection may be broken as follows:

```
! close
```

This will break the connection to the current node. The shell is terminated. If the same node address is given later a new connection and shell are created. It is not necessary to close connections individually by hand this way before exiting because all connections created by *csimc* are automatically closed when *csimc* exits for any reason.

6.2.4. Serial port connection

Each CSIMC has 1 RS232 serial port. When first powered up this port can become the interface to the host for the CSIMC network. Then the serial ports on other nodes in the network are free to be used for any general serial communication. *Csimc* provides one easy way to do this as follows:

```
! serial [n [b]]
```

Connect the stdin and stdout of *csimc* to the serial port on the node with address ***n*** at baud rate ***b***. If there is a current connection the node address is optional in which case the current node will be used. The baud rate is always optional and may only be specified if the node address is given explicitly; if not specified the default baud rate is 1200. The other RS232 parameters are fixed as N/8/1 (no parity, 8bits, 1 stop bit). If successful, a status message such as the following will be printed:

```
Connecting to Node 1 serial port @ 1200 ..
 .. ready. Quit with Ctrl-D.
```

Now everything typed will be sent to the RS232 port on node 1, and anything received on the port will be printed. This remains in effect indefinitely until a Control-D is typed (or EOF occurs if being scripted) at which time the connection is closed and *csimc* issues its own prompt again.

6.2.5. History

```
! history [n]
```

Typing this command without an argument will show the previous command lines typed to this instance of *csimc*, oldest on top. Each command will be preceded by a number. Typing this command with that number will reissue the command.

6.2.6. Load script file

```
! load filename.cmc
```

Loads the given .*cmc* script file to the currently connected node. The file name is with respect to the current directory unless the name begins with /.

6.2.7. Session tracing

```
! trace [filename]
```

Starts and stops tracing. If the file name is included, tracing is started and all input and commands to *csimc* and all node traffic will be appended to the given file. If the file name is not included, tracing is turned off. Error messages and other chit-chat from *csimc* itself are not logged.

6.2.8. Help

```
!
```

A line ! or anything unrecognized while in command mode prints a summary of all commands available. Also printed is a list of the current connections.

6.2.9. Interruptions

```
! interrupt
```

Tell the shell on the current connection to stop any looping and return to reading more input. Typing **control-C** has the same effect.

```
! Reboot
```

Tell <u>all</u> nodes to reboot. No current connection is necessary. All connections to all nodes, if any, will be closed. The **R** is capitalized to make it harder to invoke by accident but beware *there is no "are you sure?" message*.

6.2.10. Exiting

To exit the *csimc* program type **control-D** (end-of-file from scripts). All threads created from this instance of *csimc* will be terminated.

6.3. Configuration File

Both *csimc* and *csimcd* share a single configuration file, by default **$TELHOME/archive/config/csimc.cfg**. Both programs allow an alternate file to be specified using their **-c** options. Not all entries in the file are used by both programs.

The format of the config file is a set of *name*=*value* pairs. If *value* must include whitespace it must be surrounded with quotes (') or double-quotes ("). Blank lines and everything following a pound sign (#) to the next line are ignored for use as comments.

We explain by example. Consider the following file:

```
TTY = "/dev/ttyS3"              # network RS232 connection
HOST = "luna.u.edu"             # Internet host name or IP of csimcd
PORT = "4321"                   # Internet port number on host of csimcd

INIT1 = "basic.cmc radec.cmc"   # scripts to preload on node 1 (if -l)
INIT2 = "basic.cmc radec.cmc"   # scripts to preload on node 2 (if -l)
INIT3 = "basic.cmc focus.cmc"   # scripts to preload on node 3 (if -l)
INIT4 = "basic.cmc dome.cmc"    # scripts to preload on node 4 (if -l)

SER3 = "/dev/ttypf 9600"        # assign /dev/ttypf@9600BPS to node 3
```

The first line specifies on which RS232 line the CSIMC network is connected. This may be overridden by the **-b** option of *csicmd*. This entry is only used by *csimcd*.

The next two lines specify the TCP/IP host and port at which to contact the *csimcd* gateway daemon. The *csimc* client program uses both entries. The *csimcd* daemon always runs on the *localhost* but honors the Port entry. Both entries are optional and both programs have command line switches to averride as appropriate.

Next are entries with names of the form **INIT*n****.* Each lists script files which will be preloaded onto node ***n*** by *csimc* if it is executed with its **-l** option. ***n*** must be 0..31. As many files as desired may be listed, separated by a space. If more than one are listed the collection must be surrounded by double quotes ("). In the example here, the four lines specify that nodes on the network with addresses set to 1 through 4 will be loaded. All nodes are loaded with functions from the file *basic.cmc*. In addition, nodes 1 and 2 are loaded with functions in the file *radec.cmc* and nodes 3 and 4 are loaded with functions in *focus.cmc* and *dome.cmc*, respectively. All script file names which do not begin with / are assumed to be located first in the current directory then in **$TELHOME/archive/config**.

Names of the form **SER*n*** indicate which UNIX pseudo-tty to assign as a surrogate for the serial port on board ***n*** and at what baud rate, if any. Standard rates up to 9600 are supported. The pty name and baud rate must be separated by a space.Take care to use ptys high enough not likely to interfere with other processes that use them. The normal convention is to scan and use the next available ptty at runtime but in this case we prefer to assign a known value so it may be used by processes otherwise unaware they are using a ptty. If successful, the named terminal may then be used with any serial device, such as GPS receiver or weather station. *Do not assign the serial port of a node acting as the host interface*.

6.4. Sample Session

Follows is a sample session with *csimc*. We assume the program was started with no command line option. The user types what is shown in **bold**. Indenting is shown for clarity and is encouraged but is not required. Commentary is shown delimited by "//" but need not be typed in of course. The node address in the prompt and all responses are hypothetical.

```
xx> !2                               // connect to node 2
Making new connection with node 2
02> maxvel = 100000;                 // set max velocity to 100,000 steps/sec
02> maxacc = 100000;                 // set max acceleration to 100,000 stp/s^2
02> limacc = maxacc*10;              // set more aggressive limit acceleration
02> plimbit = 0x10;                  // set pos lim bit for safety
02> nlimbit = 0x04;                  // set neg lim bit for safety
02> =mpos;                           // query current motor step count
123456
02> mtpos = mpos+500000;             // initiate relative move of 500,000 steps
02> =mpos;                           // get prompt immediately.. read cur pos
123500
02> =mpos;                           // query again a little while later
335645
02> while (working);                 // tired of checking by hand.. just wait
02> =mpos;                           // prompted when done.. confirm position
623456
02> define wait() {while(working);}  // tired of waiting by hand. define a func
02> define mMove($nsteps) {          // write function to move nsteps and wait
02.   mtpos = mpos + $nsteps;        //   target is current plus desired move
02.   wait();                        //   wait until there
02. }
02> mMove (-500000);                 // use it to move back 500,000 steps again
02> =mpos;                           // prompted when complete.. check it.
123456
02> !2                               // connect to node 2
Already connected to node 2          // Doh!
02> !3                               // connect to node 3 (assume used befor)
Resuming on node 3 with TId 23
03> =mpos;                           // report motor position.. this is not 2!
-3454
03> !2                               // back to node 2
Resuming on node 2 with TId 24
02> =mpos;                           // show 2's motor pos
123456                               // yup, we're back on 2
02> !l find.cmc                      // load a script containing findhome()
find.cmc: load complete              // confirms script was found and loaded
02> stats();                         // check for ourself
    Node: 10303 version 111 FLEX  00010=2 myaddr    Gateway
    Time:    33 motlat   0% COP    0:05 dT     0:05 up
  Memory: 7884 + 1606 + 1592 + 198 + 98 + 34 + 30 = 11442
     abs:    35 bytes     1 args      0 tmpv
     max:    31 bytes     2 args      0 tmpv
     min:    31 bytes     2 args      0 tmpv
    sign:    41 bytes     1 args      0 tmpv
   pause:    56 bytes     1 args      1 tmpv
 findhom:   482 bytes     1 args      2 tmpv
 findlim:   254 bytes     1 args      2 tmpv
 testhom:   210 bytes     0 args      1 tmpv
 testlim:   154 bytes     0 args      1 tmpv
 testtra:   190 bytes     0 args      2 tmpv
  limits:    75 bytes     0 args      0 tmpv
  prtest:   215 bytes     1 args      2 tmpv
   watch:   310 bytes     1 args      7 tmpv
 Tx Pkts: 40268 total 40269 1stry     0 retry     0 fail
 Rx Pkts: 65535 total 65535 good  37207 mine    182 dups      0 rogue
 Peer 34:     6 TId     12% CTStk   15% RTStk   69% CStk   0:00 up
 Peer 33:     7 TId      2% CTStk   15% RTStk   28% CStk   0:19 up
 Peer 32:    24 TId     20% CTStk   23% RTStk    8% CStk   0:05 up
02> mtvel=5000;                      // start a move.. wait a while.. then kill
^C
02>
```

7. *Csimcio*: Script Interface

The *csimcio* program is useful communicating with CSIMC nodes from a script. It uses UNIX *stdin* and *stdout* for all data flow.

7.1. Usage

The *csimcio* program has the following command line options:

csimcio [-f *filename*] [-h *host*] [-m *nlines*] [-n *addr*] [-p *port*] [-r] [-v] [-w *secs*]

-f: load the given *.cmc* file then exit; default is to read from *stdin* until EOF.

-h: connect to *csimcd* at the given TCP/IP host address; the default is localhost or 127.0.0.1

-m: wait for and print the given number of lines of response, then exit.

-n: connect to the CSIMC node wiht the given addre; the default is 0.

-p: conect to *csimcd* on the given TCP/IP port address; the default is 7623.

-r: wait for and print one line of response, then exit.

-v: verbose output.

-w: wait for the given number of seconds of quiet (no responses), then exit.

8. Programming Examples

Follows are some hypothetical examples of code sent to CSIMC nodes.

Write a handy function to wait for an action to complete.

```
define sync() {while(working) continue;}
```

Move positive until see a switch closure to +5V on input line 3 then move back to, stop and report that position:

```
$0 = 1<<3;                           // mask for bit
ipolar = $0;                         // use 0->5V edge direction
iedge = $0;                          // arm edge-trigger
mtvel = maxvel;                      // move in positive direction full speed
while (!(iedge & $0));               // until see bit.. edge sets mtrig
mtpos = mtrig;                       // move back to trigger position
sync();                              // wait for done
=mpos;                               // report position
```

Rock back and forth 30 times by 100,000 steps either way from current position:

```
$0 = 100000;                         // how far to rock
$1 = mpos;                           // save start position
for ($2 = 0; $2 < 30; $2++) {        // loop 30 times
     mtpos = $1+(($2&1) ? $0 : -$0); // goal alternates +/- from start
     sync();                         // wait for done
}
```

Test the mtrack and etrack functions.. should follow the same path if no slippage.

```
define testtrack() {
    $0 = 30000;                      // max motor move from starting pos
    $1 = $0*esteps/msteps;           // same, in encoder counts
    while (1) {
        mtrack (clock+10000, 10000, $0/10, -$0/5, $0/2, -$0);
        sync();
        etrack (clock+10000, 10000, $1/10, -$1/5, $1/2, -$1);
        sync();
    }
}
```

Handy function to pause for $ms milliseconds

```
define pause ($ms) { for ($0=clock+$ms; clock<$0; ); }
```

Watch for and report any motion or input changes, pausing $ms ms between checks. Assume `homebit`, `nlimbit` and `plimbit` are set to the desired bit for home, negative and positive limits, respectively. Report levels as lower case, edges as upper case.

```
define watch($ms) {
    while (1)  {
        printf ("%d ", clock);
        if (ilevel&homebit) printf ("h"); else printf ("-");
        if (iedge&homebit)  printf ("H"); else printf ("-");
        if (ilevel&nlimbit) printf ("n"); else printf ("-");
        if (iedge&nlimbit)  printf ("N"); else printf ("-");
        if (ilevel&plimbit) printf ("p"); else printf ("-");
        if (iedge&plimbit)  printf ("P"); else printf ("-");
        printf (" 0x%x 0x%x %d %d %d %d %d %d\n", $7=ilevel, $6=iedge,
                $4=working, $5=ontrack, $1=epos, $3=evel, $0=mpos, $2=mvel);
        while ($0 == mpos && $1 == epos && $2 == mvel && $3 == evel &&
                $4 == working && $5 == ontrack && $6 == iedge && $7 == ilevel)
            continue;
        pause($ms);
    }
}
```

Generate a 10 Hz square wave (period 100ms) on output bit 4:

```
while (1) {                            // forever
     obits |= (1<<4);                  // set bit 4
     pause (50);                       // wait first half-period
     obits &= ~(1<<4);                 // clear bit 4
     pause (50);                       // wait second half-period
}
```

Define a function, **emovew (n)**, which will initiate and wait for the encoder to move *n* steps from its current position. Then invoke the function to move -10,000 encoder steps:

```
define emovew ($n) {
     etpos = epos + $n;
     sync();
}

// move back 10,000 encoder steps and wait for done
emovew (-10000);
```

Write a function that finds home. The final approach direction is defined by $way, which is +1 if home is to be used while moving positive, or -1 if moving negative. .Assume `ipolar`, `homebit`, `nlimbit` and `plimbit` are set to the desired polarity and bits for home, negative and positive limits, respectively. First some helper functions.

```
/* go $way until the given bit triggers with opposite than its usual polarity
 */
define goback ($way, $bit)
{
     ipolar ^= $bit;                   // looking for opposite side
     iedge = $bit;                     // arm
     mtvel = $way*maxvel;              // go
     while (!(iedge & $bit));          // wait until pass
     ipolar ^= $bit;                   // back to triggering originally
     mtpos = mpos;                     // stop here
     sync();                           // syncronous
}
```

```
/* move away from either limit */
define nolim() {
    mtpos = mpos;                    // sanity stop where we are
    sync();                          // wait for stop
    if ((ilevel|iedge) & plimbit)    // if at or past pos lim
        goback (-1, plimbit);        //   go neg
    if ((ilevel|iedge) & nlimbit)    // if at or past neg lim
        goback (1, nlimbit);         //   go pos
}


/* find home in pos/neg direction, according to $way = +/-1.
 * N.B. requires ipolar, homebit, plimbit and nlimbit to be set up.
 */
define findhome($way) {
    $0 = homebit|plimbit|nlimbit;    // handy mask of all bits
    if (!$0) {                       // required!
        printf ("-1: no bits\n");
        return;
    }

    /* allow for case of home and a limit together. */
    nolim();

    /* find first approx to home, allowing for bouncing off limit.
     * "right" and "left" are WRT $way==+1
     */
    iedge = $0;                      // arm edge-triggered
    $way = $way < 0 ? -1 : 1;        // insure purely +1 or -1
    mtvel = $way*maxvel;             // start for home, going right
    $1 = $way>0 ? plimbit : nlimbit; // ignore if pass left-going limit
    while (!(iedge & homebit)) {     // wait to see home (sets e/mtrig)
        if (iedge & $1) {            //   but if hit right limit
            printf("3: hit limit\n");//     report bounce
            goback(-$way, homebit);  //     go back
        }
    }

    /* found approx home going right fast, now repeat more slowly */
    printf ("2: rough guess\n");     // starting final run
    mtpos = mtrig-$way*4*2000;       // get in front by 4 secs @ 2000
    sync();                          // wait until there
    iedge = homebit;                 // arm home
    printf ("1: fine tuning\n");     // final run
    mtvel = $way*2000;               // go forward, slowly
    while (!(iedge & homebit));      // wait to see home (sets e/mtrig)
    mtpos = mtrig;                   // go back to set
    sync();                          // wait until stopped
    epos = mpos = 0;                 // set as new home
    printf ("0: done\n");            // report done
}
```

FInd the encoder position of either the positive or negative limit switch. The desired switch is selected by calling with +1 for the positive limit or -1 for the negative limit. Assume `homebit`, `nlimbit` and `plimbit` are set to the desired bit for home, negative and positive limits, respectively.

```
/* find and report encoder position of either limit, according to $way = +1/-1.
 * N.B. requires ipolar, plimbit and nlimbit to be set up.
 */
define findlim($way) {
    $0 = plimbit | nlimbit;           // handy mask of both bits
    if (!$0) {                        // required!
        printf ("-1: no bits\n");
        return;
    }

    /* need a running start to find edge */
    nolim();

    /* go */
    iedge = $0;                       // insure latches are cleared
    $way = $way < 0 ? -1 : 1;         // we count on exactly +/-1
    mtvel = $way*maxvel/2;            // go at modest speed in $way dir
    while (!(iedge & $0));            // wait for encounter. sets etrig
    etpos = etrig - $way*esteps/72;   // back off a little
    sync();                           // wait until stopped
    printf ("0: %d\n", etrig);        // report
}
```

Write a function that commands and waits for a desired dome encoder count and tolerance. First we define some functions to capture the underlying hardware connections. Assume bit 4 HI makes the motor rotate the dome CCW, bit 3 HI rotates CW.

```
/* turn off dome motor and wait for stop */
define domeStop() {
    obits &= ~((1<<3)|(1<<4));        // turn off both motor direction bits
     do {                             // repeat:
         $0 = epos;                   //   save encoder value now
         pause (500);                 //   wait 500 ms
     } while ($0 != epos);            // until no change in position
}


/* initiate rotate CW */
define domeCW () {
     domeStop();                      // first stop
     obits |= (1<<3);                 // then turn on CW motor bit
}


/* initiate rotate CCW */
define domeCCW () {
     domeStop();                      // first stop
     obits |= (1<<4);                 // then turn on CCW motor bit
}


/* return absolute value */
define abs($x)   { return ($x < 0 ? -$x : $x); }
```

```
/* move dome so encoder is at $target +- $tol.
 * return 0 if succeed, else -1.
 * we assume the encoder increases rotating CW
 */
define setDome ($target, $tol) {
     $0 = epos;                      // init previous encoder pos
     $1 = epos - $target;            // init previous error
     $4 = 0;                         // count loops stopped to check for stuck

     // loop until in tolerance or detect stuck
     while (1) {
         pause (200);                // domes can be sluggish
         $2 = epos - $target;        // error now
         $3 = epos - $0;             // movement since last time
         if (abs($2) <= $tol) {      // if in tolerance
             if ($3 == 0)            //   if stopped
                 return (0);         //     done
             domeStop();             //   stop
         } else {                    // else not in tolerance
             if ($3 == 0) {          //   if stopped
                 if (++$4 > 5)       //     if stays stopped for several loops
                     break;          //       stuck
                 if ($2 < esteps/2)  //     if CW of target
                     domeCCW();      //       go CCW
                 else                //     else
                     domeCW();       //       go CW
             } else {                //   else moving
                 $4 = 0;             //     reset stuck loop counter
                 if (abs($2)>abs($1))//     if going the wrong way
                     domeStop();     //       stop
             }
         }

         $0 = epos;                  // update position
         $1 = epos - $target;        // update error
     }

     // if get here, dome is stuck
     printf ("Dome is stuck\n");
     return (-1);
}
```

Now invoke the function to move to 123,456 ±2,000.

```
// Command dome to 123,456, ±2,000. Wait until in place and stopped.
setDome (123456, 2000);
printf ("Ok\n");
```

9. Host API

This section describes the CSIMC functions available to C applications running on the host computer. These work in conjunction with the *csimcd* node controller daemon process (see § 5.). The basic idea to communicate with a given node is to open a connection to it and talk to the new **shell** using the language defined in § 4.2. The API also includes functions to issue special thread control commands.

The connection mechanism is a convenient UNIX file descriptor and may be managed using the standard system calls as desired, such as *read*, *write*, *select*, *fcntl* and so forth. Handy formatted write and write/read functions are also provided. Direct use of *fprintf* and *fscanf* is also possible by using *fdopen* to convert the file descriptor from **csi_open()** to a FILE pointer and disabling buffering by using *setbuf*. Closing any CSI file descriptor with the UNIX *close* function will work as far as the associated node is concerned but does not update state information cached in the process's own memory. Closing as a side effect of the process existing is ok because this cache gets discarded anyway.

Each new file descriptor is automatically assigned a virtual host address unique system-wide in the range 32..63. Therefore, up to 32 simultaneous connections to one or several nodes may be active at one time. If the node's shell exits for any reason, the file descriptor will become closed in the UNIX process. If the UNIX process exits, all its file descriptors are closed in the usual way which causes all shells to which the process was connected to be killed.

All functions return -1 if they fail and append error messages to the log file for **csimcd** (see 5.3.). These functions and supporting declarations are found in **csimc.h**.

Host API Function	Description
int csi_close (int *fd*)	Terminate the CSIMC network connection *fd*. If connected to a shell it is killed. Do not just use the UNIX *close* system call on *fd*. Exiting the UNIX process works fine however.
int csi_open (**char *_host_,** **int _port,_** **int _addr_)**	Connect to the **csimcd** network gateway running on the given TCP/IP *host* listening to the given *port* and create a connection to a new shell on the given *addr*. Return a file descriptor for communication with the new shell using *read*, *write* etc. If *host* is NULL the local host is assumed; if *port* is 0 the default 7623 is assumed.
int csi_sopen (**char *_host_,** **int _port_,** **int _addr_,** **int _baud_)**	Connect to the **csimcd** network gateway running on the given TCP/IP *host* listening to the given *port* and create a connection to the RS232 serial port on the given *addr* at the given *baud* rate. Standard rates up to 38400 baud are supported. Return a file descriptor to a pseudo-tty surrogate. Use **csi_close** when finished. See **csi_open** for default *host* and *port*.
int csi_f2h (int *fd*)	Given a file descriptor from any **csi_open** variant return the host address.
int csi_f2n (int *fd*)	Given a file descriptor from any **csi_open** variant return the node address.
int csi_intr (int *fd*)	Interrupt the shell to which *fd* refers so it returns to waiting for input.
int csi_rebootAll (**char *_host_,** **int _port_)**	Connect to the **csimcd** network gateway running on the given TCP/IP *host* listening to the given *port* and issue an immediate reboot <u>to all nodes</u>. On successful return call **csi_close**() with the *fd* obtained from this call and each prior **csi_open** to complete the process; then start over making new connections if desired.

Host API Function	**Description(cont.)**
int csi_r (int *fd*, char *buf*[], int *buflen*)	Handy function to wait for and read up through the next newline or *buflen*-1 chars, whichever comes first, into *buf*[] from *fd*. '\0' is added to the end. Returns length, 0 if EOF, -1 if error. *Note well: there is no buffering behind this call.. so if you fall behind all CSIMC network communication will block until you catch up.*
int csi_rix (int *fd,* char **fmt*, ...)	Handy function that sends an expression to the shell at *fd* that results in an integer value then calls strtol(buf, NULL, 0) with the result. For example, to get the current clock in seconds: `c = csi_rix (fd, "=clock/1000;");` Note there is no error return value.
int csi_w (int *fd*, char **fmt*, ...)	Handy function to write one formatted command to *fd*. The *fmt* and remaining optional arguments work just like C's *printf()*. Returns the length of the final message if ok, else returns -1.
int csi_wr (int *fd*, char *buf*[], int *buflen*, char **fmt*, ...)	Handy function that is equivalent to `csi_w(fd, fmt, ...)` followed by `csi_r(fd, buf, buflen)` all in one.

10. Implementation Notes

The information in this appendix is generally not required for typical installation and operation of the CSIMC. It is included for those interested in additional details and for completeness.

10.1. Datalink Protocol

Communication among CSIMC nodes and the host consists of data packet and a token-ring control packet. RS485 is used between nodes operating in 2-wire multidrop mode. The host uses RS232 to any one CSIMC node which also serves as a gateway to the 485 LAN. Follows is a description of the datalink protocol used by the CSIMC network.

10.1.1. Data Packet Format

One data packet is depicted in the following diagram. Each box represents one byte.

1	2	3	4	5	6	7	8...
Sync	**To**	**From**	**Seq** / **Type**	**Count**	**Header Sum**	**Data Sum***	**Data*** ...

***** not sent if **Count** is zero

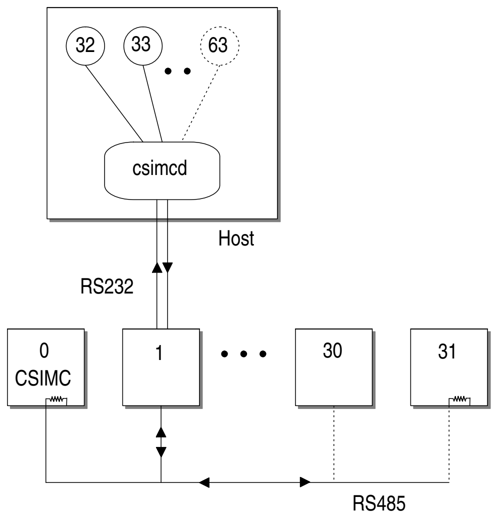
Key:

Byte	Field	Data Packet Byte Description
1	**Sync**	$88_{16} = 136_{10}$. *This value must not appear at any other place in the protocol.*
2	**To**	Network address to which packet is being sent. Addrs 0..31 refer to CSIMC nodes; 32..63 are virtual host node addresses; 63 is the error logging address; 254 means broadcast to all nodes except the one indicated in **From**. Broadcast packets are not ACKed. All other addresses reserved (see §10.1.4.).
3	**From**	Network address from which packet is being sent. Addrs 0..31 refer to CSIMC nodes; 32..63 refer to virtual host node addresses. All others reserved.
4	**Seq**	Upper 4 bits of byte 4 form a sequence number that is incremented by sender before each new packet is sent. Used to eliminate dups if ACK packet itself gets broken.
	Type	Lower 4 bits of byte 4 holds a code to indicate the purpose of the packet: 0: **SHELL: Data** is a shell message to **To** from **From**. 1: **BOOTREC: Data** is boot record. See format in §10.2.1. 2: **INTR:** Interrupt shell on **To** whose peer is **From**. **Count** is 0. 3: **KILL:** Kill thread on **To** whose peer is **From**. **Count** is 0. 4: **ACK:** Acknowledge. **Count** is 0 unless from GetVar. **Seq** matches original 5: **REBOOT:** Reboot node **To**. **Count** is 0. 6: **PING:**Ping: Check existence of **To**. Response is just ACK packet. **Count** is 0. 7: **SETVAR**: Set variable: **Data** contains reference followed by value. 8: Reserved permanently to insure **Type** field never equals **Sync**. 9: **GETVAR**: Get variable: **Data** contains reference. **ACK** has value in **Data**. 10: **SERDATA:** Serial Data: **Data** is to **To** or from **From** the RS232 port. 11: **SERSETUP**: Serial Setup: **Data** is baud rate. Must precede any **SERDATA**. 12..15: Reserved

Byte	Field	Data Packet Byte Description (cont.)
5	**Count**	Total number of bytes in **Data** portion of packet, 0..63.
6	**Header Sum**	Check sum of first 6 bytes in packet. Addition is performed with a 16 bit accumulator. Any bits then in the high order byte are shifted right 8 and added back in; repeated until the high byte is 0. If sum ends up **Sync**, it is set to 1.
7	**Data Sum**	Check sum of **Data** portion. Same algorithm as **Header Sum**. Not included if **Count** is zero.
8 ...	**Data**	**Count** bytes of data. Any byte value equal to **Sync** is sent as the two-byte sequence ED 00; ED is sent as ED EC. **Count** includes all such expansions.

10.1.2. Topology

Below is a diagram showing how the major components of the CSIMC network are connected. The Host computer is connected to one CSIMC through an RS232 connection. Up to 31 additional CSIMC boards may be connected using a 2-wire RS485 LAN. The two nodes on the physical end of the network are jumpered to provide a termination for the transmission line. Any node at any address may serve as the bridge.



10.1.3. Operation

Running on the Host is a gateway daemon process, *csimcd.* This provides a common socket contact point and implements the protocol by which applications connect to any desired boards. Each socket is assigned a network address in the range 32 through 63.

The network uses a token-passing scheme. The host serves as the token broker and monitor. No CSIMC board may transmit until it receives the Token. When a node gets the Token it may send at most one packet to any other address. If it has nothing to send it immediately returns the Token to the host. If the node wants to send a packet it does so and waits up to 1 second for an ACK. When it receives an ACK, or times out, it sends the Token back to the Host. A node may retry up to ten more times. The token must be returned after each retry. If the host does not receive the Token back after 5 seconds it logs a token-timeout. Whether returned or timed out, the host then sends the Token to the next node in turn. After the last network node address is issued, the Token is considered having been passed to the host. The host then sends at most one packet from each client connection using the same ACK scheme described above. This process repeats forever.

The Token packet is not directly acknowledged. Trouble is indicated by the lack of the Token being returned on time.

10.1.4. Token Packet Format

For efficiency, the Token itself is a special abbreviated form of Packet with only 2 bytes. It consists of the **Sync** value followed by the address of the node getting the Token plus 64=0x40. This bias distinguishes a Token packet from a data packet. Thus, since node addresses range from 0 through 31=0x1F, Token values to nodes are in the range 64=0x40 through 95=0x5F. When a node returns the Token to the host, it uses the value 96=0x60.

10.2. Power-On

When the CSIMC is first powered on, it checks whether a shunt is inserted in JP5 connecting pins 1 and 2. If installed, a self-test is executed and runs until power is cycled. Refer to §3.1. for full details on running the self-test.

If there is no shunt installed at power-up CSIMC boots from on-board FLASH and starts looking for characters to arrive on either the RS232 or RS485 ports, presumably packets. The ports are logically tied together as a passive gateway, copying all characters received each way to the other port.

10.2.1. LAN Firmware updates

The CSIMC FLASH memory is capable of being completely reloaded via either the RS232 or 485 connection. See §6. for a description of *csimc*, a utility program which is used to perform firmware updates.

The format of each BOOT packet consists of one or more records packed contiguously, each defined as follows.

Field Name	N Bytes	Description
Type	1	Bit 0: data record Bit 1: execution record Bit 2: Data will be written to EEPROM memory Bit 3: Data will be written to FLASH memory
Length	1	number of bytes in Data field
Address	2	If Type Bit 0: address of first byte of Data; MSB first. If Type Bit 1: address to begin execution; MSB first.
Data	<Length>	data to be stored

These records are contained in a file by convention with extension **.cmf** ("CSIMC Firmware"). The format of this file is one line of ASCII text, a newline, then one or more of these records in binary form. The leading text is the version number of the firmware. This version will match the response from issuing the command "=version;" to a running CSIMC node running this boot code. Even though the file is predominantly binary, since the first line is ASCII the version can be conveniently displayed using the following UNIX command:

```
$ head -1 file.cmf
```

The CSIMC powers up and boots from its internal FLASH. This code is bootstrapped during manufacturing using the BDM pod port. The pod is not necessary from then on unless a board becomes so corrupted it can not be reloaded from the network. If this happens the board must be returned to CSI.

Progress during the LAN loading is indicated by LEDs D20-27. The following table shows the value and meaning of each step.

D20-27 Value	LAN Boot Step Description
1	BOOT packet detected, motor stopped
2	Primary bootstrap loader copied from FLASH to RAM
3	FLASH is erased
4	First BOOT packet decoded and written to FLASH

The loader now enters a loop accepting more BOOT packets and loading as they arrive. The count in the LEDs is incremented for each packet to show progress. When the final packet arrives, the watchdog timer is allowed to expire to cause a reboot. If all went well, the node will come up with its new code, lighting all LEDs D20-27.

10.3. Development Tools

Follows are the hardware and software tools used to create and produce a CSIMC.

10.3.1. Debugger Pod

A brand new CSIMC must be loaded using a special interface adaptor known as a BDM12 Pod. This pod is also capable of interrogating memory and processor state and is vital during development and board checkout. This pod is available from Kevin Ross and the Seattle Robotics Society. As of March 2000 the cost was $89.95 plus $4 shipping.

Contact:

Kevin Ross
PO Box 1714
Duvall, WA 9801

FAX 425-788-5985

Related links:

mailto:kevinro@nwlink.com

http://www.nwlink.com/~kevinro/products.html

http://www.nwlink.com/~kevinro/products.html#68HC12

http://www.seattlerobotics.org

The BDM12 includes a cable and connector for CSIMC JP1. Orient so the ribbon cable comes out over the board. To attach the BDM12 to the computer make a serial cable using the following guide:

```
 Computer: DB-9 female          Pod: DB-9 male       Computer: DB25 female

   1-4-6                                                        8-20-6
   2      --------------------------    2    ----------------------     3
   3      --------------------------    3    ----------------------     2
   5      --------------------------    5    ----------------------     7
   8      --------------------------    8    ----------------------     5
```

On the BDM12 itself set SW1=Off SW2=On SW3=Off SW4=Off. Then use the *bdm12* program to create an interactive terminal session. Type "?" for a summary of instructions. Refer to BDM12 and Motorola HC12 documentation for full details.

10.3.2. Embedded C Compiler

All software running on the CSIMC is written in ANSI C using the ICC12 Version 6 Compiler and Tools from ImageCraft. This is a full ANSI C implementation for the HC12 processor. The compiler and support have both been excellent. As of March 2000 the cost is $100 for the Linux version.

Contact:

ImageCraft
706 Colorado Ave.
Palo Alto, CA 94303

Phone (650) 493-9326
FAX (650) 493-9329

Related links:

mailto:info@imagecraft.com

http://www.imagecraft.com/software/mdevtools.html.ImageCraft

10.3.3. Downloading a new CSIMC

Once a BDM12 pod has been connected and the ICC12 compiler are installed on a Linux system, a new CSIMC can be loaded with its software by going to the **telescope/CSIMC/embedded** directory and just typing:

```
make loadall
```

10.4. Equation of Motion

The CSIMC uses a velocity-based pursuit-intercept algorithm as the basis for all motion. It is specially designed to accurately and smoothly pursue and follow arbitrary paths defined by step values stated as a function of time in the presence of small to modest amounts of axis slip.

The update period is about 32ms, or about 31 times per second. At each update the position and velocity of the target is computed. Velocity is computed as the change in position from the previous update, acceleration is the change in velocity. For a fixed target position or velocity, for example from the **mtpos** or **mtvel** variables being assigned, these values are synthesized from the initial conditions and time elapsed since the command was issued. For arbitrary paths, for example from the **mtrack** function, the values are interpolated to the current clock value using a cubic spline fit .

At each update, the ideal intercept velocity to issue is computed as follows:

$$V_g = V_t - K(x_g - x_t)$$

where

$$K = K_V \frac{A_{\max}}{V_{\max}}$$

x_g and V_g denote the current motor or encoder "gun" position and velocity; similarly x_t and V_t denote the current "target"; K_V is the **kv** parameter; and $A_{\max}$ and $V_{\max}$ are the **maxacc** and **maxvel** parameters. This ideal is then mitigated by **maxvel** and **maxacc** with respect to the velocity issued at the previous update. All calculations are performed in motor steps. If there are more encoder steps than motor steps (*i.e.,* **esteps** > **msteps**) the goal accuracy is 0 encoder steps, otherwise it is 0 motor steps.

If the target is moving at constant velocity V_T and is at position $x_t = x_T + V_T t$ at time t, the gun position is the solution to the above nonhomogeneous differential equation:

$$x_g(t) = V_T t + x_T + (x_g(0) - x_T)e^{-Kt}$$

$$V_g(t) = V_T - (x_g(0) - x_T)Ke^{-Kt}$$

$$A_g(t) = (x_g(0) - x_T)K^2 e^{-Kt}$$

If the velocity or acceleration computed from these equations exceeds the respective maxima, the clamping algorithm effectively produces a parabolic trajectory with constant acceleration $A_{\max}$. In this regime, the equations of motion become:

$$x_g(t) = \frac{A_{\max}}{2} t^2 + V_g(0)t + x_g(0)$$

$$V_g(t) = A_{\max} t + V_g(0)$$

$$A_g(t) = A_{\max}$$

The two regimes merge when their accelerations are both $A_{\max}$.

Index

7623 27, 33
@n 22

A/D 20
accuracy 1
ad0 20
address 4
ANSI C 45
API 39
arguments 22

balanced 1
BDM12 44
bdm12 44

C Compiler 45
Circuit Board 4
clock 21
close 28
Comments 17
Configuration 2
Connection 28
control-D 30
core function 22
Csimc 27
csimc.cfg 26, 30
Csimcd 25
csimcd.log 26
Csimcio 33
csi_close 39
csi_f2h 39
csi_f2n 39
csi_intr 39
csi_open 39
csi_r 40
csi_rebootAll 39
csi_rix 40
csi_sopen 39
csi_w 40
csi_wr 40

Daemon 25
Data Types 17
Datalink Protocol 41

editor 27
eerror 18
encoder 1
Encoder circuit 10
epos 18
Equation of Motion 45
Error Codes 13
esign 18
esteps 19
etpos 19
etrack 18
etrig 19
etvel 19
evel 19
Example Configuration 2
Examples 34
Expressions 23

firmware 28
Firmware 43
function arguments 22

global variables 22

Hardware Specifications 1
heatsink 8
Help 29
history 29
homebit 21
host 25
HOST 27
host 33
Host 39

ICC12 45
iedge 20
if caveat 24
ilevel 21
ImageCraft 1, 45
input 1
Input circuit 9
integers 17
Interactive 27
Interruptions 29
ipolar 21

limacc 19
load 29
local variables 22
localhost 27, 30, 33
Log file 26
logging packets 25

maxacc 19
maxvel 19
mdir 19
Motion 18
Motor connector circuit 11
mpos 19
msteps 19
mtpos 19
mtrack 18
mtrig 19
mtvel 19
multi-tasking 2
mvel 19
myaddr 21

nlimbit 21

olevel 21
ontrack 19
output 1
output circuit 9

Packet Format 41
peer 21
Ping 25
plimbit 21
Pod 44
PORT 27
port 27
Port 30
port 33
Power 8
Power supply circuit 9
Power-On 43
precision 1
printf 18
prompt 21
Protocol 17
pullup 5

quadrature 1

Reboot 30
Remote Variables 22
return 22
RS232 5, 7, 29, 39

Script 33
scripts 27
Seattle Robotics 44
Self-Test 12
serial 29
shell 39
Shells 17
SIGHUP 26
Software Specifications 2
Specifications 1
stack variables 22
Statements 23
Statistics 14
stop 18
Switches 4-6
Sync 41
Syntax 17

Threads 17
timeout 19
toffset 19
Token 43
Topology 42
tracing 29
Track 18

Index

undef 22
user defined functions 22

vascale 20
verbose 26
version 21

working 20