The HAL (Hardware Abstraction Layer) Handbook

June 25, 2007

The EMC Team

This handbook is a work in progress. If you are able to help with writing, editing, or graphic preparation please contact any member of the writing team or join and send an email to emc-users@lists.sourceforge.net.

Copyright (c) 2000-6 LinuxCNC.org

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and one Back-Cover Text: "This HAL Handbook is the product of several authors writing for linuxCNC.org. As you find it to be of value in your work, we invite you to contribute to its revision and growth." A copy of the license is included in the section entitled "GNU Free Documentation License". If you do not find the license you may order a copy from Free Software Foundation, Inc. 59 Temple Place, Suite 330 Boston, MA 02111-1307

Contents

I Introduction & Tutorial **8**

1 Introduction **9**

1.1 What is HAL? . . . 9

1.1.1 HAL is based on traditional system design techniques . . . 9

1.1.1.1 Part Selection . . . 9

1.1.1.2 Interconnection Design . . . 9

1.1.1.3 Implementation . . . 10

1.1.1.4 Testing . . . 10

1.1.2 Summary . . . 10

1.2 HAL Concepts . . . 10

1.3 HAL components . . . 12

1.3.1 External Programs with HAL hooks . . . 12

1.3.2 Internal Components . . . 12

1.3.3 Hardware Drivers . . . 12

1.3.4 Tools and Utilities . . . 13

1.4 Tinkertoys, Erector Sets, Legos and the HAL . . . 13

1.4.1 Tower . . . 13

1.4.2 Erector Sets . . . 13

1.4.3 Tinkertoys . . . 14

1.4.4 A Lego Example . . . 14

1.5 Timing Issues In HAL . . . 15

1.6 Dynamic Linking and Configuration . . . 16

2 HAL Tutorial **17**

2.1 Before we start . . . 17

2.1.1 Notation . . . 17

2.1.2 The RTAPI environment . . . 17

2.2 Tab-completion . . . 18

2.3 A Simple Example . . . 18

2.3.1 Loading a realtime component . . . 18

2.3.2 Examining the HAL . . . 18

2.3.3 Making realtime code run . . . 19

2.3.4 Changing parameters . . . 20

2.3.5 Saving the HAL configuration . . . 21

2.3.6 Restoring the HAL configuration . . . 22

2.4 Looking at the HAL with halmeter . . . 22

2.4.1 Starting halmeter . . . 22
2.4.2 Using halmeter . . . 22
2.5 A slightly more complex example. . . . 25
2.5.1 Installing the components . . . 25
2.5.2 Connecting pins with signals . . . 26
2.5.3 Setting up realtime execution - threads and functions . . . 27
2.5.4 Setting parameters . . . 28
2.5.5 Run it! . . . 29
2.6 Taking a closer look with halscope. . . . 29
2.6.1 Starting Halscope . . . 29
2.6.2 Hooking up the "scope probes" . . . 31
2.6.3 Capturing our first waveforms . . . 32
2.6.4 Vertical Adjustments . . . 33
2.6.5 Triggering . . . 33
2.6.6 Horizontal Adjustments . . . 35
2.6.7 More Channels . . . 36

II HAL Reference 37

3 General Reference Information 38
3.1 Notation . . . 38
3.1.1 Typographical Conventions . . . 38
3.1.2 Names . . . 38
3.2 General Naming Conventions . . . 38
3.3 Hardware Driver Naming Conventions . . . 39
3.3.1 Pin/Parameter names . . . 39
3.3.1.1 Examples . . . 40
3.3.2 Function Names . . . 40
3.3.2.1 Examples . . . 40

4 Canonical Device Interfaces 41
4.1 Digital Input . . . 41
4.1.1 Pins . . . 41
4.1.2 Parameters . . . 41
4.1.3 Functions . . . 41
4.2 Digital Output . . . 41
4.2.1 Pins . . . 41
4.2.2 Parameters . . . 42
4.2.3 Functions . . . 42
4.3 Analog Input . . . 42
4.3.1 Pins . . . 42
4.3.2 Parameters . . . 42
4.3.3 Functions . . . 42
4.4 Analog Output . . . 42
4.4.1 Parameters . . . 43
4.4.2 Functions . . . 43

4.5 Encoder . . . 43
4.5.1 Pins . . . 43
4.5.2 Parameters . . . 44
4.5.3 Functions . . . 44

5 Tools and Utilities **45**
5.1 Halcmd . . . 45
5.2 Halmeter . . . 45
5.3 Halscope . . . 45
5.4 Halshow . . . 46
5.4.1 Hal Tree Area . . . 46
5.4.2 Hal Show Area . . . 47
5.4.3 Hal Watch Area . . . 50

6 Hardware Drivers **52**
6.1 Parport . . . 52
6.1.1 Installing . . . 52
6.1.2 Removing . . . 53
6.1.3 Pins . . . 53
6.1.4 Parameters . . . 53
6.1.5 Functions . . . 55
6.1.6 Common problems . . . 55
6.2 probe_parport . . . 55
6.2.1 Installing . . . 55
6.3 AX5214H . . . 55
6.3.1 Installing . . . 56
6.3.2 Removing . . . 56
6.3.3 Pins . . . 56
6.3.4 Parameters . . . 56
6.3.5 Functions . . . 57
6.4 Servo-To-Go . . . 57
6.4.1 Installing: . . . 57
6.4.2 Removing . . . 57
6.4.3 Pins . . . 57
6.4.4 Parameters . . . 58
6.4.5 Functions . . . 58
6.5 Mesa Electronics m5i20 “Anything I/O Card” . . . 58
6.5.1 Removing . . . 59
6.5.2 Pins . . . 59
6.5.3 Parameters . . . 60
6.5.4 Functions . . . 60
6.5.5 Connector pinout . . . 60
6.5.5.1 Connecor P2 . . . 61
6.5.5.2 Connector P3 . . . 61
6.5.5.3 Connector P4 . . . 62
6.5.5.4 LEDs . . . 63

6.6 Vital Systems Motenc-100 and Motenc-LITE . . . 63
6.6.1 Removing . . . 64
6.6.2 Pins . . . 64
6.6.3 Parameters . . . 64
6.6.4 Functions . . . 65
6.7 Pico Systems PPMC (Parallel Port Motion Control) . . . 65
6.7.1 Removing . . . 65
6.7.2 Pins . . . 66
6.7.3 Parameters . . . 66
6.7.4 Functions . . . 67

7 Internal Components **68**
7.1 Stepgen . . . 68
7.1.1 Installing . . . 68
7.1.2 Removing . . . 68
7.1.3 Pins . . . 68
7.1.4 Parameters . . . 70
7.1.5 Step Types . . . 71
7.1.6 Functions . . . 71
7.2 Freqgen . . . 75
7.2.1 Installing . . . 75
7.2.2 Removing . . . 75
7.2.3 Pins . . . 75
7.2.4 Parameters . . . 77
7.2.5 Step Types . . . 77
7.2.6 Functions . . . 77
7.3 PWMgen . . . 78
7.3.1 Installing . . . 78
7.3.2 Removing . . . 78
7.3.3 Pins . . . 78
7.3.4 Parameters . . . 79
7.3.5 Output Types . . . 79
7.3.6 Functions . . . 79
7.4 Encoder . . . 80
7.4.1 Installing . . . 80
7.4.2 Removing . . . 80
7.4.3 Pins . . . 81
7.4.4 Parameters . . . 81
7.4.5 Functions . . . 81
7.5 PID . . . 82
7.5.1 Installing . . . 82
7.5.2 Removing . . . 82
7.5.3 Pins . . . 82
7.5.4 Parameters . . . 82
7.5.5 Functions . . . 84
7.6 Simulated Encoder . . . 85

7.6.1 Installing . . . 85
7.6.2 Removing . . . 85
7.6.3 Pins . . . 85
7.6.4 Parameters . . . 85
7.6.5 Functions . . . 85
7.7 Debounce . . . 86
7.7.1 Installing . . . 86
7.7.2 Removing . . . 86
7.7.3 Pins . . . 86
7.7.4 Parameters . . . 86
7.7.5 Functions . . . 86
7.8 Blocks . . . 87
7.8.1 Available Blocks . . . 87
7.8.2 Installing . . . 87
7.8.3 Removing . . . 88
7.8.4 Pins . . . 88
7.8.5 Parameters . . . 88
7.8.6 Functions . . . 89
7.9 Siggen . . . 90
7.9.1 Installing . . . 90
7.9.2 Removing . . . 90
7.9.3 Pins . . . 90
7.9.4 Parameters . . . 90
7.9.5 Functions . . . 90

8 Virtual Control Panels **91**
8.1 Introduction . . . 91
8.2 pyVCP . . . 91
8.3 Using pyVCP with AXIS . . . 92
8.4 pyVCP Widget reference . . . 93
8.4.0.1 Syntax . . . 93
8.4.1 LED . . . 93
8.4.2 Button . . . 94
8.4.3 Checkbutton . . . 94
8.4.4 Radiobutton . . . 94
8.4.5 Number . . . 95
8.4.6 Bar . . . 95
8.4.7 Meter . . . 95
8.4.8 Spinbox . . . 96
8.4.9 Scale . . . 96
8.4.10 Jogwheel . . . 96
8.4.11 Hbox . . . 97
8.4.12 Vbox . . . 97
8.4.13 Label . . . 97
8.5 VCP: A small example . . . 98
8.6 VCP: Another small example with EMC . . . 99
8.7 VCP Syntax . . . 99
8.7.1 Block . . . 99

III HAL Programming **100**

9 *comp*: a tool for creating HAL modules **101**
9.1 Introduction . . . 101
9.2 Definitions . . . 101
9.3 Instance creation . . . 102
9.4 Syntax . . . 102
9.5 Other restrictions on comp files . . . 103
9.6 Convenience Macros . . . 103
9.7 Compiling `.comp` files in the source tree . . . 104
9.8 Compiling components outside the source tree . . . 104
9.9 Examples . . . 105
9.9.1 constant . . . 105
9.9.2 sincos . . . 105
9.9.3 out8 . . . 105
9.9.4 hal_loop . . . 106

10 Creating Userspace Python Components with the 'hal' module **107**
10.1 Basic usage . . . 107
10.2 Userspace components and delays . . . 108
10.3 Create pins and parameters . . . 108
10.3.1 Changing the prefix . . . 108
10.4 Reading and writing pins and parameters . . . 108
10.4.1 Driving output (HAL_OUT) pins . . . 109
10.4.2 Driving bidirectional (HAL_IO) pins . . . 109
10.5 Exiting . . . 109
10.6 Project ideas . . . 109

A Legal Section **110**
A.1 GNU Free Documentation License Version 1.1, March 2000 . . . 110
A.1.1 GNU Free Documentation License Version 1.1, March 2000 . . . 110

Part I

Introduction & Tutorial

Chapter 1

Introduction

1.1 What is HAL?

HAL stands for Hardware Abstraction Layer. At the highest level, it is simply a way to allow a number of "building blocks" to be loaded and interconnected to assemble a complex system. The "Hardware" part is because HAL was originally designed to make it easier to configure EMC for a wide variety of hardware devices. Many of the building blocks are drivers for hardware devices. However, HAL can do more than just configure hardware drivers.

1.1.1 HAL is based on traditional system design techniques

HAL is based on the same principles that are used to design hardware circuits and systems, so it is useful to examine those principles first.

Any system (including a CNC machine), consists of interconnected components. For the CNC machine, those components might be the main controller, servo amps or stepper drives, motors, encoders, limit switches, pushbutton pendants, perhaps a VFD for the spindle drive, a PLC to run a toolchanger, etc. The machine builder must select, mount and wire these pieces together to make a complete system.

1.1.1.1 Part Selection

The machine builder does not need to worry about how each individual part works. He treats them as black boxes. During the design stage, he decides which parts he is going to use - steppers or servos, which brand of servo amp, what kind of limit switches and how many, etc. The integrator's decisions about which specific components to use is based on what that component does and the specifications supplied by the manufacturer of the device. The size of a motor and the load it must drive will affect the choice of amplifier needed to run it. The choice of amplifier may affect the kinds of feedback needed by the amp and the velocity or position signals that must be sent to the amp from a control.

In the HAL world, the integrator must decide what HAL components are needed. Usually every interface card will require a driver. Additional components may be needed for software generation of step pulses, PLC functionality, and a wide variety of other tasks.

1.1.1.2 Interconnection Design

The designer of a hardware system not only selects the parts, he also decides how those parts will be interconnected. Each black box has terminals, perhaps only two for a simple switch, or dozens for a servo drive or PLC. They need to be wired together. The motors connect to the servo amps, the limit switches connect to the controller, and so on. As the machine builder works on the design, he creates a large wiring diagram that shows how all the parts should be interconnected.

When using HAL, components are interconnected by signals. The designer must decide which signals are needed, and what they should connect.

1.1.1.3 Implementation

Once the wiring diagram is complete it is time to build the machine. The pieces need to be acquired and mounted, and then they are interconnected according to the wiring diagram. In a physical system, each interconnection is a piece of wire that needs to be cut and connected to the appropriate terminals.

HAL provides a number of tools to help "build" a HAL system. Some of the tools allow you to "connect" (or disconnect) a single "wire". Other tools allow you to save a complete list of all the parts, wires, and other information about the system, so that it can be "rebuilt" with a single command.

1.1.1.4 Testing

Very few machines work right the first time. While testing, the builder may use a meter to see whether a limit switch is working or to measure the DC voltage going to a servo motor. He may hook up an oscilloscope to check the tuning of a drive, or to look for electrical noise. He may find a problem that requires the wiring diagram to be changed; perhaps a part needs to be connected differently or replaced with something completely different.

HAL provides the software equivalents of a voltmeter, oscilloscope, signal generator, and other tools needed for testing and tuning a system. The same commands used to build the system can be used to make changes as needed.

1.1.2 Summary

This document is aimed at people who already know how to do this kind of hardware system integration, but who do not know how to connect the hardware to EMC.

The traditional hardware design as described above ends at the edge of the main control. Outside the control are a bunch of relatively simple boxes, connected together to do whatever is needed. Inside, the control is a big mystery – one huge black box that we hope works.

HAL extends this traditional hardware design method to the inside of the big black box. It makes device drivers and even some internal parts of the controller into smaller black boxes that can be interconnected and even replaced just like the external hardware. It allows the "system wiring diagram"to show part of the internal controller, rather than just a big black box. And most importantly it allows the integrator to test and modify the controller using the same methods he would use on the rest of the hardware.

Terms like motors, amps, and encoders are familiar to most machine integrators. When we talk about using extra flexible eight conductor shielded cable to connect an encoder to the servo input board in the computer, the reader immediately understands what it is and is led to the question, "what kinds of connectors will I need to make up each end." The same sort of thinking is essential for the HAL but the specific train of thought may take a bit to get on track. Using HAL words may seem a bit strange at first, but the concept of working from one connection to the next is the same.

This idea of extending the wiring diagram to the inside of the controller is what HAL is all about. If you are comfortable with the idea of interconnecting hardware black boxes, you will probably have little trouble using HAL to interconnect software black boxes.

1.2 HAL Concepts

This section is a glossary that defines key HAL terms but it is a bit different than a traditional glossary because these terms are not arranged in alphabetical order. They are arranged by their relationship or flow in the HAL way of things.

Component: When we talked about hardware design, we referred to the individual pieces as "parts", "building blocks", "black boxes", etc. The HAL equivalent is a "component" or "HAL component". (This document uses "HAL component" when there is likely to be confusion with other kinds of components, but normally just uses "component".) A HAL component is a piece of software with well-defined inputs, outputs, and behavior, that can be installed and interconnected as needed.

Parameter: Many hardware components have adjustments that are not connected to any other components but still need to be accessed. For example, servo amps often have trim pots to allow for tuning adjustments, and test points where a meter or scope can be attached to view the tuning results. HAL components also can have such items, which are referred to as "parameters". There are two types of parameters: Input parameters are equivalent to trim pots - they are values that can be adjusted by the user, and remain fixed once they are set. Output parameters cannot be adjusted by the user - they are equivalent to test points that allow internal signals to be monitored.

Pin: Hardware components have terminals which are used to interconnect them. The HAL equivalent is a "pin" or "HAL pin". ("HAL pin" is used when needed to avoid confusion.) All HAL pins are named, and the pin names are used when interconnecting them. HAL pins are software entities that exist only inside the computer.

Physical_Pin: Many I/O devices have real physical pins or terminals that connect to external hardware, for example the pins of a parallel port connector. To avoid confusion, these are referred to as "physical pins". These are the things that “stick out” into the real world.

Signal: In a physical machine, the terminals of real hardware components are interconnected by wires. The HAL equivalent of a wire is a "signal" or "HAL signal". HAL signals connect HAL pins together as required by the machine builder. HAL signals can be disconnected and reconnected at will (even while the machine is running).

Type: When using real hardware, you would not connect a 24 volt relay output to the +/-10V analog input of a servo amp. HAL pins have the same restrictions, which are based upon their type. Both pins and signals have types, and signals can only be connected to pins of the same type. Currently there are 4 types, as follows:

- BIT - a single TRUE/FALSE or ON/OFF value
- FLOAT - a 32 bit floating point value, with approximately 24 bits of resolution and over 200 bits of dynamic range.
- U32 - a 32 bit unsigned integer, legal values are 0 to +4294967295
- S32 - a 32 bit signed integer, legal values are -2147483648 to +2147483647

Function: Real hardware components tend to act immediately on their inputs. For example, if the input voltage to a servo amp changes, the output also changes automatically. However software components cannot act "automatically". Each component has specific code that must be executed to do whatever that component is supposed to do. In some cases, that code simply runs as part of the component. However in most cases, especially in realtime components, the code must run in a specific sequence and at specific intervals. For example, inputs should be read before calculations are performed on the input data, and outputs should not be written until the calculations are done. In these cases, the code is made available to the system in the form of one or more "functions". Each function is a block of code that performs a specific action. The system integrator can use "threads" to schedule a series of functions to be executed in a particular order and at specific time intervals.

Thread: A "thread" is a list of functions that runs at specific intervals as part of a realtime task. When a thread is first created, it has a specific time interval (period), but no functions. Functions can be added to the thread, and will be executed in order every time the thread runs.

As an example, suppose we have a parport component named hal_parport. That component defines one or more HAL pins for each physical pin. The pins are described in that component's doc section: their names, how each pin relates to the physical pin, are they inverted, can you change polarity, etc. But that alone doesn't get the data from the HAL pins to the physical pins. It takes code to do that, and that is where functions come into the picture. The parport component needs at least two functions: one to read the physical input pins and update the HAL pins, the other to take data from the HAL pins and write it to the physical output pins. Both of these functions are part of the parport driver.

1.3 HAL components

Each HAL component is a piece of software with well-defined inputs, outputs, and behavior, that can be installed and interconnected as needed. This section lists available components and a brief description of what each does. Complete details for each component are available later in this document.

1.3.1 External Programs with HAL hooks

motion A realtime module that accepts NML motion commands and interacts with HAL

iocontrol A user space module that accepts NML I/O commands and interacts with HAL

classicladder A PLC using HAL for all I/O

halui A user space program that interacts with HAL and sends NML commands, it is intended to work as a full User Interface using external knobs & switches

1.3.2 Internal Components

stepgen Software step pulse generator with position loop. See section 7.1

freqgen Software step pulse generator. See section 7.2

encoder Software based encoder counter. See section 7.4

pid Proportional/Integral/Derivative control loops. See section 7.5

siggen A sine/cosine/triangle/square wave generator for testing. See section 7.9

supply a simple source for testing

blocks assorted useful components (mux, demux, or, and, integ, ddt, limit, wcomp, etc.)

1.3.3 Hardware Drivers

hal_ax5214h A driver for the Axiom Measurement & Control AX5241H digital I/O board

hal_m5i20 Mesa Electronics 5i20 board

hal_motenc Vital Systems MOTENC-100 board

hal_parport PC parallel port. See section 6.1

hal_ppmc Pico Systems family of controllers (PPMC, USC and UPC)

hal_stg Servo To Go card (version 1 & 2)

hal_vti Vigilant Technologies PCI ENCDAC-4 controller

1.3.4 Tools and Utilities

halcmd Command line tool for configuration and tuning. See section 5.1

halgui GUI tool for configuration and tuning (not implemented yet).

halmeter A handy multimeter for HAL signals. See section 5.2

halscope A full featured digital storage oscilloscope for HAL signals. See section 5.3

Each of these building blocks is described in detail in later chapters.

1.4 Tinkertoys, Erector Sets, Legos and the HAL

A first introduction to HAL concepts can be mind boggling. Building anything with blocks can be a challenge but some of the toys that we played with as kids can be an aid to building things with the HAL.

1.4.1 Tower

> I'm watching as my son and his six year old daughter build a tower from a box full of random sized blocks, rods, jar lids and such. The aim is to see how tall they can make the tower. The narrower the base the more blocks left to stack on top. But the narrower the base, the less stable the tower. I see them studying both the next block and the shelf where they want to place it to see how it will balance out with the rest of the tower.

The notion of stacking cards to see how tall you can make a tower is a very old and honored way of spending spare time. At first read, the integrator may have gotten the impression that building a HAL was a bit like that. It can be but with proper planning an integrator can build a stable system as complex as the machine at hand requires.

1.4.2 Erector Sets[1]

What was great about the sets was the building blocks, metal struts and angles and plates, all with regularly spaced holes. You could design things and hold them together with the little screws and nuts.

> I got my first erector set for my fourth birthday. I know the box suggested a much older age than I was. Perhaps my father was really giving himself a present. I had a hard time with the little screws and nuts. I really needed four arms, one each for the screwdriver, screw, parts to be bolted together, and nut. Perseverence, along with father's eventual boredom, got me to where I had built every project in the booklet. Soon I was lusting after the bigger sets that were also printed on that paper. Working with those regular sized pieces opened up a world of construction for me and soon I moved well beyond the illustrated projects.

Hal components are not all the same size and shape but they allow for grouping into larger units that will do useful work.In this sense they are like the parts of an Erector set. Some components are long and thin. They essentially connect high level commands to specific physical pins. Other components are more like the rectangular platforms upon which whole machines could be built. An integrator will quickly get beyond the brief examples and begin to bolt together components in ways that are unique to them.

[1] The Erector Set was an invention of AC Gilbert

1.4.3 Tinkertoys[2]

> Wooden Tinker toys had a more humane feel that the cold steel of Erector Sets. The heart of construction with Tinker Toys was a round connector with eight holes equally spaced around the circumference. It also had a hole in the center that was perpendicular to all the holes around the hub.
>
> Hubs were connected with rods of several different lengths. Builders would make large wheels by using these rods as spokes sticking out from the center hub.
>
> My favorite project was a rotating space station. Short spokes radiated from all the holes in the center hub and connected with hubs on the ends of each spoke. These outer hubs were connected to each other with longer spokes. I'd spend hours dreaming of living in such a device, walking from hub to hub around the outside as it slowly rotated producing near gravity in weightless space. Supplies traveled through the spokes in elevators that transfered them to an from rockets docked at the center hub while they transfered their precious cargos.

The idea of one pin or component being the hub for many connections is also an easy concept within the HAL. Examples two and four (see section 2) connect the meter and scope to signals that are intended to go elsewhere. Less easy is the notion of a hub for several incoming signals but that is also possible with proper use of functions within that hub component that handle those signals as they arrive from other components.

Another thought that comes forward from this toy is a mechanical representation of HAL threads. A thread might look a bit like a centipede, caterpillar, or earwig. A backbone of hubs, HAL components, strung together with rods, HAL signals. Each component takes in it own parameters and input pins and passes on output pins and parameters to the next component. Signals travel along the backbone from end to end and are added to or modified by each component in turn.

Threads are all about timing and doing a set of tasks from end to end. A mechanical representation is available with Tinkertoys also when we think of the length of the toy as a measure of the time taken to get from one end to the other. A very different thread or backbone is created by connecting the same set of hubs with different length rods. The total length of the backbone can be changed by the length of rods used to connect the hubs. The order of operations is the same but the time to get from beginning to end is very diferent.

1.4.4 A Lego Example[3]

When Lego blocks first arrived in our stores they were pretty much all the same size and shape. Sure there were half sized one and a few quarter sized as well but that rectangular one did most of the work. Lego blocks interconnected by snapping the holes in the underside of one onto the pins that stuck up on another. By overlapping layers, the joints between could be made very strong, even around corners or tees.

> I watched my children and grandchildren build with legos – the same legos. There are a few thousand of them in an old ratty but heavy duty cardboard box that sits in a corner of the recreation room. It stays there in the open because it was too much trouble to put the box away and then get it back out for every visit and it is always used during a visit. There must be Lego parts in there from a couple dozen different sets. The little booklets that came with them are long gone but the magic of building with interlocking pieces all the same size is something to watch.

Notice the following description of building a set of motion components in the HAL and how much like a wall of lego blocks it is.

[2] Tinkertoy is now a registered trademark of the Hasbro company.
[3] The Lego name is a trademark of the Lego company.

> The motion module exports a pin for each axis in cartesean space, and another pin for each axis in joint space. When it is loaded, it automatically creates a "jumper" signal for each axis, and automatically connects those signals from the joint pin to the cartesean pin. So you automatically have "trivkins" as soon as you load the motion module. (trivkins – trivial kinematics is the case where each motor moves a single axis at 90 degrees to the others)

The motion module is like a pair of legos in a line end to end. Trivkins is just like a single block overlapping the two. The in and out motion pins are plugged into each other by the block resting above. But the parallel goes on.

> If you need some other kinematics, you then load a specific kins component. This component "knows" the names of the pins that the motion module uses for each axis, both joint and cartesean. When the module loads, it again automatically creates signals and connects its own pins to the motion module's pins (which will disconnect the "jumpers"). It could also know the thread names used by the motion module, and could automatically add its own functions to those threads.

Trivkins is removed so that the motion blocks can be spread apart and by using other blocks, a different bridge is built between input and output pins. In Lego terms, trivkins might be a gray block and xxkins might be a yellow block.

> So the net result is that 24 HAL signals and two HAL functions are configured, with no action needed by the integrator other than loading the module. (24 signals are from 6 axis * 2 because we have joint and cartesean * 2 because we have forward and inverse kinematics. Two functions because we have forward and inverse.) Because these HAL signals exist, they can be metered or scoped or whatever for testing. But because both modules know their names and know how to automatically connect them, the integrator doesn't have to know or care.

This kind of automatic HAL configuration is possible because all kinematics modules "plug in" the same way.

1.5 Timing Issues In HAL

Threads is going to take a major intellectual push because unlike the physical wiring models between black boxes that we have said that HAL is based upon, simply connecting two pins with a hal-signal falls far short of the action of the physical case.

True relay logic consists of relays connected together, and when a contact opens or closes, current flows (or stops) immediately. Other coils may change state, etc, and it all just "happens". But in PLC style ladder logic, it doesn't work that way. Usually in a single pass through the ladder, each rung is evaluated in the order in which it appears, and only once per pass. A perfect example is a single rung ladder, with a NC contact in series with a coil. The contact and coil belong to the same relay.

If this were a conventional relay, as soon as the coil is energized, the contacts begin to open and de-energize it. That means the contacts close again, etc, etc. The relay becomes a buzzer.

With a PLC, if the coil is OFF and the contact is closed when the PLC begins to evaluate the rung, then when it finishes that pass, the coil is ON. The fact that turning on the coil opens the contact feeding it is ignored until the next pass. On the next pass, the PLC sees that the contact is open, and de-energizes the coil. So the relay still switches rapidly between on and off, but at a rate determined by how often the PLC evaluates the rung.

In HAL, the function is the code that evaluates the rung(s). In fact a HAL-aware realtime version of ClassicLadder would export a function to do exactly that. Meanwhile, a thread is the thing that runs the function at specific time intervals. Just like you can choose to have a PLC evaluate all its rungs every 10mS, or every second, you can define HAL threads with different periods.

What distinguishes one thread from another is _not_ what the thread does - that is determined by which functions are connected to it. The real distinction is simply how often a thread runs.

In EMC we might have a 15uS thread, a 1mS thread, and a 10mS thread. These would be created based on "Period", "ServoPeriod", and "TrajPeriod" respectively - the actual times would depend on the ini. That is one part of the config process, and although it could be done manually, it would normally be automatic.

The next step is to decide what each thread needs to do. Some of those decisions would also be automatic - the motion module would automatically connect its "PlanTrajectory" function to the TrajPeriod thread, and its "ControlMotion" function to the ServoPeriod thread.

Other connections would be made by the integrator (at least the first time). These might include hooking the STG driver's encoder read and DAC write functions to the servo thread, or hooking stepgen's function to the fast thread, along with the parport function(s) to write the steps to the port.

1.6 Dynamic Linking and Configuration

It is indeed possible to configure HAL with a form of dynamic linking. But it is different than DLLs as used by Microsoft(tm) or shared libraries as used in Linux. Both DLLs and shared libraries essentially say "Here I am, I have this code you might want to use", where "you" is other modules. Then when those other modules or programs are loaded, they say "I need a function called 'X', is there one?" and if the answer is YES, they link to it.

With HAL, a component still says "Here I am, I have this code you might want to use", but "you" is the system integrator. The integrator gets to decide what functions are used and doesn't have to worry about another module needing "function X" and not finding it.

HAL can follow the normal DLL model as well. Although most components will simply export pins, functions, and parameters, and then wait for the integrator (or a saved file) to interconnect them, we can write modules that (attempt to) make connections when they are installed. One specific place where this would work well is kinematics as illustrated in the Lego section 1.4.4 .

Chapter 2

HAL Tutorial

2.1 Before we start

Configuration moves from theory to device – HAL device that is. For those who have had just a bit of computer programming, this section is the "Hello World" of the HAL. As noted above halcmd can be used to create a working system. It is a command line or text file tool for configuration and tuning. The following examples illustrate its setup and operation.

2.1.1 Notation

Command line examples are presented in **`bold typewriter`** font. Responses from the computer will be in `typewriter` font. Text inside square brackets `[like-this]` is optional. Text inside angle brackets `<like-this>` represents a field that can take on different values, and the adjacent paragraph will explain the appropriate values. Text items separated by a vertical bar means that one or the other, but not both, should be present. All command line examples assume that you are in the `emc2/` directory, and paths will be shown accordingly when needed.

2.1.2 The RTAPI environment

RTAPI stands for Real Time Application Programming Interface. Many HAL components work in realtime, and all HAL components store data in shared memory so realtime components can access it. Normal Linux does not support realtime programming or the type of shared memory that HAL needs. Fortunately there are realtime operating systems (RTOS's) that provide the neccessary extensions to Linux. Unfortunately, each RTOS does things a little differently.

To address these differences, the EMC team came up with RTAPI, which provides a consistent way for programs to talk to the RTOS. If you are a programmer who wants to work on the internals of EMC, you may want to study `emc2/src/rtapi/rtapi.h` to understand the API. But if you are a normal person all you need to know about RTAPI is that it (and the RTOS) needs to be loaded into the memory of your computer before you do anything with HAL.

For this tutorial, we are going to assume that you have successfully compiled the emc2/ source tree. In that case, all you need to do is load the required RTOS and RTAPI modules into memory. Just run the following command:

```
emc2$ halrun
halcmd:
```

With the realtime OS and RTAPI loaded, we can move into the first example. Notice that the prompt has changed from the shell's "$" to "halcmd". This is because subsequent commands will be interpreted as HAL commands, not shell commands. `halrun` is a simple shell script, and it is more or less equivalent to running

```
emc2$ realtime start
emc2$ halcmd -kf
```

When `halcmd` exits, halrun stops the realtime system, just like

```
emc2$ realtime stop
```

You can also supply arguments to halrun that are passed on to halcmd, or give the name of a .hal file. Because `halrun` stops the realtime system when it exits, the hal file run in this way will typically end with a command that waits for completion, like `loadrt -w halscope`.

2.2 Tab-completion

Your version of halcmd may include tab-completion. Instead of completing filenames as a shell does, it completes commands with HAL identifiers. Try pressing tab after starting a HAL command:

```
halcmd: lo<TAB>
loadrt    loadusr   lock
halcmd: loadrt d<TAB>
ddt       debounce
```

2.3 A Simple Example

2.3.1 Loading a realtime component

For the first example, we will use a HAL component called `siggen`, which is a simple signal generator. A complete description of the `siggen` component can be found in section 7.9 of this document. It is a realtime component, implemented as a Linux kernel module. To load `siggen` use the `halcmd loadrt` command:

```
halcmd: loadrt siggen
```

2.3.2 Examining the HAL

Now that the module is loaded, it is time to introduce `halcmd`, the command line tool used to configure the HAL. This tutorial will introduce some halcmd features, for a more complete description try `man halcmd`, or see the `halcmd` reference in section 5.1 of this document. The first halcmd feature is the show command. This command displays information about the current state of the HAL. To show all installed components:

```
halcmd: show comp
Loaded HAL Components:
ID     Type  Name                                        PID   State
32769  RT    siggen                                            ready
9775   User  halcmd9775                                   9775 initializing
```

Since `halcmd` itself is a HAL component, it will always show up in the list[1]. The list also shows the `siggen` component that we installed in the previous step. The "RT" under "Type" indicates that `siggen` is a realtime component.

Next, let's see what pins `siggen` makes available:

[1]The number after halcmd in the component list is the process ID. It is possible to run more than one copy of halcmd at the same time (in different windows for example), so the PID is added to the end of the name to make it unique.

```
halcmd: show pin
Component Pins:
Owner  Type  Dir    Value      Name 02    float -W   0.00000e+00  siggen.0.cosine
32769  float OUT  0.00000e+00  siggen.0.sawtooth
32769  float OUT  0.00000e+00  siggen.0.sine
32769  float OUT  0.00000e+00  siggen.0.square
32769  float OUT  0.00000e+00  siggen.0.triangle
```

This command displays all of the pins in the HAL - a complex system could have dozens or hundreds of pins. But right now there are only five pins. All five of these pins are floating point, and all five carry data out of the `siggen` component. Since we have not yet executed the code contained within the component, all the pins have a value of zero.

The next step is to look at parameters:

```
halcmd: show param
Parameters:
Owner  Type  Dir    Value      Name
 32769    float  RW   1.00000e+00  siggen.0.amplitude
 32769    float  RW   1.00000e+00  siggen.0.frequency
 32769    float  RW   0.00000e+00  siggen.0.offset
 32769    s32    RO            0   siggen.0.update.time
 32769    s32    RW            0   siggen.0.update.tmax
```

The show param command shows all the parameters in the HAL. Right now each parameter has the default value it was given when the component was loaded. Note the column labeled `Dir`. The parameters labeled `-W` are writeable ones that are never changed by the component itself, instead they are meant to be changed by the user to control the component. We will see how to do this later. Parameters labeled `R-` are read only parameters. They can be changed only by the component. Finally, parameter labeled `RW` are read-write parameters. That means that thay are changed by the component, but can also be changed by the user. Note: the parameters `siggen.0.update.time` and `siggen.0.update.tmax` are for debugging purposes, and won't be covered in this section.

Most realtime components export one or more functions to actually run the realtime code they contain. Let's see what function(s) `siggen` exported:

```
halcmd: show funct
Exported Functions:
Owner CodeAddr   Arg    FP  Users  Name
 32769   b7f74ac5 b7d0c0b4 YES    0   siggen.0.update
```

The siggen component exported a single function. It requires floating point. It is not currently linked to any threads, so "users" is zero[2].

2.3.3 Making realtime code run

To actually run the code contained in the function `siggen.0.update`, we need a realtime thread. Eventually `halcmd` will have a `newthread` command that can be used to create a thread, but that requires some significant internal changes. For now, we have a component called `threads` that is used to create a new thread. Lets create a thread called `test-thread` with a period of 1mS (1000000nS):

```
halcmd: loadrt threads name1=test-thread period1=1000000
```

Let's see if that worked:

[2]The codeaddr and arg fields were used in development, and should probably be removed from the halcmd listing.

```
halcmd: show thread
Realtime Threads:
   Period   FP   Name    (Time, Max-Time)
     999849 YES  test-thread    ( 0, 0 )
```

It did. The period is not exactly 1000000nS because of hardware limitations, but we have a thread that runs at approximately the correct rate, and which can handle floating point functions. The next step is to connect the function to the thread:

```
halcmd: addf siggen.0.update test-thread
```

Up till now, we've been using `halcmd` only to look at the HAL. However, this time we used the `addf` (add function) command to actually change something in the HAL. We told `halcmd` to add the function `siggen.0.update` to the thread `test-thread`, and if we look at the thread list again, we see that it succeeded:

```
halcmd: show thread
Realtime Threads:
   Period   FP   Name    (Time, Max-Time)
     999849 YES  test-thread    ( 0, 0 )
                 1 siggen.0.update
```

There is one more step needed before the `siggen` component starts generating signals. When the HAL is first started, the thread(s) are not actually running. This is to allow you to completely configure the system before the realtime code starts. Once you are happy with the configuration, you can start the realtime code like this:

```
halcmd: start
```

Now the signal generator is running. Let's look at its output pins:

```
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 32769    float OUT   2.12177e-01  siggen.0.cosine
 32769    float OUT  -5.64055e-01  siggen.0.sawtooth
 32769    float OUT   9.79820e-01  siggen.0.sine
 32769    float OUT  -1.00000e+00  siggen.0.square
 32769    float OUT   1.28110e-01  siggen.0.triangle
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 32769    float OUT   5.19530e-01  siggen.0.cosine
 32769    float OUT   6.73893e-01  siggen.0.sawtooth
 32769    float OUT  -8.54452e-01  siggen.0.sine
 32769    float OUT   1.00000e+00  siggen.0.square
 32769    float OUT   3.47785e-01  siggen.0.triangle
```

We did two `show pin` commands in quick succession, and you can see that the outputs are no longer zero. The sine, cosine, sawtooth, and triangle outputs are changing constantly. The square output is also working, however it simply switches from +1.0 to -1.0 every cycle.

2.3.4 Changing parameters

The real power of HAL is that you can change things. For example, we can use the `setp` command to set the value of a parameter. Let's change the amplitude of the signal generator from 1.0 to 5.0:

```
halcmd: setp siggen.0.amplitude 5
emc2$
```

Check the parameters and pins again:

```
halcmd: setp siggen.0.amplitude 5
halcmd: show param
Parameters:
Owner  Type  Dir    Value      Name
 32769    float  RW   5.00000e+00  siggen.0.amplitude
 32769    float  RW   1.00000e+00  siggen.0.frequency
 32769    float  RW   0.00000e+00  siggen.0.offset
 32769    s32    RO          397   siggen.0.update.time
 32769    s32    RW       109100   siggen.0.update.tmax
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 32769    float OUT   4.78453e+00  siggen.0.cosine
 32769    float OUT  -4.53106e+00  siggen.0.sawtooth
 32769    float OUT   1.45198e+00  siggen.0.sine
 32769    float OUT  -5.00000e+00  siggen.0.square
 32769    float OUT   4.02213e+00  siggen.0.triangle
```

Note that the value of parameter `siggen.0.amplitude` has changed to 5.000, and that the pins now have larger values.

2.3.5 Saving the HAL configuration

Most of what we have done with `halcmd` so far has simply been viewing things with the `show` command. However two of the commands actually changed things. As we design more complex systems with HAL, we will use many commands to configure things just the way we want them. HAL has the memory of an elephant, and will retain that configuration until we shut it down. But what about next time? We don't want to manually enter a bunch of commands every time we want to use the system. We can save the configuration of the entire HAL with a single command:

```
halcmd: save
# components
loadrt threads name1=test-thread period1=1000000
loadrt siggen
# signals
# links
# parameter values
setp siggen.0.amplitude  5.00000e+00
setp siggen.0.frequency  1.00000e+00
setp siggen.0.offset  0.00000e+00
# realtime thread/function links
addf siggen.0.update test-thread
```

The output of the `save` command is a sequence of HAL commands. If you start with an "empty" HAL and run all these commands, you will get the configuration that existed when the `save` command was issued. To save these commands for later use, we simply redirect the output to a file:

```
halcmd: save all saved.hal
```

2.3.6 Restoring the HAL configuration

To restore the HAL configuration stored in `saved.hal`, we need to execute all of those HAL commands. To do that, we use `halcmd -f <filename>` which reads commands from a file:

```
emc2$ halcmd -f saved.hal
```

2.4 Looking at the HAL with halmeter

You can build very complex HAL systems without ever using a graphical interface. However there is something satisfying about seeing the result of your work. The first and simplest GUI tool for the HAL is halmeter. It is a very simple program that is the HAL equivalent of the handy Fluke multimeter (or Simpson analog meter for the old timers).

We will use the siggen component again to check out halmeter. If you just finished the previous example, then siggen is already loaded. If not, we can load it just like we did before:

```
emc2$ halrun
halcmd: loadrt siggen
halcmd: loadrt threads name1=test-thread period1=1000000
halcmd: addf siggen.0.update test-thread
halcmd: start
halcmd: setp siggen.0.amplitude 5
```

2.4.1 Starting halmeter

At this point we have the siggen component loaded and running. It's time to start halmeter. Since halmeter is a GUI app, X must be running.

```
halcmd: loadusr halmeter
```

At the same time, a halmeter window opens on your screen, looking something like figure 2.1.

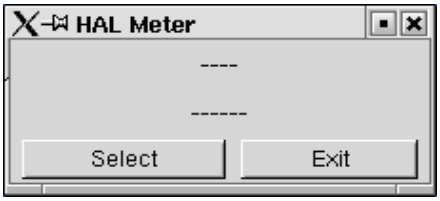


Figure 2.1: Halmeter at startup, nothing selected

2.4.2 Using halmeter

The meter in figure 2.1 isn't very useful, because it isn't displaying anything. To change that, click on the 'Select' button, which will open the probe selection dialog (figure 2.2).

This dialog has three tabs. The first tab displays all of the HAL pins in the system. The second one displays all the signals, and the third displays all the parameters. We would like to look at the pin `siggen.0.triangle` first, so click on it then click the 'OK' button. The probe selection dialog will close, and the meter looks something like figure 2.3.

You should see the value changing as siggen generates its triangle wave. Halmeter refreshes its display about 5 times per second.

If you want to quickly look at a number of pins, you can use the 'Accept' button in the source selection dialog. Click on 'Select' to open the dialog again. This time, click on another pin, like

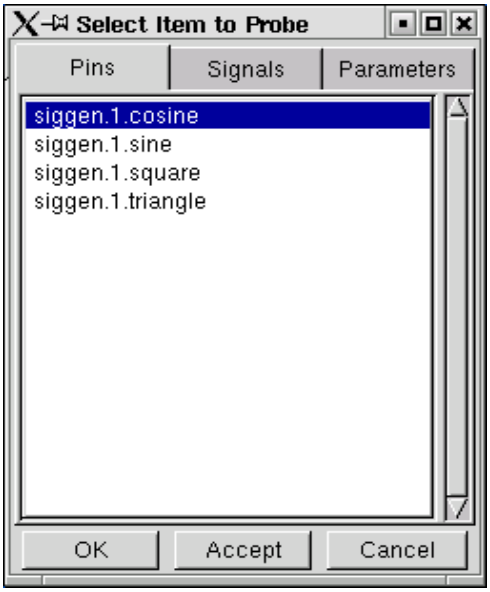


Figure 2.2: Halmeter source selection dialog

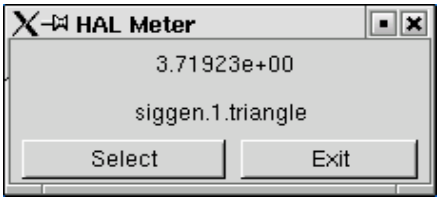


Figure 2.3: Halmeter displaying the value of a pin

siggen.0.cosine, and then click 'Accept'. When you click 'Accept', the meter immediately begins to display the newly selected item, but the dialog does not close. Try displaying a parameter instead of a pin. Click on the 'Parameters' tab, then select a parameter and click 'Accept' again. You can very quickly move the "meter probes" from one item to the next with a couple of clicks.

To shut down halmeter, just click the exit button.

If you want to look at more than one pin, signal, or parameter at a time, you can just start more halmeters. The halmeter window was intentionally made very small so you could have a lot of them on the screen at once. [3]

[3] Halmeter is due for a rewrite. The rewrite will do a number of things to make it nicer. Scientific notation will go away - it is a pain to read. Some form of ranging (including autoranging) will be added to allow it to display a wide range of numbers without using scientific notation. An "analog bar graph" display will also be added to give a quick indication of trends. When the rewrite is done, these screenshots and the accompanying text will be revised to match the new version.

2.5 A slightly more complex example.

Up till now we have only loaded one HAL component. But the whole idea behind the HAL is to allow you to load and connect a number of simple components to make up a complex system. The next example will use two components.

Before we can begin building this new example, we want to start with a clean slate. If you just finished one of the previous examples, we need to remove the all components and reload the RTAPI and HAL libraries:

```
halcmd: exit
emc2$ halrun
```

2.5.1 Installing the components

Now we are going to load the step pulse generator component. For a detailed description of this component refer to section 7.2. For now, we can skip the details, and just run the following commands:[4]

```
halrun: loadrt freqgen step_type=0,0
halcmd: loadrt siggen
halcmd: loadrt threads name1=fast fp1=0 period1=50000 name2=slow period2=1000000
```

The first command loads two step generators, both configured to generate stepping type 0. The second command loads our old friend siggen, and the third one creates two threads, a fast one with a period of 50 micro-seconds and a slow one with a period of 1mS. The fast thread doesn't support floating point functions.

As before, we can use `halcmd show` to take a look at the HAL. This time we have a lot more pins and parameters than before:

```
halcmd: show pin
Component Pins:
Owner  Type  Dir    Value      Name
 03    float -W   0.00000e+00  siggen.0.cosine
 03    float -W   0.00000e+00  siggen.0.sawtooth
 03    float -W   0.00000e+00  siggen.0.sine
 03    float -W   0.00000e+00  siggen.0.square
 03    float -W   0.00000e+00  siggen.0.triangle
 02    s32   -W            0   freqgen.0.counts
 02    bit   -W     FALSE      freqgen.0.dir
 02    float -W   0.00000e+00  freqgen.0.position
 02    bit   -W     FALSE      freqgen.0.step
 02    float R-   0.00000e+00  freqgen.0.velocity
 02    s32   -W            0   freqgen.1.counts
 02    bit   -W     FALSE      freqgen.1.dir
 02    float -W   0.00000e+00  freqgen.1.position
 02    bit   -W     FALSE      freqgen.1.step
 02    float R-   0.00000e+00  freqgen.1.velocity
halcmd: show param
Parameters:
Owner  Type  Dir    Value      Name
 03    float  -W   1.00000e+00  siggen.0.amplitude
 03    float  -W   1.00000e+00  siggen.0.frequency
 03    float  -W   0.00000e+00  siggen.0.offset
```

[4]The "\" at the end of a long line indicates line wrapping (needed for formatting this document). When entering the commands at the command line, simply skip the "\" (do not hit enter) and keep typing from the following line.

```
 02    u32    -W     000000001  freqgen.0.dirhold
 02    u32    -W     000000001  freqgen.0.dirsetup
 02    float  R-   0.00000e+00  freqgen.0.frequency
 02    float  -W   0.00000e+00  freqgen.0.maxaccel
 02    float  -W   1.00000e+15  freqgen.0.maxfreq
 02    float  -W   1.00000e+00  freqgen.0.position-scale
 02    s32    R-            0   freqgen.0.rawcounts
 02    u32    -W     000000001  freqgen.0.steplen
 02    u32    -W     000000001  freqgen.0.stepspace
 02    float  -W   1.00000e+00  freqgen.0.velocity-scale
 02    u32    -W     000000001  freqgen.1.dirhold
 02    u32    -W     000000001  freqgen.1.dirsetup
 02    float  R-   0.00000e+00  freqgen.1.frequency
 02    float  -W   0.00000e+00  freqgen.1.maxaccel
 02    float  -W   1.00000e+15  freqgen.1.maxfreq
 02    float  -W   1.00000e+00  freqgen.1.position-scale
 02    s32    R-            0   freqgen.1.rawcounts
 02    u32    -W     000000001  freqgen.1.steplen
 02    u32    -W     000000001  freqgen.1.stepspace
 02    float  -W   1.00000e+00  freqgen.1.velocity-scale
```

2.5.2 Connecting pins with signals

What we have is two step pulse generators, and a signal generator. Now it is time to create some HAL signals to connect the two components. We are going to pretend that the two step pulse generators are driving the X and Y axis of a machine. We want to move the table in circles. To do this, we will send a cosine signal to the X axis, and a sine signal to the Y axis. The siggen module creates the sine and cosine, but we need "wires" to connect the modules together. In the HAL, "wires" are called signals. We need to create two of them. We can call them anything we want, for this example they will be `X_vel` and `Y_vel`. To create them we use the the `newsig` command. We also need to specify the type of data that will flow through these "wires", in this case it is floating point:

```
halcmd: newsig X_vel float
halcmd: newsig Y_vel float
```

To make sure that worked, we can look at all the signals:

```
halcmd: show sig
Signals:
Type      Value      Name
float   0.00000e+00  X_vel
float   0.00000e+00  Y_vel
```

The next step is to connect the signals to component pins. The signal `X_vel` is intended to run from the cosine output of the signal generator to the velocity input of the first step pulse generator. The first step is to connect the signal to the signal generator output. To connect a signal to a pin we use the `linksp` command.

```
halcmd: linksp X_vel siggen.0.cosine
```

To see the effect of the `linksp` command, we show the signals again:

```
halcmd: show sig
ignals:
Type      Value      Name
float   0.00000e+00  X_vel
                        <== siggen.0.cosine
float   0.00000e+00  Y_vel
```

When a signal is connected to one or more pins, the show command lists the pins immediately following the signal name. The "arrow" shows the direction of data flow - in this case, data flows from pin `siggen.0.cosine` to signal `X_vel`. Now let's connect the `X_vel` to the velocity input of a step pulse generator:

```
halcmd: linksp X_vel freqgen.0.velocity
```

We can also connect up the Y axis signal `Y_vel`. It is intended to run from the sine output of the signal generator to the input of the second step pulse generator:

```
halcmd: linksp Y_vel siggen.0.sine
halcmd: linksp Y_vel freqgen.1.velocity
```

Now let's take a final look at the signals and the pins connected to them:

```
halcmd: show sig
Signals:
Type      Value      Name
float   0.00000e+00  X_vel
                        <== siggen.0.cosine
                        ==> freqgen.0.velocity
float   0.00000e+00  Y_vel
                        <== siggen.0.sine
                        ==> freqgen.1.velocity
```

The `show sig` command makes it clear exactly how data flows through the HAL. For example, the `X_vel` signal comes from pin `siggen.0.cosine`, and goes to pin `freqgen.0.velocity`.

2.5.3 Setting up realtime execution - threads and functions

Thinking about data flowing through "wires" makes pins and signals fairly easy to understand. Threads and functions are a little more difficult. Functions contain the computer instructions that actually get things done. Thread are the method used to make those instructions run when they are needed. First let's look at the functions available to us:

```
halcmd: show funct
Exported Functions:
Owner CodeAddr   Arg    FP  Users  Name
 03   D89051C4 D88F10FC YES    0   siggen.0.update
 02   D8902868 D88F1054 YES    0   freqgen.capture_position
 02   D8902498 D88F1054 NO     0   freqgen.make_pulses
 02   D89026F0 D88F1054 YES    0   freqgen.update_freq
```

In general, you will have to refer to the documentation for each component to see what its functions do. In this case, the function `siggen.0.update` is used to update the outputs of the signal generator. Every time it is executed, it calculates the values of the sine, cosine, triangle, and square outputs. To make smooth signals, it needs to run at specific intervals.

The other three functions are related to the step pulse generators:

The first one, `freqgen.capture_position`, is used for position feedback. It captures the value of an internal counter that counts the step pulses as they are generated. Assuming no missed steps, this counter indicates the position of the motor.

The main function for the step pulse generator is `freqgen.make_pulses`. Every time `make_pulses` runs it decides if it is time to take a step, and if so sets the outputs accordingly. For smooth step pulses, it should run as frequently as possible. Because it needs to run so fast, `make_pulses` is

highly optimized and performs only a few calculations. Unlike the others, it does not need floating point math.

The last function, `freqgen.update_freq`, is responsible for doing scaling and some other calculations that need to be performed only when the frequency command changes.

What this means for our example is that we want to run `siggen.0.update` at a moderate rate to calculate the sine and cosine values. Immediately after we run `siggen.0.update`, we want to run `freqgen.update_freq` to load the new values into the step pulse generator. Finally we need to run `freqgen.make_pulses` as fast as possible for smooth pulses. Because we don't use position feedback, we don't need to run `freqgen.capture_position` at all.

We run functions by adding them to threads. Each thread runs at a specific rate. Let's see what threads we have available:

```
halcmd: show thread
Realtime Threads:
   Period   FP   Name
    1005720 YES  slow    ( 0, 0 )
      50286 NO   fast    ( 0, 0 )
```

The two threads were created when we loaded `threads`. The first one, `slow`, runs every millisecond, and is capable of running floating point functions. We will use it for s`iggen.0.update` and `freqgen.update_freq`. The second thread is `fast`, which runs every 50 microseconds, and does not support floating point. We will use it for `freqgen.make_pulses`. To connect the functions to the proper thread, we use the `addf` command. We specify the function first, followed by the thread:

```
halcmd: addf siggen.0.update slow
halcmd: addf freqgen.update_freq slow
halcmd: addf freqgen.make_pulses fast
```

After we give these commands, we can run the `show thread` command again to see what happened:

```
halcmd: show thread
Realtime Threads:
   Period   FP   Name    (Time, Max-Time)
    1005720 YES  slow    ( 0, 0 )
                  1 siggen.0.update
                  2 freqgen.update-freq
      50286 NO   fast    ( 0, 0 )
                  1 freqgen.make-pulses
```

Now each thread is followed by the names of the functions, in the order in which the functions will run.

2.5.4 Setting parameters

We are almost ready to start our HAL system. However we still need to adjust a few parameters. By default, the siggen component generates signals that swing from +1 to -1. For our example that is fine, we want the table speed to vary from +1 to -1 inches per second. However the scaling of the step pulse generator isn't quite right. By default, it generates an output frequency of 1 step per second with an input of 1.000. It is unlikely that one step per second will give us one inch per second of table movement. Let's assume instead that we have a 5 turn per inch leadscrew, connected to a 200 step per rev stepper with 10x microstepping. So it takes 2000 steps for one revolution of the screw, and 5 revolutions to travel one inch. that means the overall scaling is 10000 steps per inch. We need to multiply the velocity input to the step pulse generator by 10000 to get the proper output. That is exactly what the parameter `freqgen.n.velocity-scale` is for. In this case, both the X and Y axis have the same scaling, so we set the scaling parameters for both to 10000:

```
halcmd: setp freqgen.0.velocity-scale 10000
halcmd: setp freqgen.1.velocity-scale 10000
```

This velocity scaling means that when the pin `freqgen.0.velocity` is 1.000, the step generator will generate 10000 pulses per second (10KHz). With the motor and leadscrew described above, that will result in the axis moving at exactly 1.000 inches per second. This illustrates a key HAL concept - things like scaling are done at the lowest possible level, in this case in the step pulse generator. The internal signal `X_vel` is the velocity of the table in inches per second, and other components such as `siggen` don't know (or care) about the scaling at all. If we changed the leadscrew, or motor, we would change only the scaling parameter of the step pulse generator.

2.5.5 Run it!

We now have everything configured and are ready to start it up. Just like in the first example, we use the `start` command:

```
halcmd: start
```

Although nothing appears to happen, inside the computer the step pulse generator is cranking out step pulses, varying from 10KHz forward to 10KHz reverse and back again every second. Later in this tutorial we'll see how to bring those internal signals out to run motors in the real world, but first we want to look at them and see what is happening.

2.6 Taking a closer look with halscope.

The previous example generates some very interesting signals. But much of what happens is far too fast to see with halmeter. To take a closer look at what is going on inside the HAL, we want an oscilloscope. Fortunately HAL has one, called halscope.

2.6.1 Starting Halscope

Halscope has two parts - a realtime part that is loaded as a kernel module, and a user part that supplies the GUI and display. However, you don't need to worry about this, because the userspace portion will automatically request that the realtime part be loaded.

```
halcmd: loadusr halscope
```

The scope GUI window will open, immediately followed by a "Realtime function not linked" dialog that looks like figure 2.4[5].

This dialog is where you set the sampling rate for the oscilloscope. For now we want to sample once per millisecond, so click on the 1.03mS thread "slow" (formerly "siggen.thread", see footnote), and leave the multiplier at 1. We will also leave the record length at 4047 samples, so that we can use up to four channels at one time. When you select a thread and then click "OK", the dialog disappears, and the scope window looks something like figure 2.5.

[5]Several of these screen captures refer to threads named "siggen.thread" and "stepgen.thread" instead of "slow" and "fast". When the screenshots were captured, the "threads" component didn't exist, and a different method was used to create threads, giving them different names. Also, the screenshots show pins, etc, as "stepgen.xxx" rather than "freqgen.xxx". The original name of the freqgen module was stepgen, and I haven't gotten around to re-doing all the screen shots since it was renamed. The name "stepgen" now refers to a different step pulse generator, one that accepts position instead of velocity commands. Both are described in detail later in this document.

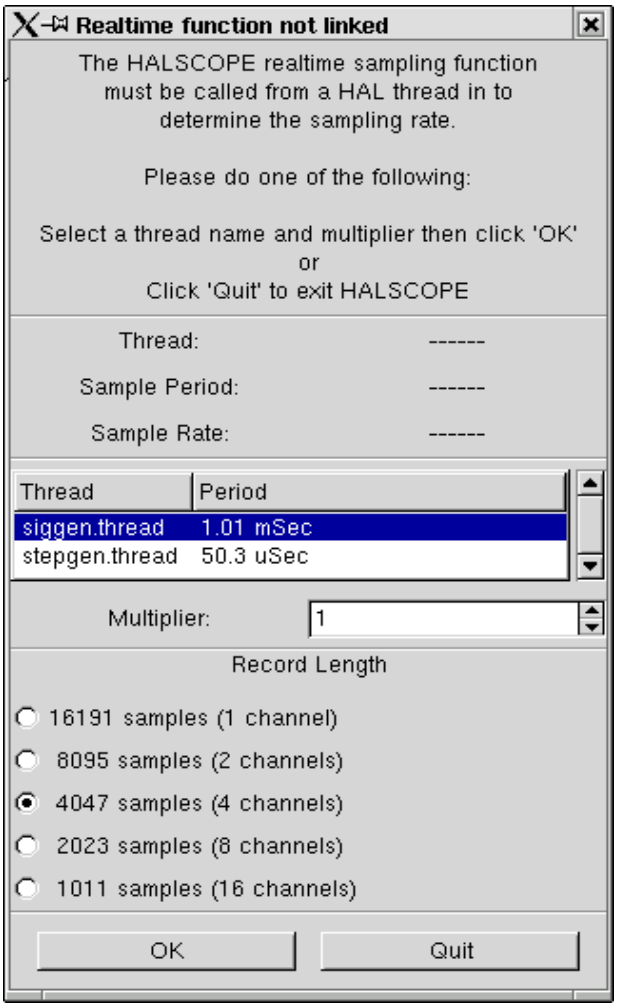


Figure 2.4: "Realtime function not linked" dialog

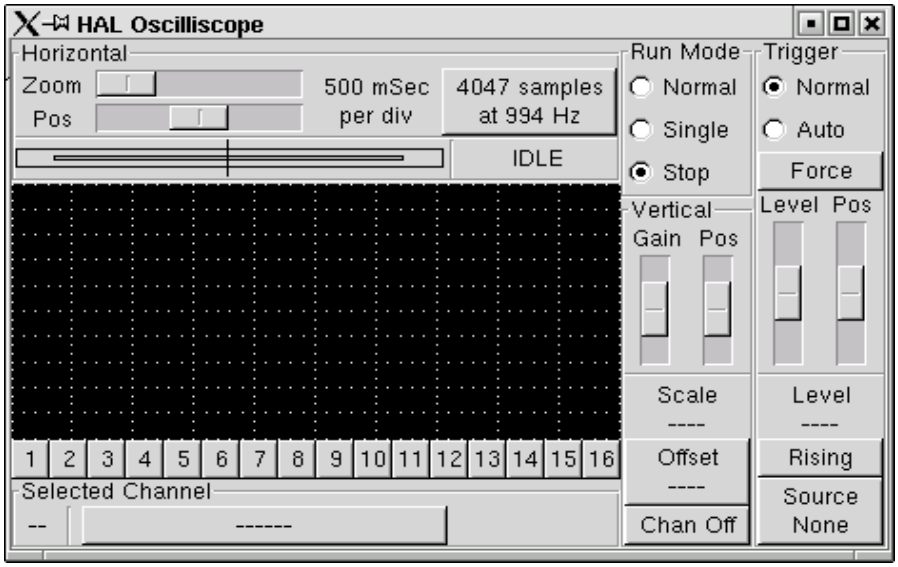


Figure 2.5: Initial scope window

2.6.2 Hooking up the "scope probes"

At this point, Halscope is ready to use. We have already selected a sample rate and record length, so the next step is to decide what to look at. This is equivalent to hooking "virtual scope probes" to the HAL. Halscope has 16 channels, but the number you can use at any one time depends on the record length - more channels means shorter records, since the memory available for the record is fixed at approximately 16,000 samples.

The channel buttons run across the bottom of the halscope screen. Click button "1", and you will see the "Select Channel Source" dialog, figure 2.6. This dialog is very similar to the one used by Halmeter. We would like to look at the signals we defined earlier, so we click on the "Signals" tab, and the dialog displays all of the signals in the HAL (only two for this example).

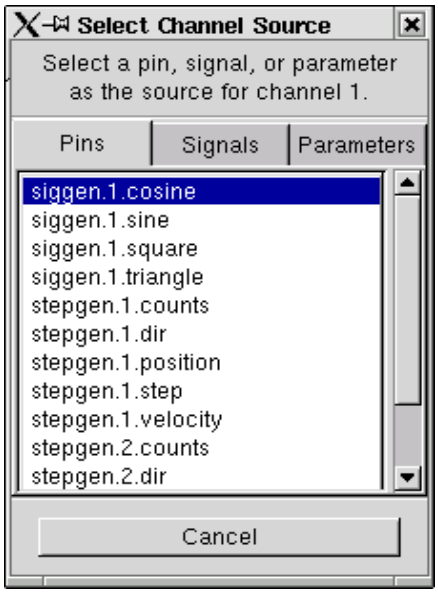


Figure 2.6: Select Channel Source dialog

To choose a signal, just click on it. In this case, we want to use channel 1 to display the signal "X_vel". When we click on "X_vel", the dialog closes and the channel is now selected. The channel 1 button is pressed in, and channel number 1 and the name "X_vel" appear below the row of buttons. That display always indicates the selected channel - you can have many channels on the screen, but the selected one is highlighted, and the various controls like vertical position and scale always work on the selected one. To add a signal to channel 2, click the "2" button. When the dialog pops up, click the "Signals" tab, then click on "Y_vel".

We also want to look at the square and triangle wave outputs. There are no signals connected to those pins, so we use the "Pins" tab instead. For channel 3, select "siggen.0.triangle" and for channel 4, select "siggen.0.square".

2.6.3 Capturing our first waveforms

Now that we have several probes hooked to the HAL, it's time to capture some waveforms. To start the scope, click the "Normal" button in the "Run Mode" section of the screen (upper right). Since we have a 4000 sample record length, and are acquiring 1000 samples per second, it will take halscope about 2 seconds to fill half of its buffer. During that time a progress bar just above the main screen will show the buffer filling. Once the buffer is half full, the scope waits for a trigger. Since we haven't configured one yet, it will wait forever. To manually trigger it, click the "Force" button in the "Trigger" section at the top right. You should see the remainder of the buffer fill, then the screen will display the captured waveforms. The result will look something like figure 2.7.

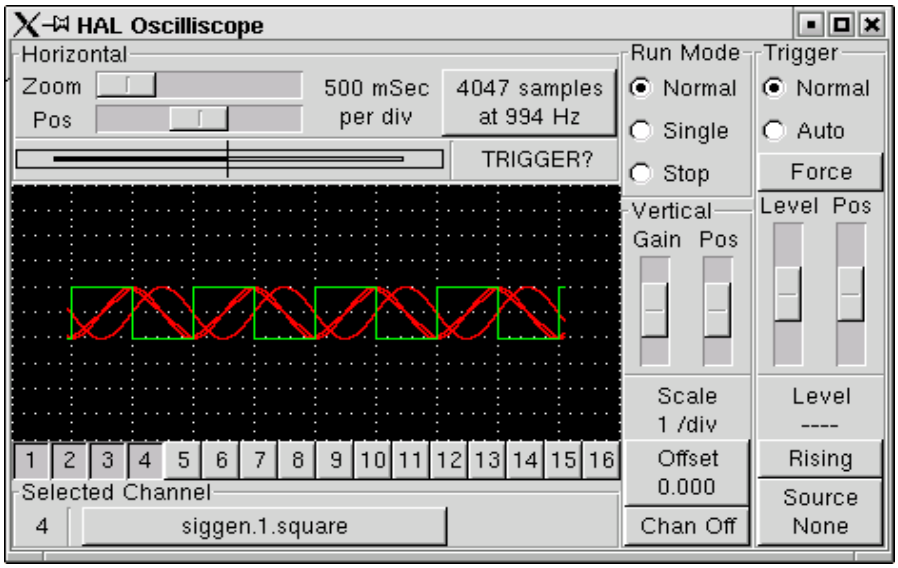


Figure 2.7: Captured Waveforms

The "Selected Channel" box at the bottom tells you that the green trace is the currently selected one, channel 4, which is displaying the value of the pin "siggen.1.square". Try clicking channel buttons 1 through 3 to highlight the other three traces.

2.6.4 Vertical Adjustments

The traces are rather hard to distinguish since all four are on top of each other. To fix this, we use the "Vertical" controls in the box to the right of the screen. These controls act on the currently selected channel. When adjusting the gain, notice that it covers a huge range - unlike a real scope, this one can display signals ranging from very tiny (pico-units) to very large (Tera-units). The position control moves the displayed trace up and down over the height of the screen only. For larger adjustments the offset button should be used (see the halscope reference in section 5.3 for details).

2.6.5 Triggering

Using the "Force" button is a rather unsatisfying way to trigger the scope. To set up real triggering, click on the "Source" button at the bottom right. It will pop up the "Trigger Source" dialog, which is simply a list of all the probes that are currently connected (Figure 2.8). Select a probe to use for triggering by clicking on it. For this example we will use channel 3, the triangle wave.

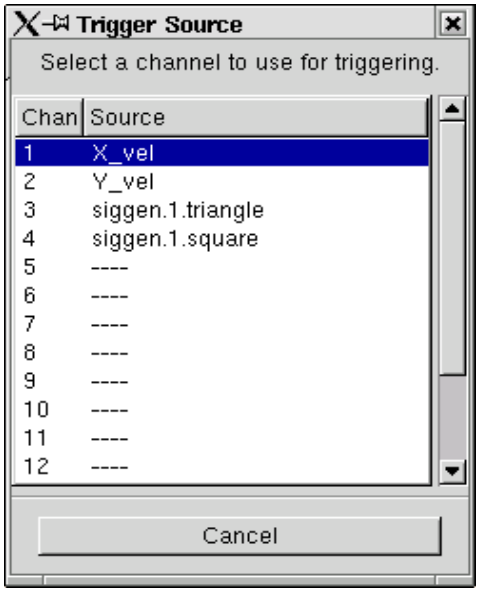


Figure 2.8: Trigger Source Dialog

After setting the trigger source, you can adjust the trigger level and trigger position using the sliders in the "Trigger" box along the right edge. The level can be adjusted from the top to the bottom of the screen, and is displayed below the sliders. The position is the location of the trigger point within the overall record. With the slider all the way down, the trigger point is at the end of the record, and halscope displays what happened before the trigger point. When the slider is all the way up, the trigger point is at the beginning of the record, displaying what happened after it was triggered. The trigger point is visible as a vertical line in the progress box above the screen. The trigger polarity can be changed by clicking the button just below the trigger level display. Note that changing the trigger position stops the scope, once the position is adjusted you restart the scope by clicking the "Normal" button in the "Run Mode" box.

Now that we have adjusted the vertical controls and triggering, the scope display looks something like figure 2.9.

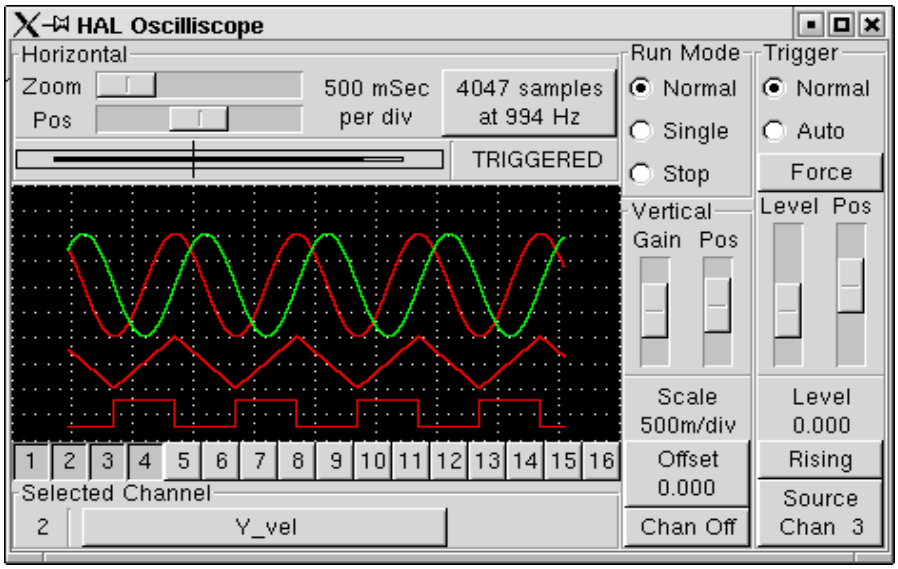


Figure 2.9: Waveforms with Triggering

2.6.6 Horizontal Adjustments

To look closely at part of a waveform, you can use the zoom slider at the top of the screen to expand the waveforms horizontally, and the position slider to determine which part of the zoomed waveform is visible. However, sometimes simply expanding the waveforms isn't enough and you need to increase the sampling rate. For example, we would like to look at the actual step pulses that are being generated in our example. Since the step pulses may be only 50uS long, sampling at 1KHz isn't fast enough. To change the sample rate, click on the button that displays the record length and sample rate to bring up the "Select Sample Rate" dialog, figure . For this example, we will click on the 50uS thread, "fast", which gives us a sample rate of about 20KHz. Now instead of displaying about 4 seconds worth of data, one record is 4000 samples at 20KHz, or about 0.20 seconds.

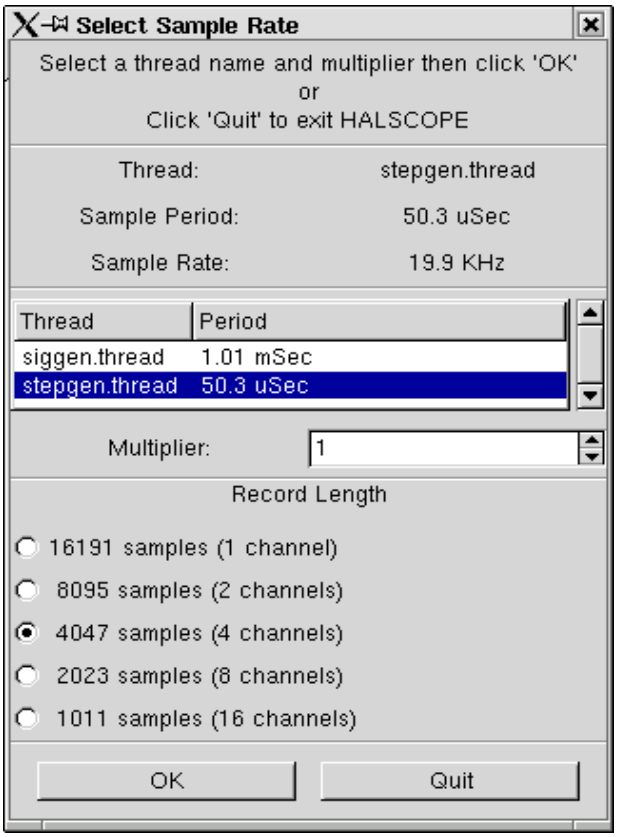


Figure 2.10: Sample Rate Dialog

2.6.7 More Channels

Now let's look at the step pulses. Halscope has 16 channels, but for this example we are using only 4 at a time. Before we select any more channels, we need to turn off a couple. Click on the channel 2 button, then click the "Off" button at the bottom of the "Vertical" box. Then click on channel 3, turn if off, and do the same for channel 4. Even though the channels are turned off, they still remember what they are connected to, and in fact we will continue to use channel 3 as the trigger source. To add new channels, select channel 5, and choose pin "stepgen.1.dir", then channel 6, and select "stepgen.1.step". Then click run mode "Normal" to start the scope, and adjust the horizontal zoom to 5mS per division. You should see the step pulses slow down as the velocity command (channel 1) approaches zero, then the direction pin changes state and the step pulses speed up again. You might want to increase the gain on channel 1 to about 20m per division to better see the change in the velocity command. The result should look like figure 2.11.

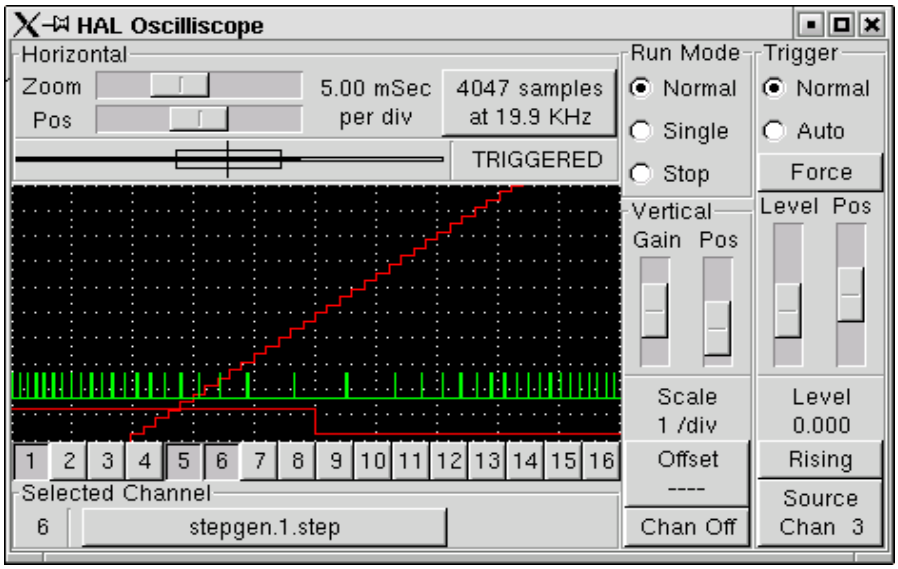


Figure 2.11: Looking at Step Pulses

Part II

HAL Reference

Chapter 3

General Reference Information

3.1 Notation

3.1.1 Typographical Conventions

Command line examples are presented in **`bold typewriter`** font. Responses from the computer will be in `typewriter` font. As of early 2006, there are no longer commands that require root privileges, so all examples will be preceded by the normal user prompt, `$`. Text inside square brackets `[like-this]` is optional. Text inside angle brackets `<like-this>` represents a field that can take on different values, and the adjacent paragraph will explain the appropriate values. Text items separated by a vertical bar means that one or the other, but not both, should be present. All command line examples assume that you are in the `emc2/` directory, and you configured/compiled emc2 for the run-in-place scenario. Paths will be shown accordingly when needed.

3.1.2 Names

All HAL entities are accessed and manipulated by their names, so documenting the names of pins, signals, parameters, etc, is very important. HAL names are a maximum of 41 characters long (as defined by HAL_NAME_LEN in hal.h). Many names will be presented in a general form, with text inside angle brackets `<like-this>` representing fields that can take on different values.

When pins, signals, or parameters are described for the first time, their names will be preceeded by their type in (SMALL CAPS) and followed by a brief description. A typical pin definition will look something like these examples:

- `(BIT) parport.<portnum>.pin-<pinnum>-in` – The HAL pin associated with the physical input pin `<pinnum>` on the 25 pin D-shell connector.
- `(FLOAT) pid.<loopnum>.output` – The output of the PID loop.

At times, a shortened version of a name may be used - for example the second pin above might be referred to simply as `.output` when it can be done without causing confusion.

3.2 General Naming Conventions

Consistent naming conventions would make HAL much easier to use. For example, if every encoder driver provided the same set of pins and named them the same way it would be easy to change from one type of encoder driver to another. Unfortunately, like many open-source projects, HAL is a combination of things that were designed, and things that simply evolved. As a result, there are many inconsistencies. This section attempts to address that problem by defining some conventions, but it will probably be a while before all the modules are converted to follow them.

Halcmd and other low-level HAL utilities treat HAL names as single entities, with no internal structure. However, most modules do have some implicit structure. For example, a board provides several functional blocks, each block might have several channels, and each channel has one or more pins. This results in a structure that resembles a directory tree. Even though halcmd doesn't recognize the tree structure, proper choice of naming conventions will let it group related items together (since it sorts the names). In addition, higher level tools can be designed to recognize such structure, if the names provide the neccessary information. To do that, all HAL modules should follow these rules:

- Dots (".") separate levels of the heirarchy. This is analogous to the slash ("/") in a filename.
- Hypens ("-") separate words or fields in the same level of the heirarchy.
- HAL modules should not use underscores or "MixedCase". [1]
- Use only lowercase letters and numbers in names.

3.3 Hardware Driver Naming Conventions[2]

3.3.1 Pin/Parameter names

Hardware drivers should use five fields (on three levels) to make up a pin or parameter name, as follows:

```
<device-name>.<device-num>.<io-type>.<chan-num>.<specific-name>
```

The individual fields are:

<device-name> The device that the driver is intended to work with. This is most often an interface board of some type, but there are other possibilities.

<device-num> It is possible to install more than one servo board, parallel port, or other hardware device in a computer. The device number identifies a specific device. Device numbers start at 0 and increment.[3]

<io-type> Most devices provide more than one type of I/O. Even the simple parallel port has both digital inputs and digital outputs. More complex boards can have digital inputs and outputs, encoder counters, pwm or step pulse generators, analog-to-digital converters, digital-to-analog converters, or other unique capabilities. The I/O type is used to identify the kind of I/O that a pin or parameter is associated with. Ideally, drivers that implement the same I/O type, even if for very different devices, should provide a consistent set of pins and parameters and identical behavior. For example, all digital inputs should behave the same when seen from inside the HAL, regardless of the device.

<chan-num> Virtually every I/O device has multiple channels, and the channel number identifies one of them. Like device numbers, channel numbers start at zero and increment.[4] If more than one device is installed, the channel numbers on additional devices start over at zero. If it is possible to have a channel number greater than 9, then channel numbers should be two digits,

[1] Underscores have all been removed, but there are still a few instances of mixed case, for example "pid.0.Pgain" instead of "pid.0.p-gain".

[2] Most drivers do not follow these conventions as of version 2.0. This chapter is really a guide for future development.

[3] Some devices use jumpers or other hardware to attach a specific ID to each board. Ideally, the driver provides a way for the user to specifically say "device-num 0 is the board with ID XXX", and the device numbers always start at 0. However at present some drivers use the board ID directly as the device number. That means it is possible to have a device number 2, without a device 0. This is a bug and will be fixed in version 2.1.

[4] One glaring exception to the "channel numbers start at zero" rule is the parallel port. Its HAL pins are numbered with the corresponding pin number on the DB-25 connector. This is convenient for wiring, but inconsistent with other drivers. There is some debate over whether this is a bug or a feature.

with a leading zero on numbers less than 10 to preserve sort ordering. Some modules have pins and/or parameters that affect more than one channel. For example a PWM generator might have four channels with four independent "duty-cycle" inputs, but one "frequency" parameter that controls all four channels (due to hardware limitations). The frequency parameter should use "0-3" as the channel number.

<specific-name> An individual I/O channel might have just a single HAL pin associated with it, but most have more than one. For example, a digital input has two pins, one is the state of the physical pin, the other is the same thing inverted. That allows the configurator to choose between active high and active low inputs. For most io-types, there is a standard set of pins and parameters, (referred to as the "canonical interface") that the driver should implement. The canonical interfaces are described in chapter 4.

3.3.1.1 Examples

motenc.0.encoder.2.position – the position output of the third encoder channel on the first Motenc board.

stg.0.din.03.in – the state of the fourth digital input on the first Servo-to-Go board.

ppmc.0.pwm.00-03.frequency – the carrier frequency used for PWM channels 0 through 3.

3.3.2 Function Names

Hardware drivers usually only have two kinds of HAL functions, ones that read the hardware and update HAL pins, and ones that write to the hardware using data from HAL pins. They should be named as follows:

<device-name>-<device-num>[.<io-type>[-<chan-num-range>]].read|write

<device-name> The same as used for pins and parameters.

<device-num> The specific device that the function will access.

<io-type> Optional. A function may access all of the I/O on a board, or it may access only a certain type. For example, there may be independent functions for reading encoder counters and reading digital I/O. If such independent functions exist, the <io-type> field identifies the type of I/O they access. If a single function reads all I/O provided by the board, <io-type> is not used.[5]

<chan-num-range> Optional. Used only if the <io-type> I/O is broken into groups and accessed by different functions.

read|write Indicates whether the function reads the hardware or writes to it.

3.3.2.1 Examples

motenc.0.encoder.read – reads all encoders on the first motenc board

generic8255.0.din.09-15.read – reads the second 8 bit port on the first generic 8255 based digital I/O board

ppmc.0.write – writes all outputs (step generators, pwm, DACs, and digital) on the first ppmc board

[5]Note to driver programmers: do NOT implement separate functions for different I/O types unless they are interruptable and can work in independent threads. If interrupting an encoder read, reading digital inputs, and then resuming the encoder read will cause problems, then implement a single function that does everything.

Chapter 4

Canonical Device Interfaces[1]

The following sections show the pins, parameters, and functions that are supplied by "canonical devices". All HAL device drivers should supply the same pins and parameters, and implement the same behavior.

Note that the only the `<io-type>` and `<specific-name>` fields are defined for a canonical device. The `<device-name>`, `<device-num>`, and `<chan-num>` fields are set based on the characteristics of the real device.

4.1 Digital Input

The canonical digital input (I/O type field: **digin**) is quite simple.

4.1.1 Pins

- (BIT) **in** – State of the hardware input.
- (BIT) **in-not** – Inverted state of the input.

4.1.2 Parameters

- None

4.1.3 Functions

- (FUNCT) **read** – Read hardware and set **in** and **in-not** HAL pins.

4.2 Digital Output

The canonical digital output (I/O type field: **digout**) is also very simple.

4.2.1 Pins

- (BIT) **out** – Value to be written (possibly inverted) to the hardware output.

[1] As of version 2.0, most of the HAL drivers don't quite match up to the canonical interfaces defined here. In version 2.1, the drivers will be changed to match these specs.

4.2.2 Parameters

- (BIT) **`invert`** – If TRUE, **`out`** is inverted before writing to the hardware.

4.2.3 Functions

- (FUNCT) **`write`** – Read **`out`** and **`invert`**, and set hardware output accordingly.

4.3 Analog Input

The canonical analog input (I/O type: **`adcin`**). This is expected to be used for analog to digital converters, which convert e.g. voltage to a continuous range of values.

4.3.1 Pins

- (FLOAT) **value** – The hardware reading, scaled according to the **scale** and **offset** parameters. **Value** = ((input reading, in hardware-dependent units) * **scale**) - **offset**

4.3.2 Parameters

- (FLOAT) **scale** – The input voltage (or current) will be multiplied by **scale** before being output to **value**.
- (FLOAT) **offset** – This will be subtracted from the hardware input voltage (or current) after the scale multiplier has been applied.
- (FLOAT) **bit_weight** – The value of one least significant bit (LSB). This is effectively the granularity of the input reading.
- (FLOAT) **hw_offset** – The value present on the input when 0 volts is applied to the input pin(s).

4.3.3 Functions

- (FUNCT) **read** – Read the values of this analog input channel. This may be used for individual channel reads, or it may cause all channels to be read

4.4 Analog Output

The canonical analog output (I/O Type: **`adcout`**). This is intended for any kind of hardware that can output a more-or-less continuous range of values. Examples are digital to analog converters or PWM generators.

Pins

- (FLOAT) **value** – The value to be written. The actual value output to the hardware will depend on the scale and offset parameters.
- (BIT) **enable** – If false, then output 0 to the hardware, regardless of the **value** pin.

4.4.1 Parameters

- (FLOAT) **offset** – This will be added to the **value** before the hardware is updated
- (FLOAT) **scale** – This should be set so that an input of 1 on the **value** pin will cause 1V
- (FLOAT) **high_limit** (optional) – When calculating the value to output to the hardware, if **value** + **offset** is greater than **high_limit**, then **high_limit** will be used instead.
- (FLOAT) **low_limit** (optional) – When calculating the value to output to the hardware, if **value** + **offset** is less than **low_limit**, then **low_limit** will be used instead.
- (FLOAT) **bit_weight** (optional) – The value of one least significant bit (LSB), in volts (or mA, for current outputs)
- (FLOAT) **hw_offset** (optional) – The actual voltage (or current) that will be output if 0 is written to the hardware.

4.4.2 Functions

(FUNCT) **write** – This causes the calculated value to be output to the hardware. If enable is false, then the output will be 0, regardles of **value**, **scale**, and **offset**. The meaning of "0" is dependent on the hardware. For example, a bipolar 12-bit A/D may need to write 0x1FF (mid scale) to the D/A get 0 volts from the hardware pin. If enable is true, read scale, offset and value and output to the adc (**scale** * **value**) + **offset**. If enable is false, then output 0.

4.5 Encoder

The canonical encoder interface (I/O type field: **`encoder`**) provides the functionality needed for homing to an index pulse and doing spindle synchronization, as well as basic position and/or velocity control. This interface should be implementable regardless of the actual underlying hardware, although some hardware will provide "better" results. (For example, capture the index position to +/- 1 count while moving faster, or have less jitter on the velocity pin.)

4.5.1 Pins

- (S32) **`count`** – Encoder value in counts.
- (FLOAT) **`position`** – Encoder value in position units (see parameter "scale").
- (FLOAT) **`velocity`** – Velocity in position units per second.
- (BIT) **`reset`** – When True, force counter to zero.
- (BIT) **`index-enable`** – (bidirectional) When True, reset to zero on next index pulse, and set pin False.

The "index-enable" pin is bi-directional, and might require a little more explanation. If "index-enable" is False, the index channel of the encoder will be ignored, and the counter will count normally. The encoder driver will never set "index-enable" True. However, some other component may do so. If "index-enable" is True, then when the next index pulse arrives, the encoder counter will be reset to zero, and the driver will set "index-enable" False. That will let the other component know that an index pulse arrived. This is a form of handshaking - the other component sets "index-enable" True to request a index pulse reset, and the driver sets it False when the request has been satisfied.

4.5.2 Parameters

- (FLOAT) **scale** – The scale factor used to convert counts to position units. It is in "counts per position unit". For example, if you have a 512 count per turn encoder on a 5 turn per inch screw, the scale should be 512*5 = 2560 counts per inch, which will result in "position" in inches and "velocity" in inches per second.
- (FLOAT) **max-index-vel** – (optional) The maximum velocity (in position units per second) at which the encoder can reset on an index pulse with +/- 1 count accuracy. This is an output from the encoder driver, and is intended to tell the user something about the hardware capabilities. Some hardware can reset the counter at the exact moment the index pulse arrives. Other hardware can only tell that an index pulse arrived sometime since the last time the read function was called. For the latter, +/- 1 count accuracy can only be achieved if the encoder advances by 1 count or less between calls to the read function.
- (FLOAT) **velocity-resolution** – (optional) The resolution of the velocity output, in position units per second. This is an output from the encoder driver, and is intended to tell the user something about the hardware capabilities. The simplest implementation of the velocity output is the change in postion from one call of the read function to the next, divided by the time between calls. This yields a rather coarse velocity signal that jitters back and forth between widely spaced possible values (quantization error). However, some hardware captures both the counts and the exact time when a count occurres (possibly with a very high resolution clock). That data allows the driver to calculate velocity with finer resolution and less jitter.

4.5.3 Functions

There is only one function, to read the encoder(s).

- (FUNCT) **read** – Capture counts, update position and velocity.

Chapter 5

Tools and Utilities

5.1 Halcmd

Halcmd is a command line tool for manipulating the HAL. There is a rather complete man page for halcmd, which will be installed if you have installed EMC2 from either source or a package. If you have compiled EMC2 for "run-in-place", the man page is not installed, but it is accessible. From the main EMC2 directory, do:

```
$ man -M docs/man halcmd
```

Chapter 2 has a number of examples of halcmd usage, and is a good tutorial for halcmd.

5.2 Halmeter

Halmeter is a "voltmeter" for the HAL. It lets you look at a pin, signal, or parameter, and displays the current value of that item. It is pretty simple to use. Start it by typing "`halmeter`" in a X windows shell. Halmeter is a GUI application. It will pop up a small window, with two buttons labeled "Select" and "Exit". Exit is easy - it shuts down the program. Select pops up a larger window, with three tabs. One tab lists all the pins currently defined in the HAL. The next lists all the signals, and the last tab lists all the parameters. Click on a tab, then click on a pin/signal/parameter. Then click on "OK". The lists will disappear, and the small window will display the name and value of the selected item. The display is updated approximately 10 times per second. If you click "Accept" instead of "OK", the small window will display the name and value of the selected item, but the large window will remain on the screen. This is convenient if you want to look at a number of different items quickly.

You can have many halmeters running at the same time, if you want to monitor several items. If you want to launch a halmeter without tying up a shell window, type "`halmeter &`" to run it in the background. You can also make halmeter start displaying a specific item immediately, by adding "`pin|sig|par[am] <name>`" to the command line. It will display the pin, signal, or parameter <name> as soon as it starts. (If there is no such item, it will simply start normally.) And finally, if you specify an item to display, you can add "-s" before the pin|sig|param to tell halmeter to use a small window. The item name will be displayed in the title bar instead of under the value, and there will be no buttons. Usefull when you want a lot of meters in a small amount of screen space.

5.3 Halscope

Halscope is an "oscilloscope" for the HAL. It lets you capture the value of pins, signals, and parameters as a function of time. Complete operating instructions should be located here eventually. For now, refer to section 2.6 in the tutorial chapter, which explains the basics.

5.4 Halshow

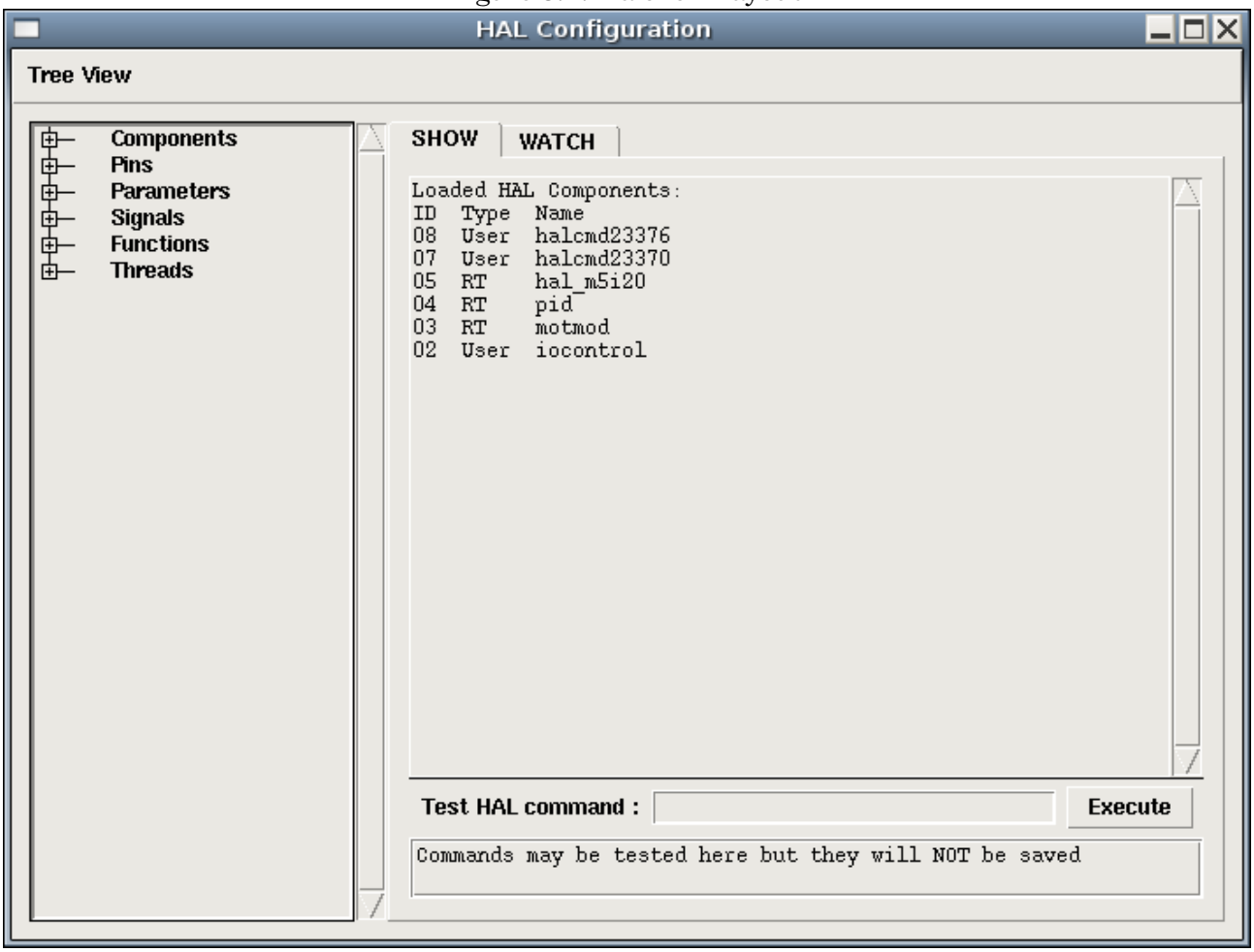
The script halshow can help you find your away around a running HAL. This is a very specialized system and it must connect to a working HAL. It can not run stand alone because it relies on the ability of HAL to report what it knows of itself through the halcmd interface library. It is discovery based. Each time halshow runs with a different EMC configuration it will be different.

As we will soon see, this ability of HAL to document itself is one key to making an effective CNC system.

5.4.1 Hal Tree Area

At the left of its display as shown in figure 5.1 is a tree view, somewhat like you might see with some file browsers. At the right is a tabbed notebook with tabs for show and watch.

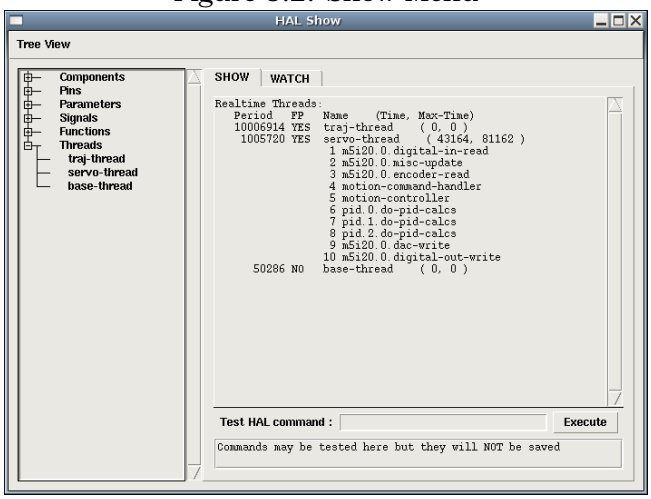
Figure 5.1: Halshow Layout



The tree shows all of the major parts of a HAL. In front of each is a small plus (+) or minus (-) sign in a box. Clicking the plus will expand that tree node to display what is under it. If that box shows a minus sign clicking it will close that section of the tree.

You can also expand or collapse the tree display using the Tree View menu at the upper left edge of the display. This menu is shown in figure xx

Figure 5.2: Show Menu



5.4.2 Hal Show Area

Clicking on the node name rather than its plus or minus sign, the word "Components" for example, will show you all that hal knows about the contents of it. Figure 5.1 shows a list exactly like you will see if you click the "Components" name while you are running a standard m5i20 servo card. The information display is exactly like those shown in traditional text based HAL analysis tools. The advantage here is that we have mouse click access. Access that can be as broad or as focused as you need.

If we take a closer look at the tree display we can see that the six major parts of a HAL can all be expanded at least one level. As these levels are expanded you can get more focused with the reply when you click on the rightmost tree node. You will find that there are some hal pins and parameters that show more than one reply. This is do to the nature of the search routines in halcmd itself. If you search one pin you may get two like this.

```
Component Pins:
Owner Type Dir Value Name
06 bit -W TRUE parport.0.pin-10-in
06 bit -W FALSE parport.0.pin-10-in-not
```

The second pins name contains the complete name of the first.

Below the show area on the right is a set of widgets that will allow you to play with the running HAL. The commands you enter here and the effect that they have on the running HAL are not saved. They will persist as long as the emc remains up but are gone as soon as it is.

The entry box labeled Test Hal Command : will accept any of the commands listed for halcmd. These include;

- loadrt, unloadrt
- addf, delf
- newsig, delsig
- linkpp, linksp, linkps, unlinkp
- setp, sets

This little editor will enter a command any time you press <enter> or push the execute button. An error message from halcmd will show below this entry widget when these commands are not

properly formed. If you are not certain how to set up a proper command you'll need to read again the documentation on halcmd and the specific modules that you are working with.

Let's use this editor to add a differential module to a hal and connect it to axis position so that we could see the rate of change in position, ie acceleration. We first need to load a hal module named blocks, add it to the servo thread, then connect it to the position pin of an axis. Once that is done we can find the output of the differentiator in halscope. So let's go. (yes I looked this one up.)

```
loadrt blocks ddt=1
```

Now look at the components node and you should see blocks in there someplace.

```
Loaded HAL Components:
ID Type Name
10 User halcmd29800
09 User halcmd29374
08 RT blocks
06 RT hal_parport
05 RT scope_rt
04 RT stepgen
03 RT motmod
02 User iocontrol
```

Sure enough there it is. Notice that its id is 08. Next we need to find out what functions are available with it so we look at functions.

```
Exported Functions:
Owner CodeAddr Arg FP Users Name
08 E0B97630 E0DC7674 YES 0 ddt.0
03 E0DEF83C 00000000 YES 1 motion-command-handler
03 E0DF0BF3 00000000 YES 1 motion-controller
06 E0B541FE E0DC75B8 NO 1 parport.0.read
06 E0B54270 E0DC75B8 NO 1 parport.0.write
06 E0B54309 E0DC75B8 NO 0 parport.read-all
06 E0B5433A E0DC75B8 NO 0 parport.write-all
05 E0AD712D 00000000 NO 0 scope.sample
04 E0B618C1 E0DC7448 YES 1 stepgen.capture-position
04 E0B612F5 E0DC7448 NO 1 stepgen.make-pulses
04 E0B614AD E0DC7448 YES 1 stepgen.update-freq
```

Here we look for owner #08 and see that blocks has exported a function named ddt.0. We should be able to add ddt.0 to the servo thread and it will do its math each time the servo thread is updated. Once again I look up the addf command and find that it uses three arguments like this.

addf <functname> <threadname> [<position>]

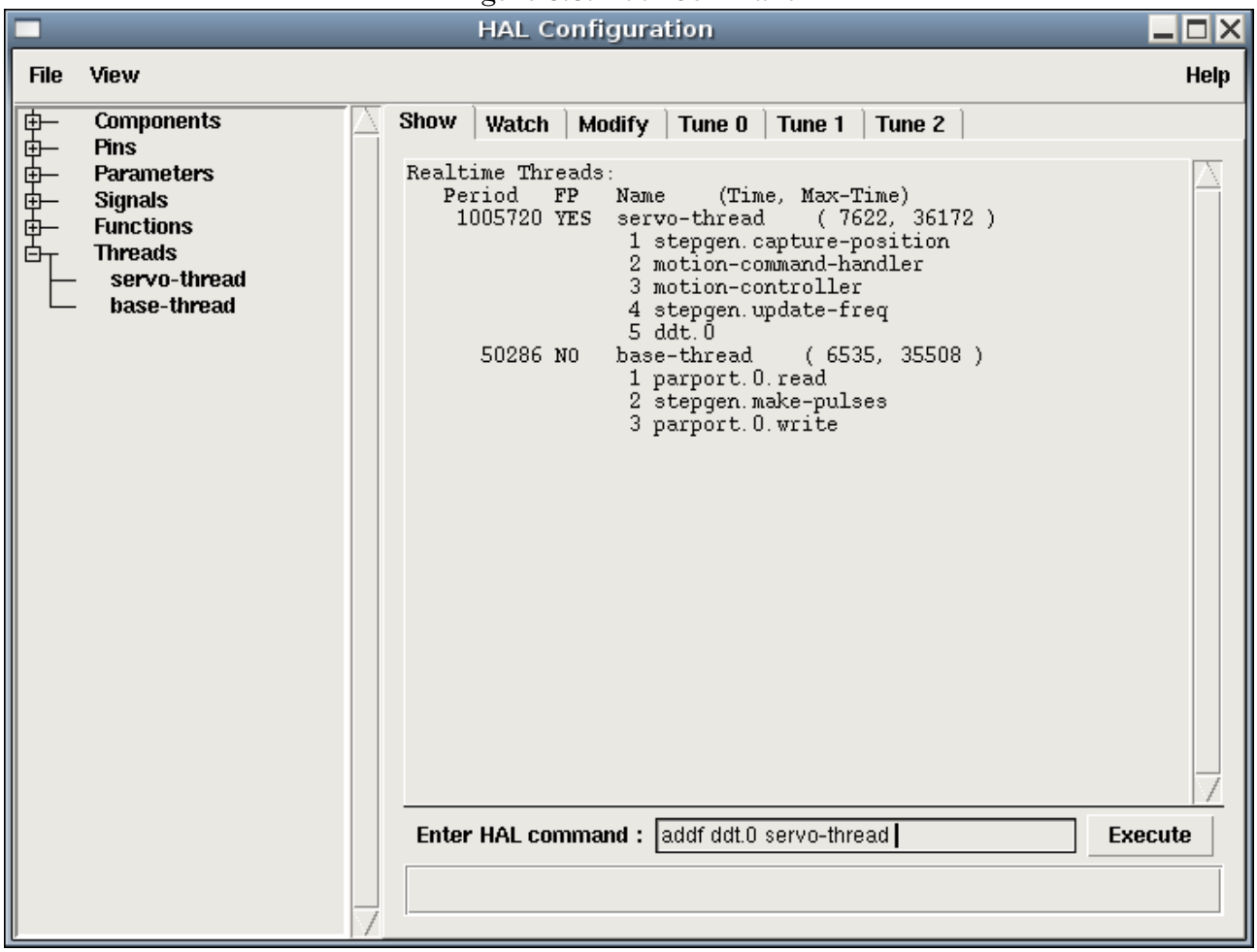
We already know the functname=ddt.0 so let's get the thread name right by expanding the thread node in the tree. Here we see two threads servo-thread and base-thread. The position of ddt.0 in the thread is not critical.

```
addf ddt.0 servo-thread
```

This is just for viewing so we leave position blank and get the last position in the thread. Figure 5.3 shows the state of halshow after this command has been issued.

Next we need to connect this block to something. But how do we know what pins are available. The answer is look under pins. There we find ddt and see this.

Figure 5.3: Addf Command



```
Component Pins:
Owner Type Dir Value Name
08 float R- 0.00000e+00 ddt.0.in
08 float -W 0.00000e+00 ddt.0.out
```

That looks easy enough to understand but what signal or pin do we want to connect to it. It could be an axis pin, a stepgen pin, or a signal. I see this when I look at axis.0.

```
Component Pins:
Owner Type Dir Value Name
03 float -W 0.00000e+00 axis.0.motor-pos-cmd ==> Xpos-cmd
```

So it looks like Xpos-cmd should be a good signal to use. Back to the editor where I enter the following command.

```
linksp Xpos-cmd ddt.0.in
```

Now if I look at the Xpos-cmd signal using the tree node I'll see what I've done.

```
Signals:
Type Value Name
float 0.00000e+00 Xpos-cmd
<== axis.0.motor-pos-cmd
==> ddt.0.in
==> stepgen.0.position-cmd
```

We see that this signal comes from axis.o.motor-pos-cmd and goes to both ddt.0.in and stepgen.0.position-cmd. By connecting our block to the signal we have avoided any complications with the normal flow of this motion command.

The Hal Show Area uses halcmd to discover what is happening in a running HAL. It gives you complete information about what it has discovered. It also updates as you issue commands from the little editor panel to modify that HAL. There are times when you want a different set of things displayed without all of the information available in this area. That is where the Hal Watch Area is of value.

5.4.3 Hal Watch Area

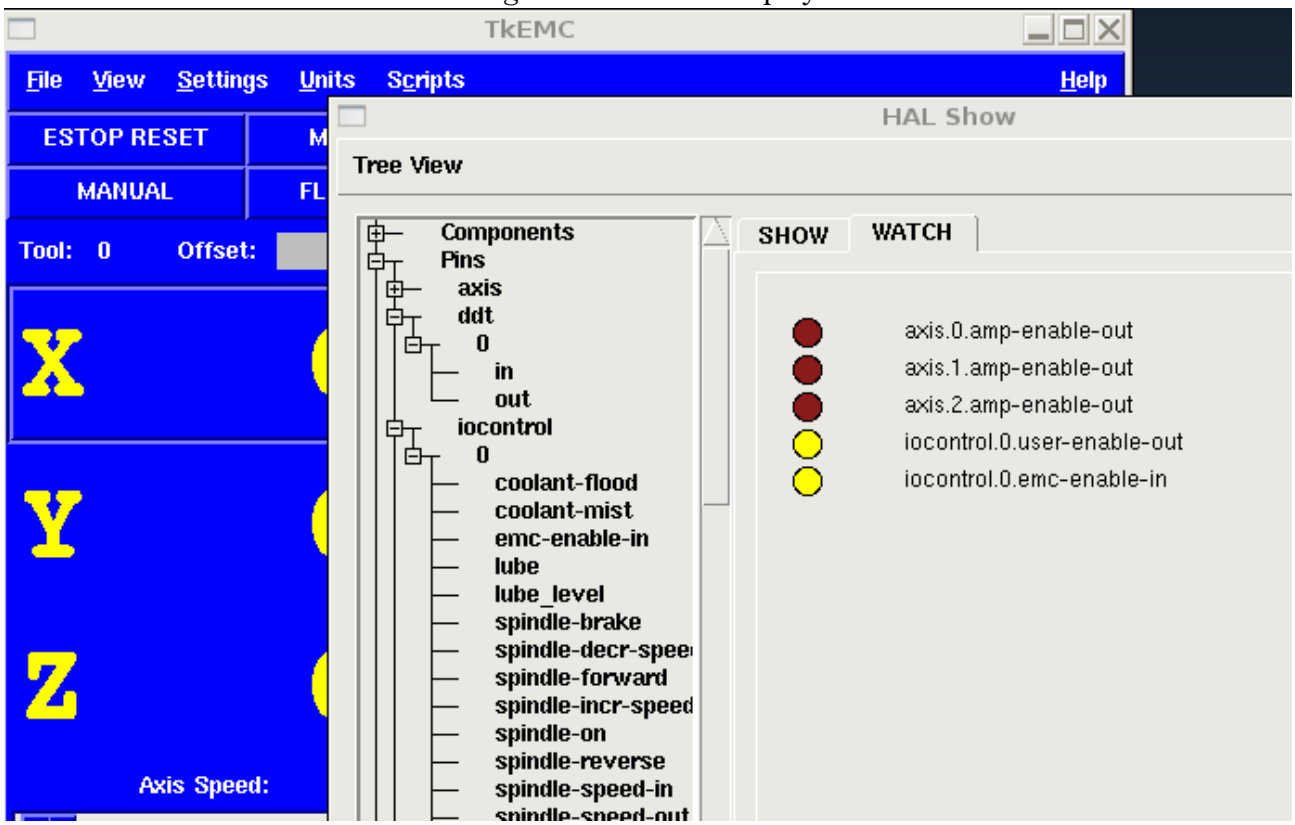
Clicking the watch tab produces a blank canvas. You can add signals and pins to this canvas and watch their values.[1] You can add signals or pins when the watch tab is displayed by clicking on the name of it. Figure 5.4 shows this canvas with several "bit" type signals. These signals include enable-out for the first three axes and two of the three iocontrol "estop" signals. Notice that the axes are not enabled even though the estop signals say that the EMC is not in estop. A quick look at themc shows that the condition of the EMC is ESTOP RESET. The amp enables do not turn true until the machine has been turned on.

The two colors of circles, aka leds, always show dark brown when a bit signal or pin is false. They show the light yellow whenever that signal is true. If you select a pin or signal that is not a bit typed signal, watch will show the numerical value.

Watch will quickly allow you to test switches or see the effect of changes that you make to EMC while using the graphical interface. Watch's refresh rate is a bit slow to see stepper pulses but you can use it for these if you move an axis very slowly or in very small increments of distance. If you've used IO_Show in EMC, the watch page in halshow can be setup to watch a parport much as it did.

[1] The refresh rate of the watch display is much lower than Halmeter or Halscope. If you need good resolution of the timing of signals these tools are much more effective.

Figure 5.4: Watch Display



Chapter 6

Hardware Drivers

6.1 Parport

Parport is a driver for the traditional PC parallel port. The port has a total of 17 physical pins. The original parallel port divided those pins into three groups: data, control, and status. The data group consists of 8 output pins, the control group consists of 4 pins, and the status group consists of 5 input pins.

In the early 1990's, the bidirectional parallel port was introduced, which allows the data group to be used for output or input. The HAL driver supports the bidirectional port, and allows the user to set the data group as either input or output. If configured as output, a port provides a total of 12 outputs and 5 inputs. If configured as input, it provides 4 outputs and 13 inputs.

In some parallel ports, the control group pins are open collectors, which may also be driven low by an external gate. On a board with open collector control pins, the "x" mode allows a more flexible mode with 8 dedicated outputs, 5 dedicated inputs, and 4 open collector pins. In other parallel ports, the control group has push-pull drivers and cannot be used as an input.[1]

No other combinations are supported, and a port cannot be changed from input to output once the driver is installed. Figure 6.1 shows two block diagrams, one showing the driver when the data group is configured for output, and one showing it configured for input.

The parport driver can control up to 8 ports (defined by MAX_PORTS in hal_parport.c). The ports are numbered starting at zero.

6.1.1 Installing

From command line:

```
emc2$ halcmd loadrt hal_parport cfg="<config-string>"
```
[2]

From a file or at the `halcmd:` prompt:

```
loadrt hal_parport cfg="<config-string>"
```

[1] HAL cannot automatically determine if the "x" mode bidirectional pins are actually open collectors (OC). If they are not, they cannot be used as inputs, and attempting to drive them LOW from an external source can damage the hardware.

To determine whether your port has "open collector" pins, load hal_parport in "x" mode, output a HIGH value on the pin. HAL should read the pin as TRUE. Next, insert a 470Ω resistor from one of the control pins to GND. If the resulting voltage on the control pin is close to 0V, and HAL now reads the pin as FALSE, then you have an OC port. If the resulting voltage is far from 0V, or HAL does not read the pin as FALSE, then your port cannot be used in "x" mode.

The external hardware that drives the control pins should also use open collector gates (e.g., 74LS05). Generally, the -out HAL pins should be set to TRUE when the physical pin is being used as an input.

On some machines, BIOS settings may affect whether "x" mode can be used. "SPP" mode is most most likely to work.

[2] The single quotes around the entire **cfg=** argument are needed to prevent the shell from misinterpreting the double quotes around the string, and any spaces or special characters in the string. Single quotes should not be used in a file or from the halcmd prompt.

The config string consists of a hex port address, followed by an optional direction, repeated for each port. The direction is "in", "out", or "x" and determines the direction of the physical pins 2 through 9, and whether to create input HAL pins for the physical control pins. If the direction is not specified, the data group defaults to output. For example:

```
loadrt hal_parport cfg="0x278 0x378 in 0x20A0 out"
```

This example installs drivers for one port at 0x0278, with pins 2-9 as outputs (by default, since neither "in" nor "out" was specified), one at 0x0378, with pins 2-9 as inputs, and one at 0x20A0, with pins 2-9 explicitly specified as outputs. Note that you must know the base address of the parallel port to properly configure the driver. For ISA bus ports, this is usually not a problem, since the port is almost always at a "well known" address, like 0278 or 0378 which is typically configured in the system BIOS. The address for a PCI card is usally shown in "lspci -v" in an "I/O ports" line. There is no default address; if `<config-string>` does not contain at least one address, it is an error.

6.1.2 Removing

```
emc2$ halcmd unloadrt hal_parport
```

6.1.3 Pins

- `(BIT) parport.<portnum>.pin-<pinnum>-out` – Drives a physical output pin.
- `(BIT) parport.<portnum>.pin-<pinnum>-in` – Tracks a physical input pin.
- `(BIT) parport.<portnum>.pin-<pinnum>-in-not` – Tracks a physical input pin, but inverted.

For each pin, `<portnum>` is the port number, and `<pinnum>` is the physical pin number in the 25 pin D-shell connector.

For each physical output pin, the driver creates a single HAL pin, for example `parport.0.pin-14-out`. Pins 1, 14, 16, and 17 are always outputs. Pins 2 through 9 are part of the data group and are output pins if the port is defined as an output port. (Output is the default.) These HAL pins control the state of the corresponding physical pins.

For each physical input pin, the driver creates two HAL pins, for example `parport.0.pin-12-in` and `parport.0.pin-12-in-not`. Pins 10, 11, 12, 13, and 15 are always input pins. Pins 2 through 9 are input pins only if the port is defined as an input port. The `-in` HAL pin is TRUE if the physical pin is high, and FALSE if the physical pin is low. The `-in-not` HAL pin is inverted – it is FALSE if the physical pin is high. By connecting a signal to one or the other, the user can determine the state of the input.

6.1.4 Parameters

- `(BIT) parport.<portnum>.pin-<pinnum>-out-invert` – Inverts an output pin.

The `-invert` parameter determines whether an output pin is active high or active low. If `-invert` is FALSE, setting the HAL `-out` pin TRUE drives the physical pin high, and FALSE drives it low. If `-invert` is TRUE, then setting the HAL `-out` pin TRUE will drive the physical pin low.

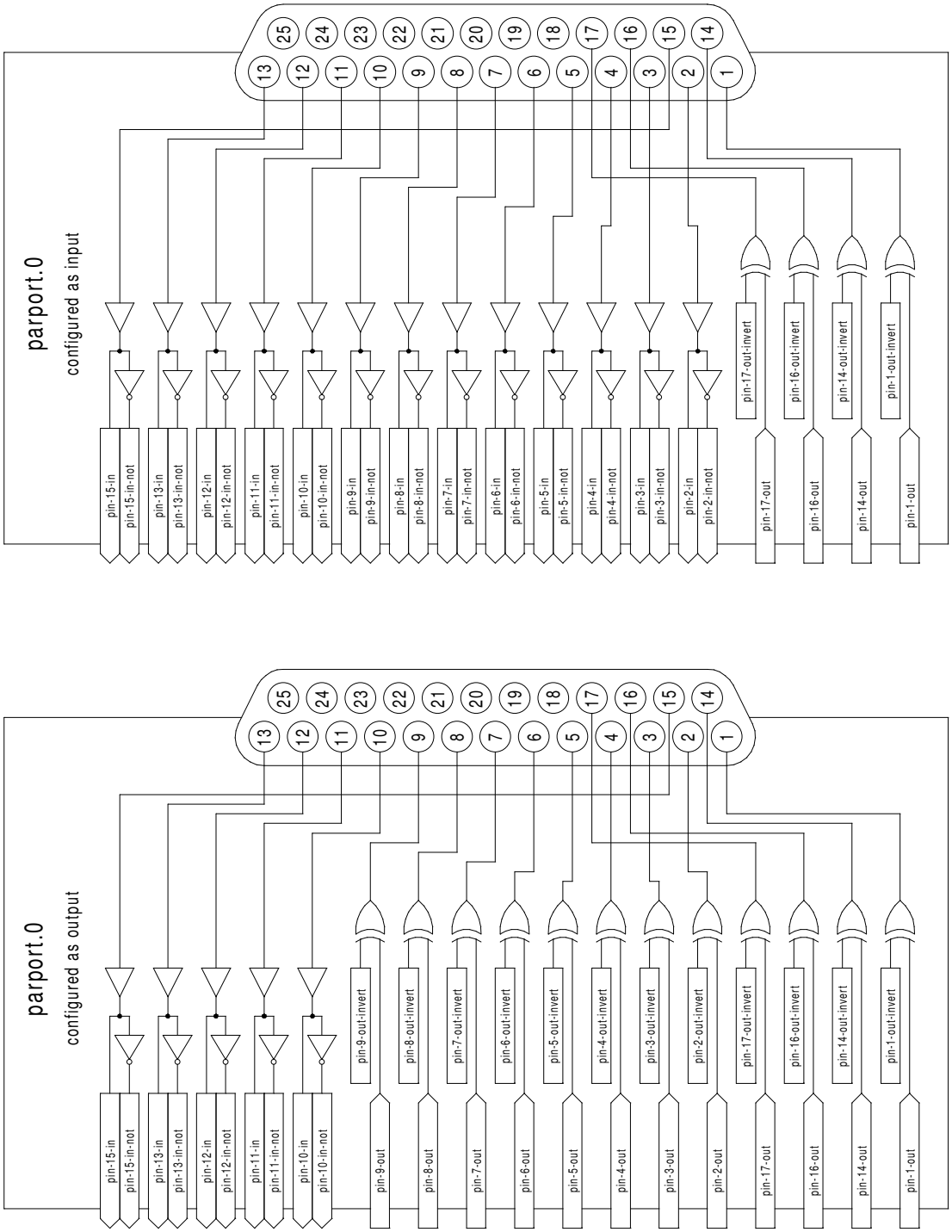


Figure 6.1: Parport Block Diagram

6.1.5 Functions

- (FUNCT) parport.<portnum>.read– Reads physical input pins of port <portnum> and updates HAL -in and -in-not pins.
- (FUNCT) parport.read-all – Reads physical input pins of all ports and updates HAL -in and -in-not pins.
- (FUNCT) parport.<portnum>.write – Reads HAL -out pins of port <portnum> and updates that port's physical output pins.
- (FUNCT) parport.write-all – Reads HAL -out pins of all ports and updates all physical output pins.

The individual functions are provided for situations where one port needs to be updated in a very fast thread, but other ports can be updated in a slower thread to save CPU time. It is probably not a good idea to use both an -all function and an individual function at the same time.

6.1.6 Common problems

If loading the module reports

```
insmod: error inserting '/home/jepler/emc2/rtlib/hal_parport.ko':
-1 Device or resource busy
```

then ensure that the standard kernel module parport_pc is not loaded and that no other device in the system has claimed the I/O ports.

If the module loads but does not appear to function, then the port address is incorrect or the probe_parport module is required.

6.2 probe_parport

In modern PCs, the parallel port may require plug and play (PNP) configuration before it can be used. The probe_parport module performs configuration of any PNP ports present, and should be loaded before hal_parport. On machines without PNP ports, it cannot be loaded.

6.2.1 Installing

```
loadrt probe_parport
loadrt hal_parport ...
```

If the Linux kernel prints a message similar to

```
parport: PnPBIOS parport detected.
```

when the parport_pc module is loaded (sudo modprobe -a parport_pc) then use of this module is probably required

6.3 AX5214H

The Axiom Measurement & Control AX5214H is a 48 channel digital I/O board. It plugs into an ISA bus, and resembles a pair of 8255 chips.[3]

[3] In fact it may be a pair of 8255 chips, but I'm not sure. If/when someone starts a driver for an 8255 they should look at the ax5214 code, much of the work is already done.

6.3.1 Installing

From command line:

```
emc2$ halcmd loadrt hal_ax5214h 'cfg="<config-string>"'
```
[4]

From a file:

```
loadrt hal_ax5214h cfg="<config-string>"
```

The config string consists of a hex port address, followed by an 8 character string of "I" and "O" which sets groups of pins as inputs and outputs. The first two character set the direction of the first two 8 bit blocks of pins (0-7 and 8-15). The next two set blocks of 4 pins (16-19 and 20-23). The pattern then repeats, two more blocks of 8 bits (24-31 and 32-39) and two blocks of 4 bits (40-43 and 44-47). If more than one board is installed, the data for the second board follows the first. As an example, the string **"0x220 IIIOIIOO 0x300 OIOOIOIO"** installs drivers for two boards. The first board is at address 0x220, and has 36 inputs (0-19 and 24-39) and 12 outputs (20-23 and 40-47). The second board is at address 0x300, and has 20 inputs (8-15, 24-31, and 40-43) and 28 outputs (0-7. 16-23, 32-39, and 44-47).

6.3.2 Removing

```
emc2$ halcmd unloadrt hal_ax5214
```

6.3.3 Pins

- (BIT) ax5214.<boardnum>.out-<pinnum> – Drives a physical output pin.
- (BIT) ax5214.<boardnum>.in-<pinnum> – Tracks a physical input pin.
- (BIT) ax5214.<boardnum>.in-<pinnum>-not – Tracks a physical input pin, inverted.

For each pin, <boardnum> is the board number (starts at zero), and <pinnum> is the I/O channel number (0 to 47).

Note that the driver assumes active LOW signals. This is so that modules such as OPTO-22 will work correctly (TRUE means output ON, or input energized). If the signals are being used directly without buffering or isolation the inversion needs to be accounted for. The in- HAL pin is TRUE if the physical pin is low (OPTO-22 module energized), and FALSE if the physical pin is high (OPTO-22 module off). The in-<pinnum>-not HAL pin is inverted – it is FALSE if the physical pin is low (OPTO-22 module energized). By connecting a signal to one or the other, the user can determine the state of the input.

6.3.4 Parameters

- (BIT) ax5214.<boardnum>.out-<pinnum>-invert – Inverts an output pin.

The -invert parameter determines whether an output pin is active high or active low. If -invert is FALSE, setting the HAL out- pin TRUE drives the physical pin low, turning ON an attached OPTO-22 module, and FALSE drives it high, turning OFF the OPTO-22 module. If -invert is TRUE, then setting the HAL out- pin TRUE will drive the physical pin high and turn the module OFF.

[4] The single quotes around the entire **cfg=** argument are needed to prevent the shell from misinterpreting the double quotes around the string, and any spaces or special characters in the string. Single quotes should not be used in a file or from the halcmd prompt.

6.3.5 Functions

- (FUNCT) ax5214.<boardnum>.read – Reads all digital inputs on one board.
- (FUNCT) ax5214.<boardnum>.write – Writes all digital outputs on one board.

6.4 Servo-To-Go

The Servo-To-Go is one of the first PC motion control cards[5] supported by EMC. It is an ISA card and it exists in different flavours (all supported by this driver). The board includes up to 8 channels of quadrature encoder input, 8 channels of analog input and output, 32 bits digital I/O, an interval timer with interrupt and a watchdog.

6.4.1 Installing:

```
emc2$ halcmd loadrt hal_stg [base=<address>] [num_chan=<nr>] \
[dio="<dio-string>"] [model=<model>]
```

The base address field is optional; if it's not provided the driver attempts to autodetect the board. The num_chan field is used to specify the number of channels available on the card, if not used the 8 axis version is assumed. The digital inputs/outputs configuration is determined by a config string passed to insmod when loading the module. The format consists of a four character string that sets the direction of each group of pins. Each character of the direction string is either "I" or "O". The first character sets the direction of port A (Port A - DIO.0-7), the next sets port B (Port B - DIO.8-15), the next sets port C (Port C - DIO.16-23), and the fourth sets port D (Port D - DIO.24-31). The model field can be used in case the driver doesn't autodetect the right card version[6]. For example:

```
emc2$ halcmd loadrt hal_stg base=0x300 num_chan=4 dio="IOIO"
```

This example installs the stg driver for a card found at the base address of 0x300, 4 channels of encoder feedback, DAC's and ADC's, along with 32 bits of I/O configured like this: the first 8 (Port A) configured as Input, the next 8 (Port B) configured as Output, the next 8 (Port C) configured as Input, and the last 8 (Port D) configured as Output

```
emc2$ halcmd loadrt hal_stg
```

This example installs the driver and attempts to autodetect the board address and board model, it installs 8 axes by default along with a standard I/O setup: Port A & B configured as Input, Port C & D configured as Output.

6.4.2 Removing

```
emc2$ halcmd unloadrt hal_stg
```

6.4.3 Pins

- (s32) stg.<channel>.counts – Tracks the counted encoder ticks.
- (FLOAT) stg.<channel>.position – Outputs a converted position.
- (FLOAT) stg.<channel>.dac-value – Drives the voltage for the corresponding DAC.

[5] a motion control card usually is a board containing devices to control one or more axes (the control devices are usually DAC's to set an analog voltage, encoder counting chips for feedback, etc.)

[6] hint: after starting up the driver, 'dmesg' can be consulted for messages relevant to the driver (e.g. autodetected version number and base address)

- (FLOAT) stg.<channel>.adc-value – Tracks the measured voltage from the corresponding ADC.
- (BIT) stg.in-<pinnum> – Tracks a physical input pin.
- (BIT) stg.in-<pinnum>-not – Tracks a physical input pin, but inverted.
- (BIT) stg.out-<pinnum> – Drives a physical output pin

For each pin, <channel> is the axis number, and <pinnum> is the logic pin number of the STG[7].

The in- HAL pin is TRUE if the physical pin is high, and FALSE if the physical pin is low. The in-<pinnum>-not HAL pin is inverted – it is FALSE if the physical pin is high. By connecting a signal to one or the other, the user can determine the state of the input.

6.4.4 Parameters

- (FLOAT) stg.<channel>.position-scale – The number of counts / user unit (to convert from counts to units).
- (FLOAT) stg.<channel>.dac-offset – Sets the offset for the corresponding DAC.
- (FLOAT) stg.<channel>.dac-gain – Sets the gain of the corresponding DAC.
- (FLOAT) stg.<channel>.adc-offset – Sets the offset of the corresponding ADC.
- (FLOAT) stg.<channel>.adc-gain – Sets the gain of the corresponding ADC.
- (BIT) stg.out-<pinnum>-invert – Inverts an output pin.

The -invert parameter determines whether an output pin is active high or active low. If -invert is FALSE, setting the HAL out- pin TRUE drives the physical pin high, and FALSE drives it low. If -invert is TRUE, then setting the HAL out- pin TRUE will drive the physical pin low.

6.4.5 Functions

- (FUNCT) stg.capture-position – Reads the encoder counters from the axis <channel>.
- (FUNCT) stg.write-dacs – Writes the voltages to the DACs.
- (FUNCT) stg.read-adcs – Reads the voltages from the ADCs.
- (FUNCT) stg.di-read – Reads physical in- pins of all ports and updates all HAL in- and in-<pinnum>-not pins.
- (FUNCT) stg.do-write – Reads all HAL out- pins and updates all physical output pins.

6.5 Mesa Electronics m5i20 "Anything I/O Card"

The Mesa Electronics m5i20 card consists of an FPGA that can be loaded with a wide variety of configurations, and has 72 pins that leave the PC. The assignment of the pins depends on the FPGA configuration. Currently there is a HAL driver for the "4 axis host based motion control" configuration, and this FPGA configurations is also provided with EMC2. It provides 8 encoder counters, 4 PWM outputs (normally used as DACs) and up to 48 digital I/O channels, 32 inputs and 16 outputs.[8]

Installing:

[7] if IIOO is defined, there are 16 input pins (in-00 .. in-15) and 16 output pins (out-00 .. out-15), and they correspond to PORTs ABCD (in-00 is PORTA.0, out-15 is PORTD.7)

[8] Ideally the encoders, "DACs", and digital I/O would comply with the canonical interfaces defined earlier, but they don't. Fixing that is on the things-to-do list.

```
emc2$ halcmd loadrt hal_m5i20 [loadFpga=1|0] [dacRate=<rate>]
```

If **loadFpga** is 1 (the default) the driver will load the FPGA configuration on startup. If it is 0, the driver assumes the configuration is already loaded. **dacRate** sets the carrier frequency for the PWM outputs, in Hz. The default is 32000, for 32KHz PWM.[9] The driver prints some usefull debugging message to the kernel log, which can be viewed with **dmesg**.

6.5.1 Removing

```
emc2$ halcmd unloadrt hal_m5i20
```

6.5.2 Pins

In the following pins, parameters, and functions, <board> is the board ID. According to the naming conventions the first board should always have an ID of zero, however this driver uses the PCI board ID, so it may be non-zero even if there is only one board. The driver attempts to emulate the "canonical encoder", however there are some differences.[10]

- (S32) m5i20.<board>.enc-<channel>-count – Encoder position, in counts.
- (FLOAT) m5i20.<board>.enc-<channel>-position – Encoder position, in user units.
- (BIT) m5i20.<board>.enc-<channel>-index – Current status of index pulse input
- (BIT) m5i20.<board>.enc-<channel>-index-enable – Bidirectional - if TRUE, counter will reset on next index pulse, and index-enable will be cleared
- (BIT) m5i20.<board>.enc-<channel>-reset-count – Bidirectional - if TRUE, counter will reset immediately, and reset-count will be cleared.
- (BIT) m5i20.<board>.dac-<channel>-enable – Enables DAC if true. DAC outputs zero volts if false?
- (FLOAT) m5i20.<board>.dac-<channel>-value – Analog output value for PWM "DAC" (in user units, see -scale and -offset)
- (BIT) m5i20.<board>.in-<channel> – State of digital input pin, see canonical digital input.
- (BIT) m5i20.<board>.in-<channel>-not – Inverted state of digital input pin, see canonical digital input.
- (BIT) m5i20.<board>.out-<channel> – Value to be written to digital output, see canonical digital output.
- (BIT) m5i20.<board>.estop-in – Dedicated estop input, more details needed.
- (BIT) m5i20.<board>.estop-in-not – Inverted state of dedicated estop input.
- (BIT) m5i20.<board>.watchdog-reset – Bidirectional, - Set TRUE to reset watchdog once, is automatically cleared. If bit value 16 is set in watchdog-control then this value is not used, and the hardware watchdog is cleared every time the dac-write function is executed.

[9] I don't know what the maximum (and minimum, if any) PWM frequency is, it should be documented here. Also, this is the kind of thing that ideally is controlled by a HAL parameter, rather than being set when the driver is initially loaded. I don't know if that is possible, it depends on the hardware and I don't have the neccessary information.

[10] Versions 2.1.6 and earlier had a significantly different encoder interface, and had bugs in the indexing code. Normal motion control worked, but homing to index, threading, and rigid tapping did not, since all of those features require the index pulse to work properly. This documentation is for versions 2.1.7 and later.

6.5.3 Parameters

- (FLOAT) m5i20.<board>.enc-<channel>-scale – The number of counts / user unit (to convert from counts to units).
- (FLOAT) m5i20.<board>.dac-<channel>-offset – Sets the DAC offset.
- (FLOAT) m5i20.<board>.dac-<channel>-gain – Sets the DAC gain (scaling).
- (BIT) m5i20.<board>.dac-<channel>-interlaced – Sets the DAC to interlaced mode. Use this mode if you are filtering the PWM to generate an anaolg voltage.[11]
- (BIT) m5i20.<board>.out-<channel>-invert – Inverts a digital output, see canonical digital output.
- (U32) m5i20.<board>.watchdog-control – Configures the watchdog. The value may be a bitwise OR of the following values:

Bit #	Value	Meaning
0	1	Watchdog is enabled
1	2	Watchdog is automatically reset by DAC writes (the HAL dac-write function)

  Typically, the useful values are 0 (watchdog disabled) or 3 (watchdog enabled, cleared by dac-write).
- (U32) m5i20.<board>.led-view – Maps some of the I/O to onboard LEDs. See table below.

6.5.4 Functions

- (FUNCT) m5i20.<board>.encoder-read – Reads all encoder counters.
- (FUNCT) m5i20.<board>.digital-in-read – Reads digital inputs.
- (FUNCT) m5i20.<board>.dac-write – Writes the voltages (PWM duty cycles) to the "DACs".
- (FUNCT) m5i20.<board>.digital-out-write – Writes digital outputs.
- (FUNCT) m5i20.<board>.misc-update – Writes watchdog timer configuration to hardware. Resets watchdog timer. Updates E-stop pin (more info needed). Updates onboard LEDs.

6.5.5 Connector pinout

The Hostmot-4 FPGA configuration has the following pinout. There are three 50-pin ribbon cable connectors on the card: P2, P3, and P4. There are also 8 status LEDs.

[11] With normal 10 bit PWM, 50% duty cycle would be 512 cycles on and 512 cycles off = ca 30 kHz with 33 MHz reference counter. With fully interleaved PWM this would be 1 cycle on, 1 cycle off for 1024 cycles (16.66 MHz if the PWM reference counter runs at 33 MHz) = much easier to filter. The 5I20 configuration interlace is somewhat between non and fully interlaced (to make it easy to filter but not have as many transistions as fully interleaved).

6.5.5.1 Connecor P2

m5i20 card connector P2	Function/HAL-pin
1	enc-01 A input
3	enc-01 B input
5	enc-00 A input
7	enc-00 B input
9	enc-01 index input
11	enc-00 index input
13	dac-01 output
15	dac-00 output
17	DIR output for dac-01
19	DIR output for dac-00
21	dac-01-enable output
23	dac-00-enable output
25	enc-03 B input
27	enc-03 A input
29	enc-02 B input
31	enc-02 A input
33	enc-03 index input
35	enc-02 index input
37	dac-03 output
39	dac-02 output
41	DIR output for dac-03
43	DIR output for dac-02
45	dac-03-enable output
47	dac-02-enable output
49	Power +5 V (or +3.3V ?)
all even pins	Ground

6.5.5.2 Connector P3

Encoder counters 4 - 7 work simultaneously with in-00 to in-11.

If you are using in-00 to in-11 as general purpose IO then reading enc-<4-7> will produce some random junk number.

m5i20 card connector P3	Function/HAL-pin	Secondary Function/HAL-pin
1	in-00	enc-04 A input
3	in-01	enc-04 B input
5	in-02	enc-04 index input
7	in-03	enc-05 A input
9	in-04	enc-05 B input
11	in-05	enc-05 index input
13	in-06	enc-06 A input
15	in-07	enc-06 B input
17	in-08	enc-06 index input
19	in-09	enc-07 A input
21	in-10	enc-07 B input
23	in-11	enc-07 index input
25	in-12	
27	in-13	
29	in-14	
31	in-15	
33	out-00	
35	out-01	
37	out-02	
39	out-03	
41	out-04	
43	out-05	
45	out-06	
47	out-07	
49	Power +5 V (or +3.3V ?)	
all even pins	Ground	

Note!: This is the intended pinout of P3. Unfortunately, in the current FPGA configuration distributed with EMC2[12], the secondary encoders, enc-04, enc-05, enc-06, and enc-07 are wrongly configured. The input pins for enc-04 and enc-05 partly overlap, as do the pins for enc-06 and enc-07. Thus it is possible to use enc-04 and enc-06 simultaneously, but using enc-04 and enc-05 is not possible since counts on enc-04 will make the count on enc-05 jump by +/- 1. If you are using pins in-00 to in-11 as general purpose inputs you are not affected by this bug.

6.5.5.3 Connector P4

The index mask masks the index input of the encoder so that the encoder index can be combined with a mechanical switch or opto detector to clear or latch the encoder counter only when the mask input bit is in proper state (selected by mask polarity bit) and encoder index occurs. This is useful for homing. The behaviour of these pins is controlled by the Counter Control Register (CCR), however there is currently no function in the driver to change the CCR. See REGMAP4[13] for a description of the CCR.

[12] emc2/src/hal/drivers/m5i20_HM5-4E.h dated 2005/06/07
[13] emc2/src/hal/drivers/m5i20/REGMAP4E

m5i20 card connector P4	Function/HAL-pin	Secondary Function/HAL-pin
1	in-16	enc-00 index mask
3	in-17	enc-01 index mask
5	in-18	enc-02 index mask
7	in-19	enc-03 index mask
9	in-20	
11	in-21	
13	in-22	
15	in-23	
17	in-24	enc-04 index mask
19	in-25	enc-05 index mask
21	in-26	enc-06 index mask
23	in-27	enc-07 index mask
25	in-28	
27	in-29	
29	in-30	
31	in-31	
33	out-08	
35	out-09	
37	out-10	
39	out-11	
41	out-12	
43	out-13	
45	out-14	
47	out-15	
49	Power +5 V (or +3.3V ?)	
all even pins	Ground	

6.5.5.4 LEDs

The status LEDs will monitor one motion channel set by the `m5i20.<board>.led-view` parameter. A call to `m5i20.<board>.misc-update` is required to update the LEDs.

LED name	Output
LED0	IRQLatch ?
LED1	enc-<channel> A
LED2	enc-<channel> B
LED3	enc-<channel> index
LED4	dac-<channel> DIR
LED5	dac-<channel>
LED6	dac-<channel>-enable
LED7	watchdog timeout ?

6.6 Vital Systems Motenc-100 and Motenc-LITE

The Vital Systems Motenc-100 and Motenc-LITE are 8- and 4-channel servo control boards. The Motenc-100 provides 8 quadrature encoder counters, 8 analog inputs, 8 analog outputs, 64 (68?) digital inputs, and 32 digital outputs. The Motenc-LITE has only 4 encoder counters, 32 digital inputs and 16 digital outputs, but it still has 8 analog inputs and 8 analog outputs. The driver automatically identifies the installed board and exports the appropriate HAL objects.[14]

Installing:

[14] Ideally the encoders, DACs, ADCs, and digital I/O would comply with the canonical interfaces defined earlier, but they don't. Fixing that is on the things-to-do list.

```
emc2$ halcmd loadrt hal_motenc
```

During loading (or attempted loading) the driver prints some usefull debugging message to the kernel log, which can be viewed with **dmesg**.

6.6.1 Removing

```
emc2$ halcmd unloadrt hal_motenc
```

6.6.2 Pins

In the following pins, parameters, and functions, <board> is the board ID. According to the naming conventions the first board should always have an ID of zero. However this driver sets the ID based on a pair of jumpers on the baord, so it may be non-zero even if there is only one board.

- (S32) motenc.<board>.enc-<channel>-count – Encoder position, in counts.
- (FLOAT) motenc.<board>.enc-<channel>-position – Encoder position, in user units.
- (BIT) motenc.<board>.enc-<channel>-index – Current status of index pulse input.
- (BIT) motenc.<board>.enc-<channel>-idx-latch – Driver sets this pin true when it latches an index pulse (enabled by latch-index). Cleared by clearing latch-index.
- (BIT) motenc.<board>.enc-<channel>-latch-index – If this pin is true, the driver will reset the counter on the next index pulse.
- (BIT) motenc.<board>.enc-<channel>-reset-count – If this pin is true, the counter will immediately be reset to zero, and the pin will be cleared.
- (FLOAT) motenc.<board>.dac-<channel>-value – Analog output value for DAC (in user units, see -gain and -offset)
- (FLOAT) motenc.<board>.adc-<channel>-value – Analog input value read by ADC (in user units, see -gain and -offset)
- (BIT) motenc.<board>.in-<channel> – State of digital input pin, see canonical digital input.
- (BIT) motenc.<board>.in-<channel>-not – Inverted state of digital input pin, see canonical digital input.
- (BIT) motenc.<board>.out-<channel> – Value to be written to digital output, seen canonical digital output.
- (BIT) motenc.<board>.estop-in – Dedicated estop input, more details needed.
- (BIT) motenc.<board>.estop-in-not – Inverted state of dedicated estop input.
- (BIT) motenc.<board>.watchdog-reset – Bidirectional, - Set TRUE to reset watchdog once, is automatically cleared.

6.6.3 Parameters

- (FLOAT) motenc.<board>.enc-<channel>-scale – The number of counts / user unit (to convert from counts to units).
- (FLOAT) motenc.<board>.dac-<channel>-offset – Sets the DAC offset.
- (FLOAT) motenc.<board>.dac-<channel>-gain – Sets the DAC gain (scaling).
- (FLOAT) motenc.<board>.adc-<channel>-offset – Sets the ADC offset.

- (FLOAT) motenc.<board>.adc-<channel>-gain – Sets the ADC gain (scaling).
- (BIT) motenc.<board>.out-<channel>-invert – Inverts a digital output, see canonical digital output.
- (U32) motenc.<board>.watchdog-control – Configures the watchdog. The value may be a bitwise OR of the following values:

Bit #	Value	Meaning
0	1	Timeout is 16ms if set, 8ms if unset
2	4	Watchdog is enabled
4	16	Watchdog is automatically reset by DAC writes (the HAL dac-write function)

  Typically, the useful values are 0 (watchdog disabled) or 20 (8ms watchdog enabled, cleared by dac-write).
- (U32) motenc.<board>.led-view – Maps some of the I/O to onboard LEDs?

6.6.4 Functions

- (FUNCT) motenc.<board>.encoder-read – Reads all encoder counters.
- (FUNCT) motenc.<board>.adc-read – Reads the analog-to-digital converters.
- (FUNCT) motenc.<board>.digital-in-read – Reads digital inputs.
- (FUNCT) motenc.<board>.dac-write – Writes the voltages to the DACs.
- (FUNCT) motenc.<board>.digital-out-write – Writes digital outputs.
- (FUNCT) motenc.<board>.misc-update – Updates misc stuff.

6.7 Pico Systems PPMC (Parallel Port Motion Control)

Pico Systems has a family of boards for doing servo, stepper, and pwm control. The boards connect to the PC through a parallel port working in EPP mode. Although most users connect one board to a parallel port, in theory any mix of up to 8 or 16 boards can be used on a single parport. One driver serves all types of boards. The final mix of I/O depends on the connected board(s). The driver doesn't distinguish between boards, it simply numbers I/O channels (encoders, etc) starting from 0 on the first card.

Installing:

```
emc2$ halcmd loadrt hal_ppmc port_addr=<addr1>[,<addr2>[,<addr3>]]
```

The **port_addr** parameter tells the driver what parallel port(s) to check. By default, **<addr1>** is 0x0378, and **<addr2>** and **<addr3>** are not used.The driver searches the entire address space of the enhanced parallel port(s) at **port_addr**, looking for any board(s) in the PPMC family. It then exports HAL pins for whatever it finds. During loading (or attempted loading) the driver prints some usefull debugging message to the kernel log, which can be viewed with **dmesg**.

6.7.1 Removing

```
emc2$ halcmd unloadrt hal_ppmc
```

6.7.2 Pins

In the following pins, parameters, and functions, <board> is the board ID. According to the naming conventions the first board should always have an ID of zero. However this driver sets the ID based on a pair of jumpers on the baord, so it may be non-zero even if there is only one board.

- (S32) ppmc.<port>.encoder.<channel>.count – Encoder position, in counts.
- (S32) ppmc.<port>.encoder.<channel>.delta – Change in counts since last read.
- (FLOAT) ppmc.<port>.encoder.<channel>.position – Encoder position, in user units.
- (BIT) ppmc.<port>.encoder.<channel>.index – Something to do with index pulse.[15]
- (BIT) ppmc.<port>.pwm.<channel>.enable – Enables a PWM generator.
- (FLOAT) ppmc.<port>.pwm.<channel>.value – Value which determines the duty cycle of the PWM waveforms. The value is divided by pwm.<channel>.scale, and if the result is 0.6 the duty cycle will be 60%, and so on. Negative values result in the duty cycle being based on the absolute value, and the direction pin is set to indicate negative.
- (BIT) ppmc.<port>.stepgen.<channel>.enable – Enables a step pulse generator.
- (FLOAT) ppmc.<port>.stepgen.<channel>.velocity – Value which determines the step frequency. The value is multiplied by stepgen.<channel>.scale, and the result is the frequency in steps per second. Negative values result in the frequency being based on the absolute value, and the direction pin is set to indicate negative.
- (BIT) ppmc.<port>.in-<channel> – State of digital input pin, see canonical digital input.
- (BIT) ppmc.<port>.in.<channel>.not – Inverted state of digital input pin, see canonical digital input.
- (BIT) ppmc.<port>.out-<channel> – Value to be written to digital output, seen canonical digital output.

6.7.3 Parameters

- (FLOAT) ppmc.<port>.enc.<channel>.scale – The number of counts / user unit (to convert from counts to units).
- (FLOAT) ppmc.<port>.pwm.<channel-range>.freq – The PWM carrier frequency, in Hz. Applies to a group of four consecutive PWM generators, as indicated by <channel-range>. Minimum is 153Hz, maximum is 500KHz.
- (FLOAT) ppmc.<port>.pwm.<channel>.scale – Scaling for PWM generator. If scale is X, then the duty cycle will be 100% when the value pin is X (or -X).
- (FLOAT) ppmc.<port>.pwm.<channel>.max-dc – Maximum duty cycle, from 0.0 to 1.0.
- (FLOAT) ppmc.<port>.pwm.<channel>.min-dc – Minimum duty cycle, from 0.0 to 1.0.
- (FLOAT) ppmc.<port>.pwm.<channel>.duty-cycle – Actual duty cycle (used mostly for troubleshooting.)
- (BIT) ppmc.<port>.pwm.<channel>.bootstrap – If true, the PWM generator will generate a short sequence of pulses of both polarities when it is enabled, to charge the bootstrap capacators used on some MOSFET gate drivers.

[15] Index handling does _not_ comply with the canonical encoder interface, and should be changed.

- (U32) ppmc.<port>.stepgen.<channel-range>.setup-time – Sets minimum time between direction change and step pulse, in units of 100nS. Applies to a group fof four consecutive PWM generators, as indicated by <channel-range>.
- (U32) ppmc.<port>.stepgen.<channel-range>.pulse-width – Sets width of step pulses, in units of 100nS. Applies to a group fof four consecutive PWM generators, as indicated by <channel-range>.
- (U32) ppmc.<port>.stepgen.<channel-range>.pulse-space-min – Sets minimum time between pulses, in units of 100nS. The maximum step rate is 1/(100nS * (pulse-width + pulse-space-min)). Applies to a group fof four consecutive PWM generators, as indicated by <channel-range>.
- (FLOAT) ppmc.<port>.stepgen.<channel>.scale – Scaling for step pulse generator. The step frequency in Hz is the absolute value of velocity * scale.
- (FLOAT) ppmc.<port>.stepgen.<channel>.max-vel – The maximum value for velocity. Commands greater than max-vel will be clamped. Also applies to negative values. (The absolute value is clamped.)
- (FLOAT) ppmc.<port>.stepgen.<channel>.frequency – Actual step pulse frequency in Hz (used mostly for troubleshooting.)
- (BIT) ppmc.<port>.out.<channel>.invert – Inverts a digital output, see canonical digital output.

6.7.4 Functions

- (FUNCT) ppmc.<port>.read – Reads all inputs (digital inputs and encoder counters) on one port.
- (FUNCT) ppmc.<port>.write – Writes all outputs (digital outputs, stepgens, PWMs) on one port.

Chapter 7

Internal Components

7.1 Stepgen

This component provides software based generation of step pulses in response to position commands. It has a built in pre-tuned position loop, so PID tuning is not required. This component is strongly recommended for stepper based EMC machines, since it eliminates the need to use (and tune) a separate PID loop. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum step rates of 10kHz to perhaps 50kHz. Figure 7.1 shows three block diagrams, each is a single step pulse generator. The first diagram is for step type '0', (step and direction). The second is for step type '1' (up/down, or pseudo-PWM), and the third is for step types 2 through 14 (various stepping patterns).

7.1.1 Installing

```
emc2$ halcmd loadrt stepgen step_type=<config-array>
```

`<config-array>` is a series of comma separated decimal integers. Each number causes a single step pulse generator to be loaded, the value of the number determines the stepping type. For example:

```
emc2# halcmd loadrt stepgen.o step_type=0,0,2
```

will install three step generators, two with step type '0' (step and direction) and one with step type '2' (quadrature). The default value for `<config-array>` is "`0,0,0`" which will install three type '0' (step/dir) generators. The maximum number of step generators is 8 (as defined by MAX_CHAN in stepgen.c). Each generator is independent, but all are updated by the same function(s) at the same time. In the following descriptions, `<chan>` is the number of a specific generator. The first generator is number 0.

7.1.2 Removing

```
emc2$ halcmd unloadrt stepgen
```

7.1.3 Pins

Each step pulse generator will have only some of these pins, depending on the step type selected.

- `(FLOAT) stepgen.<chan>.position-cmd` – Desired motor position, in position units (inches, mm, etc).
- `(s32) stepgen.<chan>.count` – Feedback position in counts, updated by `capture_position()`.

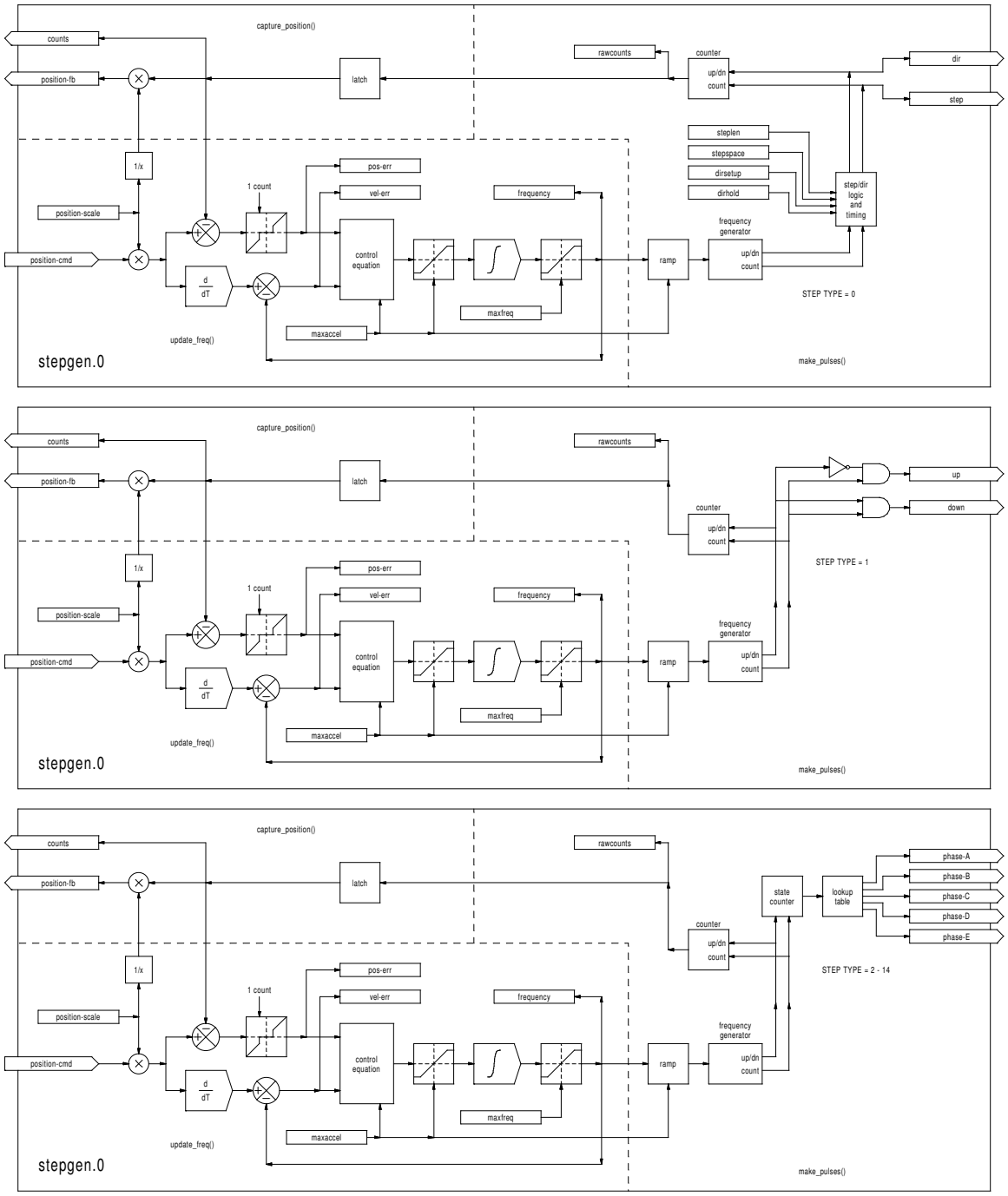


Figure 7.1: Step Pulse Generator Block Diagram

- (FLOAT) stepgen.<chan>.position-fb – Feedback position in position units, updated by capture_position().
- (BIT) stepgen.<chan>.step – Step pulse output (step type 0 only).
- (BIT) stepgen.<chan>.dir – Direction output (step type 0 only).
- (BIT) stepgen.<chan>.up – UP pseudo-PWM output (step type 1 only).
- (BIT) stepgen.<chan>.down – DOWN pseudo-PWM output (step type 1 only).
- (BIT) stepgen.<chan>.phase-A – Phase A output (step types 2-14 only).
- (BIT) stepgen.<chan>.phase-B – Phase B output (step types 2-14 only).
- (BIT) stepgen.<chan>.phase-C – Phase C output (step types 3-14 only).
- (BIT) stepgen.<chan>.phase-D – Phase D output (step types 5-14 only).
- (BIT) stepgen.<chan>.phase-E – Phase E output (step types 11-14 only).

7.1.4 Parameters

- (FLOAT) stepgen.<chan>.position-scale – Steps per position unit. This parameter is used for both output and feedback.
- (FLOAT) stepgen.<chan>.maxfreq – Maximum step rate, in steps per second. If 0.0, has no effect.
- (FLOAT) stepgen.<chan>.maxaccel – Maximum accel/decel rate, in steps per second squared. If 0.0, has no effect.
- (FLOAT) stepgen.<chan>.pos-err – The position error - difference between commanded and actual position, in steps.
- (FLOAT) stepgen.<chan>.vel-err – The velocity error - in steps per second.
- (FLOAT) stepgen.<chan>.frequency – The current step rate, in steps per second. This is the output of the position loop.
- (FLOAT) stepgen.<chan>.steplen – Length of a step pulse (step type 0 only).
- (FLOAT) stepgen.<chan>.stepspace – Minimum spacing between two step pulses (step type 0 only).
- (FLOAT) stepgen.<chan>.dirsetup – Minimum time from a direction change to the beginning of the next step pulse (step type 0 only).
- (FLOAT) stepgen.<chan>.dirhold – Minmum time from the end of a step pulse to a direction change (step type 0 only).
- (s32) stepgen.<chan>.rawcounts – The raw feedback count, updated by make_pulses().

The values of maxfreq and maxaccel are used by the internal position loop to avoid generating step pulse trains that the motor cannot follow. When set to values that are appropriate for the motor, even a large instantaneous change in commanded position will result in a smooth trapezoidal move to the new location. The algorithm works by measuring both position error and velocity error, and calculating an acceleration that attempts to reduce both to zero at the same time. For more details, including the contents of the “control equation” box, consult the code.

7.1.5 Step Types

The step generator supports 15 different "step types". Step type 0 is the most familiar, standard step and direction. When configured for step type 0, there are four extra parameters that determine the exact timing of the step and direction signals. See figure 7.2 for the meaning of these parameters. The parameters are integers, and represent a number of calls to `make_pulses()`. For example, if `make_pulses()` is called every 16uS, and steplen is 2, then the step pulses will be 2 x 16 = 32uS long. The default value for all four of the parameters is 1. Since one step requires `steplen` periods high and `stepspace` periods low, the maximum frequency is the thread frequency divided by `(steplen+stepspace)`. If `maxfreq` is set higher than that limit, it will be lowered automatically. If maxfreq is zero, it will remain zero, but the output frequency will still be limited.

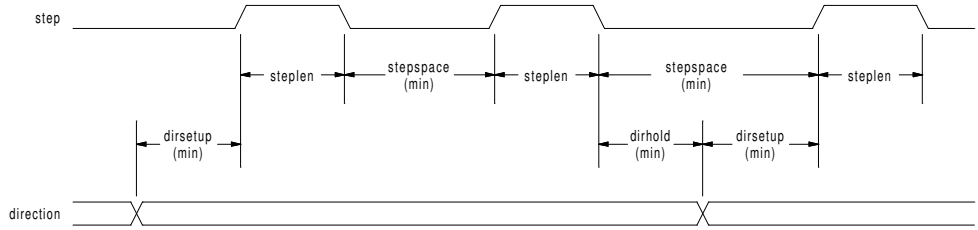


Figure 7.2: Step and Direction Timing

Step type 1 has two outputs, up and down. Pulses appear on one or the other, depending on the direction of travel. Each pulse is one thread period long, and the pulses are separated by at least one thread period. As a result, the maximum step frequency is half of the thread rate. If `maxfreq` is set higher than the limit it will be lowered. If `maxfreq` is zero, it will remain zero but the output frequency will still be limited.

Step types 2 through 14 are state based, and have from two to five outputs. On each step, a state counter is incremented or decremented. Figures 7.3, 7.4, and 7.5 show the output patterns as a function of the state counter. The maximum frequency is the same as the thread rate, and as in the other modes, `maxfreq` will be lowered if it is above the limit.

7.1.6 Functions

The component exports three functions. Each function acts on all of the step pulse generators - running different generators in different threads is not supported.

- `(FUNCT) stepgen.make-pulses` – High speed function to generate and count pulses (no floating point).

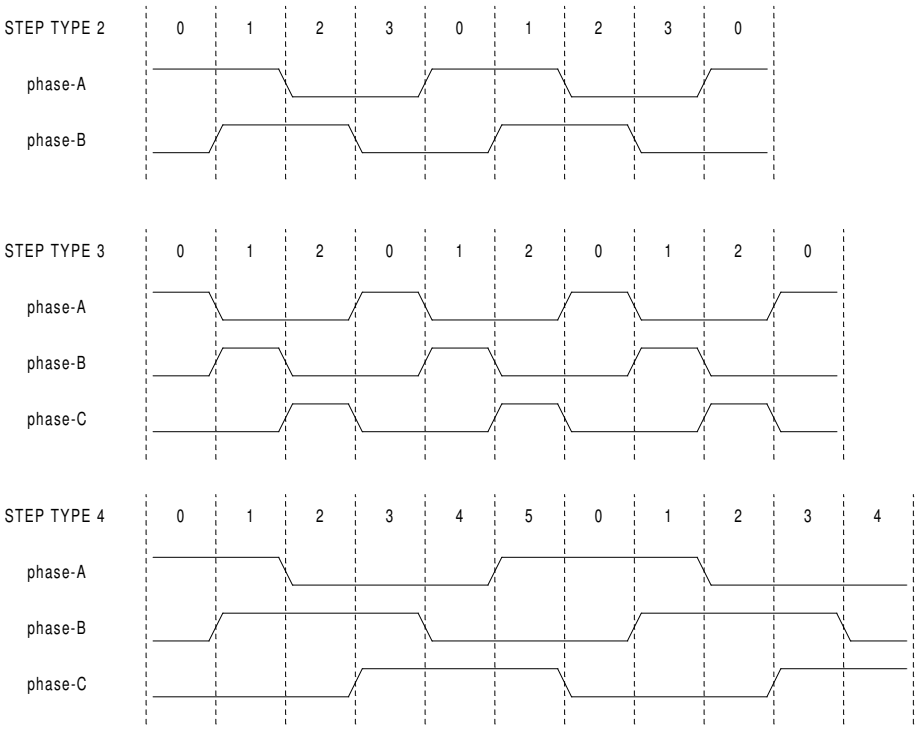


Figure 7.3: Quadrature and Three Phase Step Types

- `(FUNCT) stepgen.update-freq` – Low speed function does position to velocity conversion, scaling and limiting.
- `(FUNCT) stepgen.capture-position` – Low speed function for feedback, updates latches and scales position.

The high speed function `stepgen.make-pulses` should be run in a very fast thread, from 10 to 50uS depending on the capabilities of the computer. That thread's period determines the maximum step frequency, and is also the time unit used by the length, space, setup, and hold parameters (step type 0). The other two functions can be called at a much lower rate.

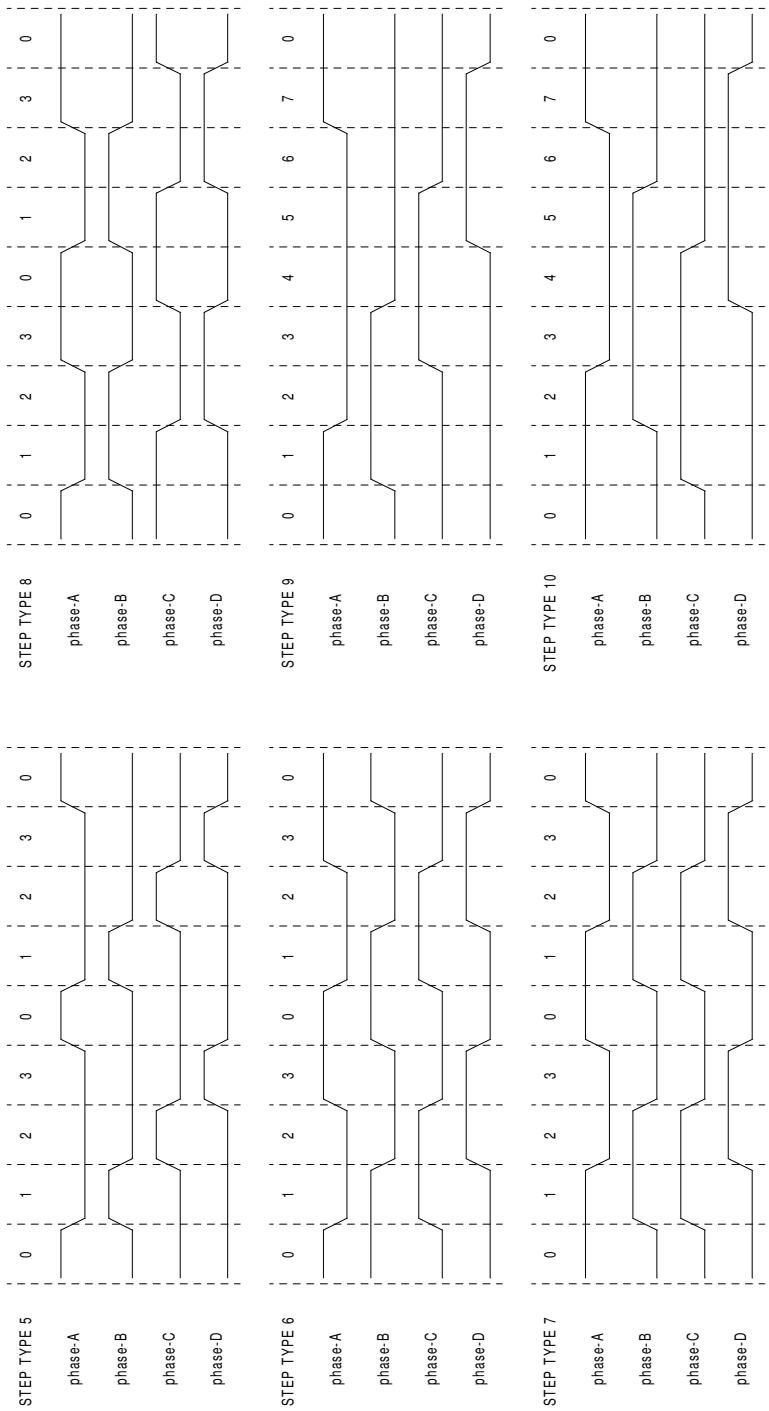


Figure 7.4: Four-Phase Step Types

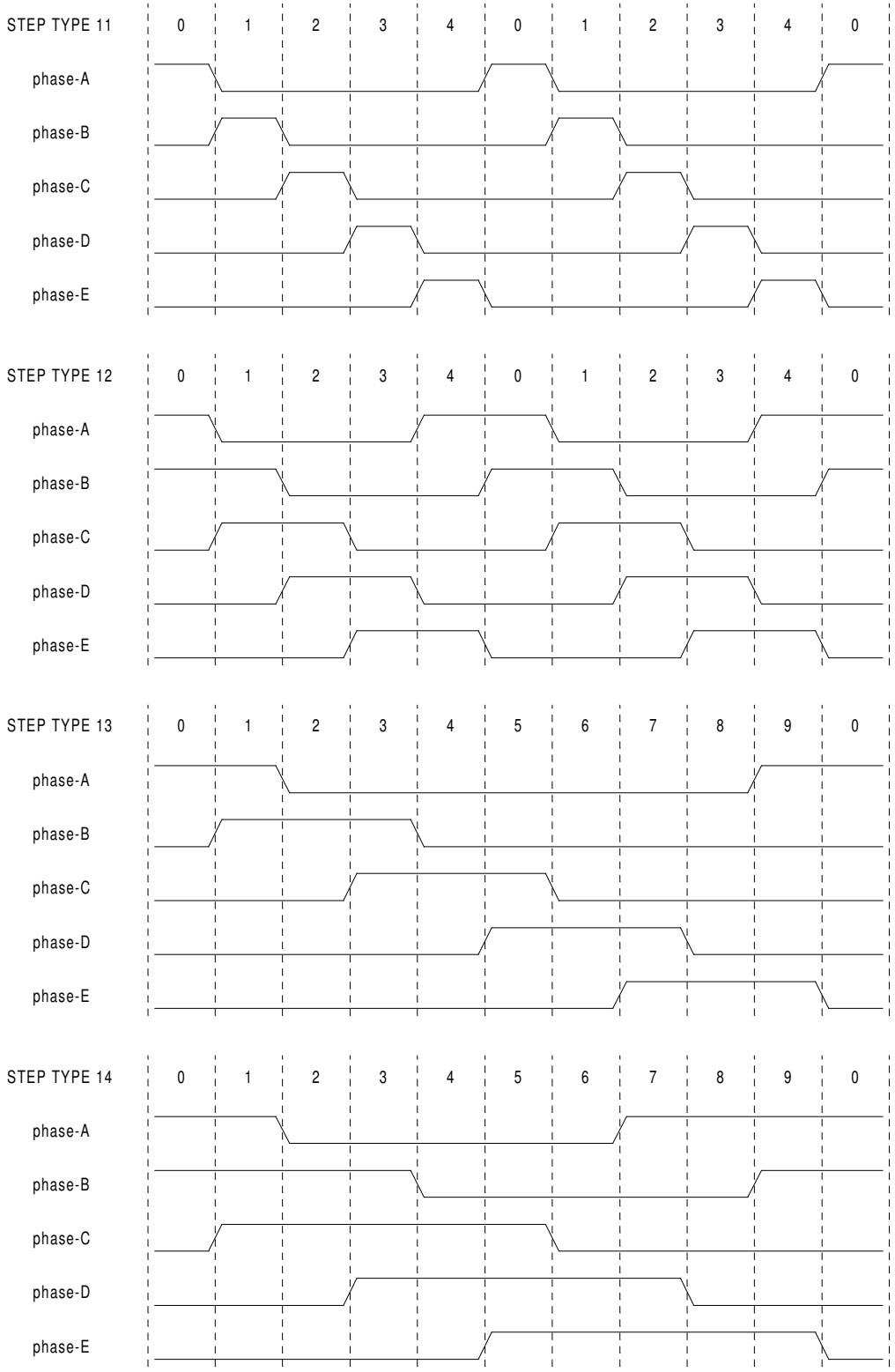


Figure 7.5: Five-Phase Step Types

7.2 Freqgen

This component provides software based generation of step pulses from a frequency or velocity command. EMC normally uses position commands, not velocity commands, and stepgen (described in section 7.1 is more appropriate. However, there may be applications where velocity based pulses are needed. One such application is using step type 1 as a pseudo-PWM output. Combined with a PID loop. the software encoder counter module, and a simple H-bridge, this has been used to run small DC servomotors, however now that the pwmgen component has been added, it is a better choice for this purpose. Another possible use for freqgen is to simply spin a stepper motor at a constant (or adjustable) rate, while ignoring position. Freqgen uses the same pulse generator core as stepgen, however it has no position loop. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum step rates of 10kHz to perhaps 50kHz. Figure 7.6 shows three block diagrams, each is a single step pulse generator. The first diagram is for step type '0', (step and direction). The second is for step type '1' (up/down, or pseudo-PWM), and the third is for step types 2 through 14 (various stepping patterns).

7.2.1 Installing

```
emc2$ halcmd loadrt freqgen step_type=<config-array>
```

`<config-array>` is a series of comma separated decimal integers. Each number causes a single frequency generator to be loaded, the value of the number determines the stepping type. For example:

```
emc2$ halcmd loadrt freqgen step_type=0,0,2
```

will install three frequency generators, two with step type '0' (step and direction) and one with step type '2' (quadrature). The default value for `<config-array>` is "`0,0,0`" which will install three type '0' (step/dir) generators. The maximum number of frequency generators is 8 (as defined by MAX_CHAN in freqgen.c). Each generator is independent, but all are updated by the same function(s) at the same time. In the following descriptions, `<chan>` is the number of a specific generator. The first generator is number 0.

7.2.2 Removing

```
emc2$ halcmd unloadrt freqgen
```

7.2.3 Pins

Each frequency generator will have only some of these pins, depending on the step type selected.

- (FLOAT) `freqgen.<chan>.velocity` – Desired velocity, in arbitrary units.
- (BIT) `freqgen.<chan>.step` – Step pulse output (step type 0 only).
- (BIT) `freqgen.<chan>.dir` – Direction output (step type 0 only).
- (BIT) `freqgen.<chan>.up` – UP pseudo-PWM output (step type 1 only).
- (BIT) `freqgen.<chan>.down` – DOWN pseudo-PWM output (step type 1 only).
- (BIT) `freqgen.<chan>.phase-A` – Phase A output (step types 2-14 only).
- (BIT) `freqgen.<chan>.phase-B` – Phase B output (step types 2-14 only).
- (BIT) `freqgen.<chan>.phase-C` – Phase C output (step types 3-14 only).
- (BIT) `freqgen.<chan>.phase-D` – Phase D output (step types 5-14 only).

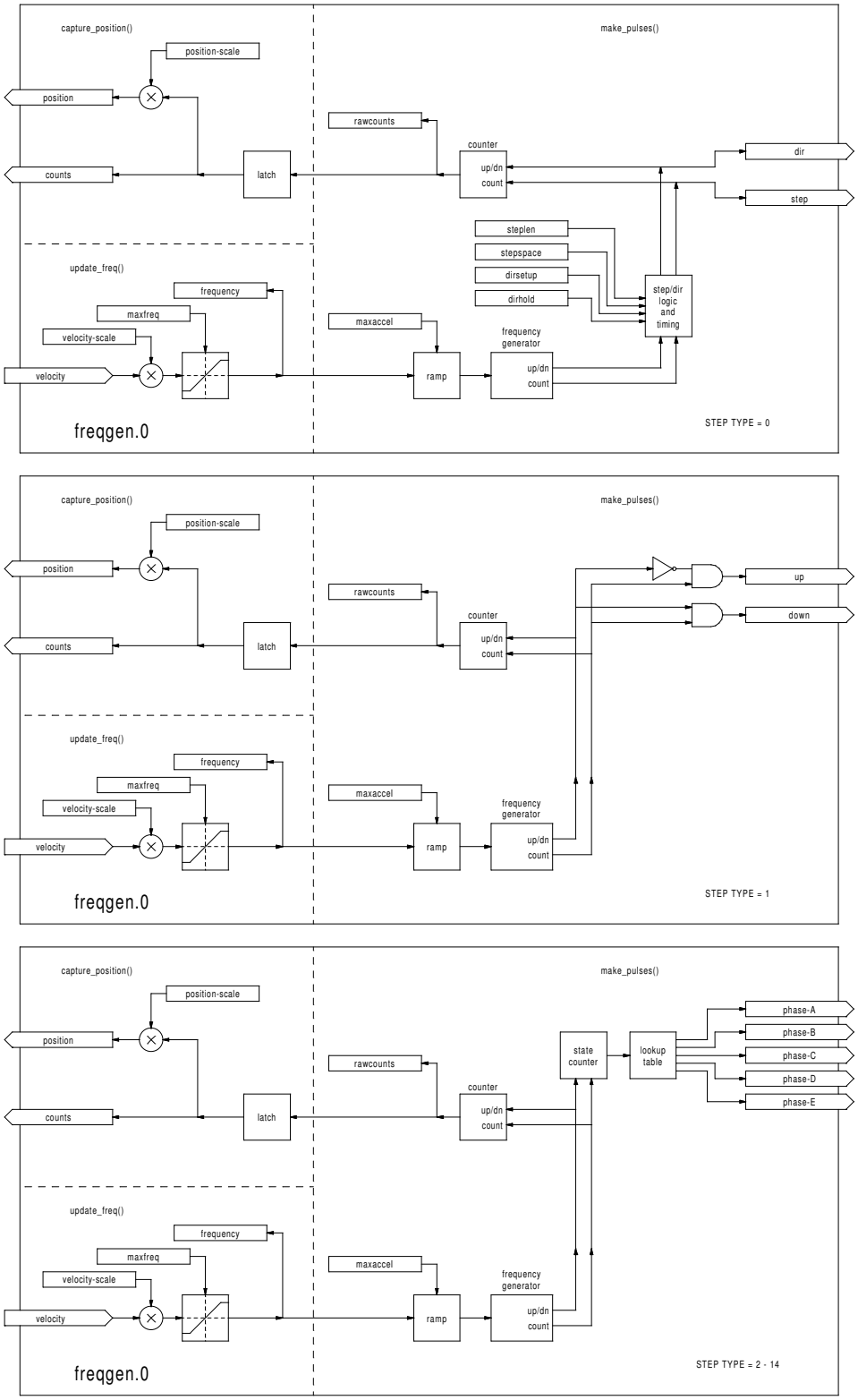


Figure 7.6: Step Pulse Generator Block Diagram

- (BIT) freqgen.<chan>.phase-E – Phase E output (step types 11-14 only).
- (s32) freqgen.<chan>.count – Feedback position in counts, updated by capture_position().
- (FLOAT) freqgen.<chan>.position-fb – Position feedback in arbitrary units updated by capture_position().

7.2.4 Parameters

- (FLOAT) freqgen.<chan>.velocity-scale – Scaling factor to convert from velocity units to pulses per second (Hz).
- (FLOAT) freqgen.<chan>.maxfreq – Maximum frequency, in Hz. If 0.0, has no effect. If set higher than internal limits, next call of update_freq() will set it to the internal limit.
- (FLOAT) freqgen.<chan>.frequency – The current frequency, in Hz. This is the value after scaling and limiting.
- (FLOAT) freqgen.<chan>.maxaccel – Maximum accel/decel rate, in Hz per second. If 0.0, has no effect.
- (FLOAT) freqgen.<chan>.steplen – Length of a step pulse (step type 0 only).
- (FLOAT) freqgen.<chan>.stepspace – Minimum spacing between two step pulses (step type 0 only).
- (FLOAT) freqgen.<chan>.dirsetup – Minimum time from a direction change to the beginning of the next step pulse (step type 0 only).
- (FLOAT) freqgen.<chan>.dirhold – Minmum time from the end of a step pulse to a direction change (step type 0 only).
- (s32) freqgen.<chan>.rawcounts – The raw feedback count, updated by make_pulses().
- (FLOAT) freqgen.<chan>.position-scale – The scale factor used to convert from feedback counts to position units.

7.2.5 Step Types

The frequency generator supports 15 different "step types". Except for stepping type 1, they are identical to those generated by the stepgen component. Refer to section 7.1.5 for more information.

There is one difference between stepgen and freqgen. Step type 1 has two outputs, up and down. Pulses appear on one or the other, depending on the direction of travel. Each pulse is one thread period long. Stepgen assumes that you need a distinct pulse for each step, so it limits the frequency to half of the thread rate, to allow for one low period between pulses. However, freqgen allows higher frequencies, up to the thread rate. This allows step type 1 to be used as a pseudo-PWM source, or filtered to use as a D-to-A converter. At the maximum frequency (equal to the thread rate), the up or down output will remain on constantly.

7.2.6 Functions

The component exports three functions. Each function acts on all of the step pulse generators - running different generators in different threads is not supported.

- (FUNCT) freqgen.make-pulses – High speed function to generate and count pulses (no floating point).
- (FUNCT) freqgen.update-freq – Low speed function to scale and limit velocity command.

- (FUNCT) freqgen.capture-position – Low speed function for feedback, updates latches and scales position.

The high speed function freqgen.make-pulses should be run in a very fast thread, from 10 to 50uS depending on the capabilities of the computer. That thread's period determines the maximum step frequency, and is also the time unit used by the length, space, setup, and hold parameters (step type 0). The other two functions can be called at a much lower rate.

7.3 PWMgen

This component provides software based generation of PWM (Pulse Width Modulation) and PDM (Pulse Density Modulation) waveforms. It is a realtime component only, and depending on CPU speed, etc, is capable of PWM frequencies from a few hundred Hertz at pretty good resolution, to perhaps 10KHz with limited resolution.

7.3.1 Installing

```
emc2$ halcmd loadrt pwmgen output_type=<config-array>
```

<config-array> is a series of comma separated decimal integers. Each number causes a single PWM generator to be loaded, the value of the number determines the output type. For example:

```
emc2$ halcmd loadrt pwmgen step_type=0,1,2
```

will install three PWM generators. The first one will use output type '0' (PWM only), the next uses output type 1 (PWM and direction) and the last one uses output type 2 (UP and DOWN). There is no default value, if <config-array> is not specified, no PWM generators will be installed. The maximum number of frequency generators is 8 (as defined by MAX_CHAN in pwmgen.c). Each generator is independent, but all are updated by the same function(s) at the same time. In the following descriptions, <chan> is the number of a specific generator. The first generator is number 0.

7.3.2 Removing

```
emc2$ halcmd unloadrt pwmgen
```

7.3.3 Pins

Each PWM generator will have the following pins:

- (FLOAT) pwmgen.<chan>.value – Command value, in arbitrary units. Will be scaled by the scale parameter (see below).
- (BIT) pwmgen.<chan>.enable – Enables or disables the PWM generator outputs.

Each PWM generator will also have some of these pins, depending on the output type selected:

- (BIT) pwmgen.<chan>.pwm – PWM (or PDM) output, (output types 0 and 1 only).
- (BIT) pwmgen.<chan>.dir – Direction output (output type 1 only).
- (BIT) pwmgen.<chan>.up – PWM/PDM output for positive input value (output type 2 only).
- (BIT) pwmgen.<chan>.down – PWM/PDM output for negative input value (output type 2 only).

7.3.4 Parameters

- (FLOAT) pwmgen.<chan>.scale – Scaling factor to convert value from arbitrary units to duty cycle.
- (FLOAT) pwmgen.<chan>.pwm-freq – Desired PWM frequency, in Hz. If 0.0, generates PDM instead of PWM. If set higher than internal limits, next call of update_freq() will set it to the internal limit. If non-zero, and dither is false, next call of update_freq() will set it to the nearest integer multiple of the make_pulses() function period.
- (BIT) pwmgen.<chan>.dither-pwm – If true, enables dithering to achieve average PWM frequencies or duty cycles that are unobtainable with pure PWM. If false, both the PWM frequency and the duty cycle will be rounded to values that can be achieved exactly.
- (FLOAT) pwmgen.<chan>.min-dc – Minimum duty cycle, between 0.0 and 1.0 (duty cycle will go to zero when disabled, regardless of this setting).
- (FLOAT) pwmgen.<chan>.max-dc – Maximum duty cycle, between 0.0 and 1.0.
- (FLOAT) pwmgen.<chan>.curr-dc – Current duty cycle - after all limiting and rounding (read only).

7.3.5 Output Types

The PWM generator supports three different "output types". Type 0 has a single output pin. Only positive commands are accepted, negative values are treated as zero (and will be affected by min-dc if it is non-zero). Type 1 has two output pins, one for the PWM/PDM signal and one to indicate direction. The duty cycle on the PWM pin is based on the absolute value of the command, so negative values are acceptable. The direction pin is false for positive commands, and true for negative commands. Finally, type 2 also has two outputs, called up and down. For positive commands, the PWM signal appears on the up output, and the down output remains false. For negative commands, the PWM signal appears on the down output, and the up output remains false. Output type 2 is suitable for driving most H-bridges.

7.3.6 Functions

The component exports two functions. Each function acts on all of the PWM generators - running different generators in different threads is not supported.

- (FUNCT) pwmgen.make-pulses – High speed function to generate PWM waveforms (no floating point).
- (FUNCT) pwmgen.update – Low speed function to scale and limit value and handle other paremeters.

The high speed function pwmgen.make-pulses should be run in a very fast thread, from 10 to 50uS depending on the capabilities of the computer. That thread's period determines the maximum PWM carrier frequency, as well as the resolution of the PWM or PDM signals. The other function can be called at a much lower rate.

7.4 Encoder

This component provides software based counting of signals from quadrature encoders. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum count rates of 10kHz to perhaps 50kHz. Figure 7.7 is a block diagram of one channel of encoder counter.

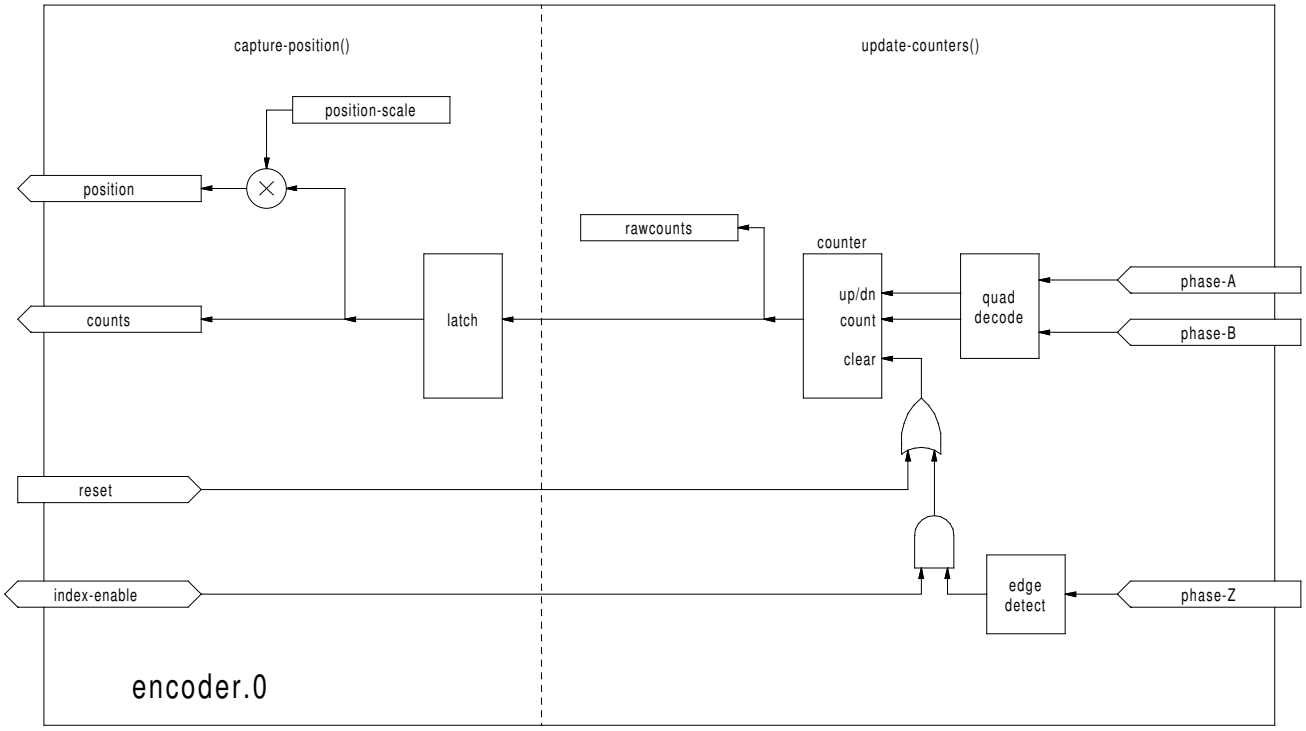


Figure 7.7: Encoder Counter Block Diagram

7.4.1 Installing

```
emc2$ halcmd loadrt encoder [num_chan=<counters>]
```

`<counters>` is the number of encoder counters that you want to install. If `numchan` is not specified, three counters will be installed. The maximum number of counters is 8 (as defined by MAX_CHAN in encoder.c). Each counter is independent, but all are updated by the same function(s) at the same time. In the following descriptions, `<chan>` is the number of a specific counter. The first counter is number 0.

7.4.2 Removing

```
emc2$ halcmd unloadrt encoder
```

7.4.3 Pins

- (BIT) encoder.<chan>.phase-A – Phase A of the quadrature encoder signal.
- (BIT) encoder.<chan>.phase-B – Phase B of the quadrature encoder signal.
- (BIT) encoder.<chan>.phase-Z – Phase Z (index pulse) of the quadrature encoder signal.
- (BIT) encoder.<chan>.reset – See canonical encoder interface, section 4.5.
- (BIT) encoder.<chan>.velocity – Estimated speed of the quadrature signal.
- (BIT) encoder.<chan>.index-enable – See canonical encoder interface.
- (s32) encoder.<chan>.count – See canonical encoder interface.
- (FLOAT) encoder.<chan>.position – See canonical encoder interface.

7.4.4 Parameters

- (s32) encoder.<chan>.raw-count – The raw count value, updated by update-counters().
- (BIT) encoder.<chan>.x4-mode – Sets encoder to 4x or 1x mode. The 1x mode is usefull for some jogwheels.
- (FLOAT) encoder.<chan>.position-scale – See canonical encoder interface, section 4.5.

7.4.5 Functions

The component exports two functions. Each function acts on all of the encoder counters - running different counters in different threads is not supported.

- (FUNCT) encoder.update-counters – High speed function to count pulses (no floating point).
- (FUNCT) encoder.capture-position – Low speed function to update latches and scale position.

7.5 PID

This component provides Proportional/Integeral/Derivative control loops. It is a realtime component only. For simplicity, this discussion assumes that we are talking about position loops, however this component can be used to implement other feedback loops such as speed, torch height, temperature, etc. Figure 7.8 is a block diagram of a single PID loop.

7.5.1 Installing

```
emc2$ halcmd loadrt pid [num_chan=<loops>] [debug=1]
```

`<loops>` is the number of PID loops that you want to install. If `numchan` is not specified, one loop will be installed. The maximum number of loops is 16 (as defined by MAX_CHAN in pid.c). Each loop is completely independent. In the following descriptions, `<loopnum>` is the loop number of a specific loop. The first loop is number 0.

If `debug=1` is specified, the component will export a few extra parameters that may be useful during debugging and tuning. By default, the extra parameters are not exported, to save shared memory space and avoid cluttering the parameter list.

7.5.2 Removing

```
emc2$ halcmd unloadrt pid
```

7.5.3 Pins

The three most important pins are

- `(FLOAT) pid.<loopnum>.command` – The desired position, as commanded by another system component.
- `(FLOAT) pid.<loopnum>.feedback` – The present position, as measured by a feedback device such as an encoder.
- `(FLOAT) pid.<loopnum>.output` – A velocity command that attempts to move from the present position to the desired position.

For a position loop, 'command' and 'feedback' are in position units. For a linear axis, this could be inches, mm, meters, or whatever is relevant. Likewise, for an angular axis, it could be degrees, radians, etc. The units of the 'output' pin represent the change needed to make the feedback match the command. As such, for a position loop 'Output' is a velocity, in inches/sec, mm/sec, degrees/sec, etc. Time units are always seconds, and the velocity units match the position units. If command and feedback are in meters, then output is in meters per second.

Each loop has two other pins which are used to monitor or control the general operation of the component.

- `(FLOAT) pid.<loopnum>.error` – Equals `.command` minus `.feedback`.
- `(BIT) pid.<loopnum>.enable` – A bit that enables the loop. If `.enable` is false, all integrators are reset, and the output is forced to zero. If `.enable` is true, the loop operates normally.

7.5.4 Parameters

The PID gains, limits, and other 'tunable' features of the loop are implemented as parameters.

- `(FLOAT) pid.<loopnum>.Pgain` – Proportional gain

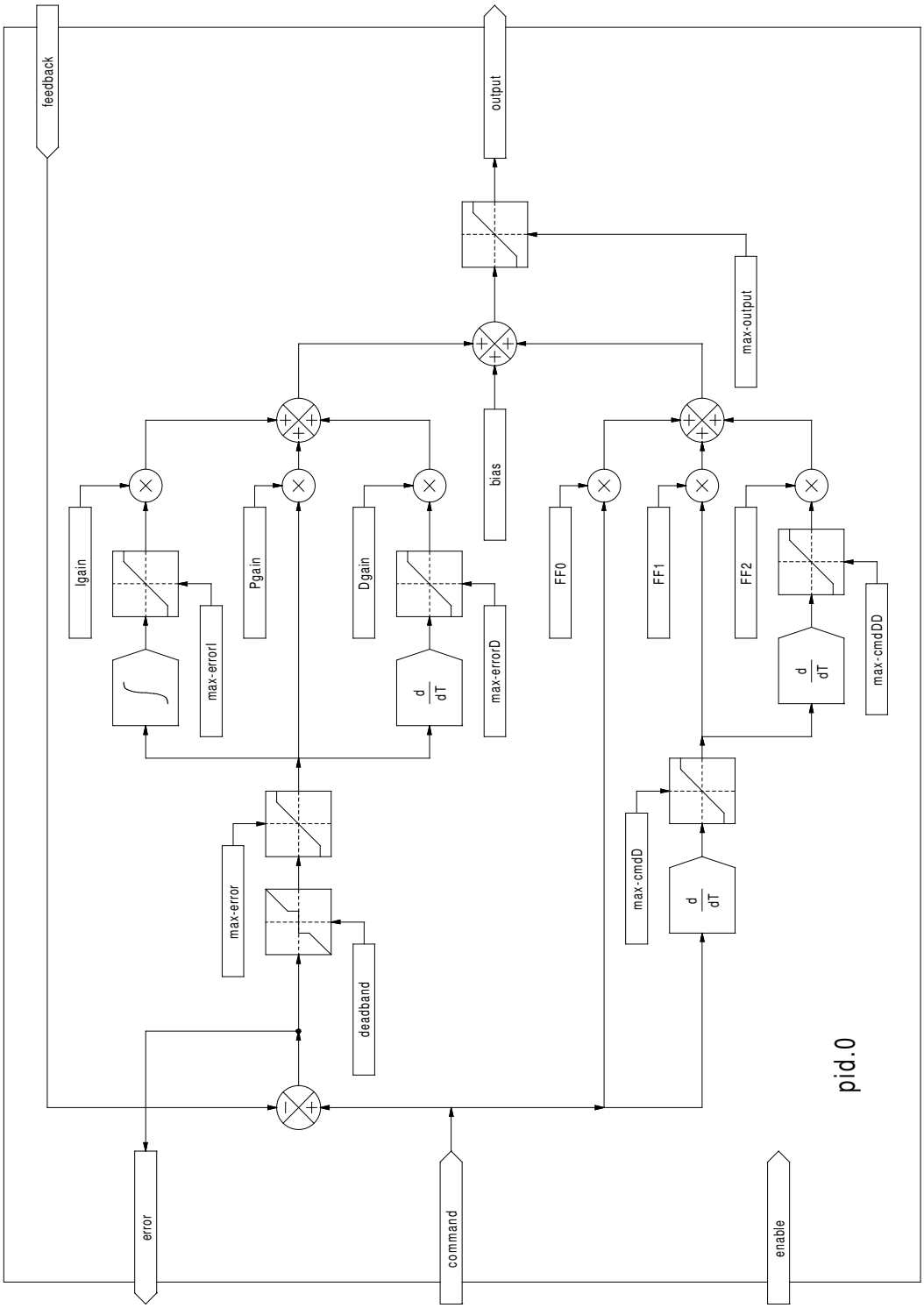


Figure 7.8: PID Loop Block Diagram

- (FLOAT) pid.<loopnum>.Igain – Integral gain
- (FLOAT) pid.<loopnum>.Dgain – Derivative gain
- (FLOAT) pid.<loopnum>.bias – Constant offset on output
- (FLOAT) pid.<loopnum>.FF0 – Zeroth order feedforward - output proportional to command (position).
- (FLOAT) pid.<loopnum>.FF1 – First order feedforward - output proportional to derivative of command (velocity).
- (FLOAT) pid.<loopnum>.FF2 – Second order feedforward - output proportional to 2nd derivative of command (acceleration)[1].
- (FLOAT) pid.<loopnum>.deadband – Amount of error that will be ignored
- (FLOAT) pid.<loopnum>.maxerror – Limit on error
- (FLOAT) pid.<loopnum>.maxerrorI – Limit on error integrator
- (FLOAT) pid.<loopnum>.maxerrorD – Limit on error derivative
- (FLOAT) pid.<loopnum>.maxcmdD – Limit on command derivative
- (FLOAT) pid.<loopnum>.maxcmdDD – Limit on command 2nd derivative
- (FLOAT) pid.<loopnum>.maxoutput – Limit on output value

All of the max??? limits are implemented such that if the parameter value is zero, there is no limit.

If debug=1 was specified when the component was installed, four additional parameters will be exported:

- (FLOAT) pid.<loopnum>.errorI – Integral of error.
- (FLOAT) pid.<loopnum>.errorD – Derivative of error.
- (FLOAT) pid.<loopnum>.commandD – Derivative of the command.
- (FLOAT) pid.<loopnum>.commandDD – 2nd derivative of the command.

7.5.5 Functions

The component exports one function for each PID loop. This function performs all the calculations needed for the loop. Since each loop has its own function, individual loops can be included in different threads and execute at different rates.

- (FUNCT) pid.<loopnum>.do_pid_calcs – Performs all calculations for a single PID loop.

If you want to understand the exact algorithm used to compute the output of the PID loop, refer to figure 7.8, the comments at the beginning of emc2/src/hal/components/pid.c, and of course to the code itself. The loop calculations are in the C function calc_pid().

[1] FF2 is not currently implemented, but it will be added. Consider this note a "FIXME" for the code

7.6 Simulated Encoder

The simulated encoder is exactly that. It produces quadrature pulses with an index pulse, at a speed controlled by a HAL pin. Mostly useful for testing.

7.6.1 Installing

```
emc2$ halcmd loadrt sim-encoder num_chan=<number>
```

`<number>` is the number of encoders that you want to simulate. If not specified, one encoder will be installed. The maximum number is 8 (as defined by MAX_CHAN in sim_encoder.c).

7.6.2 Removing

```
emc2$ halcmd unloadrt sim-encoder
```

7.6.3 Pins

- (FLOAT) `sim-encoder.<chan-num>.speed` – The speed command for the simulated shaft.
- (BIT) `sim-encoder.<chan-num>.phase-A` – Quadrature output.
- (BIT) `sim-encoder.<chan-num>.phase-B` – Quadrature output.
- (BIT) `sim-encoder.<chan-num>.phase-Z` – Index pulse output.

When `.speed` is positive, `.phase-A` leads `.phase-B`.

7.6.4 Parameters

- (U32) `sim-encoder.<chan-num>.ppr` – Pulses Per Revolution.
- (FLOAT) `sim-encoder.<chan-num>.scale` – Scale Factor for **speed**. The default is 1.0, which means that **speed** is in revolutions per second. Change to 60 for RPM, to 360 for degrees per second, 6.283185 for radians per seconed, etc.

Note that pulses per revolution is not the same as counts per revolution. A pulse is a complete quadrature cycle. Most encoder counters will count four times during one complete cycle.

7.6.5 Functions

The component exports two functions. Each function affects all simulated encoders.

- (FUNCT) `sim-encoder.make-pulses` – High speed function to generate quadrature pulses (no floating point).
- (FUNCT) `sim-encoder.update-speed` – Low speed function to read **speed**, do scaling, and set up **make-pulses**.

7.7 Debounce

Debounce is a realtime component that can filter the glitches created by mechanical switch contacts. It may also be useful in other applications where short pulses are to be rejected.

7.7.1 Installing

```
emc2$ halcmd loadrt debounce cfg="<config-string>"
```

<config-string> is a series of space separated decimal integers. Each number installs a group of identical debounce filters, the number determines how many filters are in the group. For example:

```
emc2$ halcmd loadrt debounce cfg="1 4 2"
```

will install three groups of filters. Group 0 contains one filter, group 1 contains four, and group 2 contains two filters. The default value for <config-string> is "1" which will install a single group containing a single filter. The maximum number of groups 8 (as defined by MAX_GROUPS in debounce.c). The maximum number of filters in a group is limited only by shared memory space. Each group is completely independent. All filters in a single group are identical, and they are all updated by the same function at the same time. In the following descriptions, <G> is the group number and <F> is the filter number within the group. The first filter is group 0, filter 0.

7.7.2 Removing

```
emc2$ halcmd unloadrt debounce
```

7.7.3 Pins

Each individual filter has two pins.

- (BIT) debounce.<G>.<F>.in – Input of filter <F> in group <G>.
- (BIT) debounce.<G>.<F>.out – Output of filter <F> in group <G>.

7.7.4 Parameters

Each group of filters has one parameter[2].

- (s32) debounce.<G>.delay – Filter delay for all filters in group <G>.

The filter delay is in units of thread periods. The minimum delay is zero. The output of a zero delay filter exactly follows its input - it doesn't filter anything. As delay increases, longer and longer glitches are rejected. If delay is 4, all glitches less than or equal to four thread periods will be rejected.

7.7.5 Functions

Each group of filters has one function, which updates all the filters in that group "simultaneously". Different groups of filters can be updated from different threads at different periods.

- (FUNCT) debounce.<G> – Updates all filters in group <G>.

[2]Each individual filter also has an internal state variable. There is a compile time switch that can export that variable as a parameter. This is intended for testing, and simply wastes shared memory under normal circumstances.

7.8 Blocks

Blocks is a realtime HAL component containing a number of simple functional blocks. They are primarily intended for testing and other offhand uses, but may prove useful for some EMC configurations as well.

7.8.1 Available Blocks

The complete list of available blocks is as follows:

- constant – makes a parameter value available on a pin
- comp – standard 2-input comparator with adjustable hysteresis
- wcomp – window comparator, output true when input is between adjustable upper and lower limits
- sum2 – 2-input summer, output = in0 * gain0 + in1 * gain1
- mux2 – 2-input multiplexor, output = in0 if sel is false, output = in1 if sel is true
- mux4 – 4-input multiplexor, output = in<n> based on sel0, sel1
- integ – integrator, input of +1.0 produces an output ramp of +1.0 per second
- ddt – differentiator, input ramp of +1.0 per second produces output of +1.0
- limit1 – first order limiter (limits output)
- limit2 – second order limiter (limits output and 1st derivative)
- limit3 – third order limiter (limits ouput, 1st & 2nd derivative)
- estop – latch for estops or other faults, with reset
- not – logical inverter
- and2 – 2 input logical and
- or2 – 2 input logical or
- scale – gain/offset block (out = in * gain + offset)
- lowpass – lowpass filter (out = last_out * (1 - gain) + in * gain
- match8 – 8 bit binary match detector (with input for cascading)

7.8.2 Installing

```
emc2$ halcmd loadrt blocks <blockname>=<number>
```

`<blockname>` is the name of one of the functional blocks that the component can provide, and `<number>` is the desired quantity of that block. Multiple name/number pairs may be specified on the command line to install a variety of different blocks. For example:

```
emc2$ halcmd loadrt blocks wcomp=2 sum2=1 comp=3
```

installs two window comparators (wcomp), one two-input summer (sum2), and three standard comparators (comp).

7.8.3 Removing

```
emc2$ halcmd unloadrt blocks
```

7.8.4 Pins

The various blocks each have their own input and output pins. Each pin begins with the block name, then the block number, and finally the pin function.

- (FLOAT) constant.<num>.out – Output of constant block, tracks the parameter value.
- (FLOAT) comp.<num>.in0 – Inverting input of 2-input comparator.
- (FLOAT) comp.<num>.in1 – Non-inverting input of 2-input comparator.
- (BIT) comp.<num>.out – Output, true if in1 > in0.
- (FLOAT) wcomp.<num>.in –Input of window comparator.
- (BIT) wcomp.<num>.out – Output, TRUE if min < input < max.
- (FLOAT) sum2.<num>.in0 – First input of 2-input summer.
- (FLOAT) sum2.<num>.in1 – Second input of 2-input summer.
- (FLOAT) sum2.<num>.out – Output, equals in0 * gain0 + in1 * gain1.
- (FLOAT) mux2.<num>.in0 – First input of 2-input multiplexor.
- (FLOAT) mux2.<num>.in1 –Second input of 2-input multiplexor.
- (BIT) mux2.<num>.sel – Control input of 2-input multiplexor.
- (FLOAT) mux2.<num>.out – Output, equals in0 if sel is FALSE, in1 if sel is TRUE.
- (FLOAT) integ.<num>.in – Input of integrator.
- (FLOAT) integ.<num>.out – Output of integrator.
- (FLOAT) ddt.<num>.in – Input of differentiator.
- (FLOAT) ddt.<num>.out – Output of differentiator.
- ... and so on (too many pins to list them all here, but you can easily see them with halcmd show pin)

7.8.5 Parameters

Some of the blocks have parameters that control and adjust the details of their operation. Like the pin names, they begin with the block name and the block number, then the parameter name.

- (FLOAT) constant.<num>.value – The value to be written to the out pin, default value is 1.0.
- (FLOAT) comp.<num>.hyst – Hystersis for 2-input comparator, default is 0.0.
- (FLOAT) wcomp.<num>.min – Minimum threshold for window comparator, default is -1.0.
- (FLOAT) wcomp.<num>.max – Maximum threshold for window comparator, default is +1.0.
- (FLOAT) sum2.<num>.gain0 – Value by which in0 is multiplied, default is 1.0.
- (FLOAT) sum2.<num>.gain1 – Value by which in1 is multiplied, default is 1.0.

7.8.6 Functions

Each individual block has its own function. This allows complete control over when each block executes. In general, blocks should execute in the order of signal flow. If the outputs of blocks A and B are connected to inputs of block C, then the functions for A and B should be executed before the function for C. Note that unless these functions are connected to a realtime thread so that they execute, the blocks do nothing at all.

- (FUNCT) `constant.<num>` – Writes parameter `value` to pin `out`.
- (FUNCT) `comp.<num>` – Compares `in0` and `in1` (with hysteresis), writes result to `out`.
- (FUNCT) `wcomp.<num>` – Compares `in` to `min` and `max`, writes result to `out`.
- (FUNCT) `sum2.<num>` – Computes `out = in0 * gain0 + in1 * gain1`.
- (FUNCT) `mux2.<num>` – If `sel` is TRUE, writes `in1` to `out`, else writes `in0` to `out`.
- (FUNCT) `integ.<num>` – Calculates integral of `in`, writes result to `out`.
- (FUNCT) `constant.<num>` – Calculates derivative of `in`, writes result to `out`.

7.9 Siggen

Siggen is a realtime component that generates square, triangle, and sine waves. It is primarily used for testing.

7.9.1 Installing

```
emc2$ halcmd loadrt siggen [num_chan=<chans>]
```

<chans> is the number of signal generators that you want to install. If numchan is not specified, one signal generator will be installed. The maximum number of generators is 16 (as defined by MAX_CHAN in siggen.c). Each generator is completely independent. In the following descriptions, <chan> is the number of a specific signal generator (the numbers start at 0).

7.9.2 Removing

```
emc2$ halcmd unloadrt siggen
```

7.9.3 Pins

Each generator has five output pins.

- (FLOAT) siggen.<chan>.sine – Sine wave output.
- (FLOAT) siggen.<chan>.cosine – Cosine output.
- (FLOAT) siggen.<chan>.sawtooth – Sawtooth output.
- (FLOAT) siggen.<chan>.triangle – Triangle wave output.
- (FLOAT) siggen.<chan>.square – Square wave output.

All five outputs have the same frequency, amplitude, and offset.
In addition to the output pins, there are three control pins:

- (FLOAT) siggen.<chan>.frequency – Sets the frequency in Hertz, default value is 1 Hz.
- (FLOAT) siggen.<chan>.amplitude – Sets the peak amplitude of the output waveforms, default is 1.
- (FLOAT) siggen.<chan>.offset – Sets DC offset of the output waveforms, default is 0.

For example, if siggen.0.amplitude is 1.0 and siggen.0.offset is 0.0, the outputs will swing from -1.0 to +1.0. If siggen.0.amplitude is 2.5 and siggen.0.offset is 10.0, then the outputs will swing from 7.5 to 12.5.

7.9.4 Parameters

None. [3]

7.9.5 Functions

- (FUNCT) siggen.<chan>.update – Calculates new values for all five outputs.

[3] Prior to version 2.1, frequency, amplitude, and offset were parameters. They were changed to pins to allow control by other components.

Chapter 8

Virtual Control Panels

8.1 Introduction

Traditional machine control panels are large sheets of steel with pushbuttons, knobs, lights and sometimes meters mounted on them. They have many advantages - the buttons are far more rugged than a computer keyboard, and large enough that you can usually operate the correct one by feel while looking elsewhere, for example at the tool. However, they also have disadvantages. The occupy a lot of panel space, they are expensive, and wiring them into the PC can use up a lot of I/O pins. That is where Virtual Control Panels come in.

A Virtual Control Panel (VCP) is a window on the computer screen with buttons, meters, switches, etc. When you click on a VCP button, it changes the state of a HAL pin, exactly as if you had pressed a physical button wired to an input pin on an I/O card. Likewise, a VCP LED lights up when a HAL pin goes true, just like a physical indicator lamp wired to an output pin on an I/O card. Virtual control panels are not intended to replace physical panels - sometimes there is just no substiture for a big rugged oil-tight pushbutton. But virtual panels can be used for testing or monitoring things that don't require physical buttons and lights, to temporarily replace real I/O devices while debugging ladder logic, or perhaps to simulate a physical panel before you build it and wire it to an I/O board.

Currently there are two VCP implementations included with EMC2: The older, simply named VCP, which used GTK widgets, and the newer, called pyVCP, which uses Tkinter widgets. VCP is deprecated and should not be used - it may be removed in the future.

8.2 pyVCP

The layout of a pyVCP panel is specified with an XML file that contains widget tags between <pyvcp> and </pyvcp>. For example:

```
<pyvcp>
    <label>
        <text>"This is a LED indicator"</text>
    </label>
    <led>
    </led>
</pyvcp>
```

If you place this text in a file called tiny.xml, and run

```
pyvcp -c mypanel tiny.xml
```

pyVCP will create the panel for you, which includes two widgets, a Label with the text "This is a LED indicator", and a LED, used for displaying the state of a HAL BIT signal. It will also create a HAL component named "mypanel" (all widgets in this panel are connected to pins that start with "mypanel."). Since no <halpin> tag was present inside the <led> tag, pyVCP will automatically name the HAL pin for the LED widget mypanel.led.0

For a list of widgets and their tags and options, see the widget reference below.

Once you have created your panel, connecting HAL signals to and form the pyVCP pins is done with 'halcmd linksp' as usual. If you are new to HAL, the HAL Tutorial2 is recommended.

8.3 Using pyVCP with AXIS

Since AXIS uses the same GUI toolkit (Tkinter) as pyVCP, it is possible to include a pyVCP panel on the right side of the normal AXIS user interface. A typical example is explained below.

Place your pyVCP XML file describing the panel in the same directory where your .ini file is. Say we we want to display the current spindle speed using a Bar widget. Place the following in a file called spindle.xml:

```
<pyvcp>
    <label>
        <text>"Spindle speed:"</text>
    </label>
    <bar>
        <halpin>"spindle-speed"</halpin>
        <max_>5000</max_>
    </bar>
</pyvcp>
```

Here we've made a panel with a Label and a Bar widget, specified that the HAL pin connected to the Bar should be named "spindle-speed", and set the maximum value of the bar to 5000 (see widget reference below for all options). To make AXIS aware of this file, and call it at startup, we need to specify the following in the [DISPLAY] section of the .ini file:

```
PYVCP = spindle.xml
```

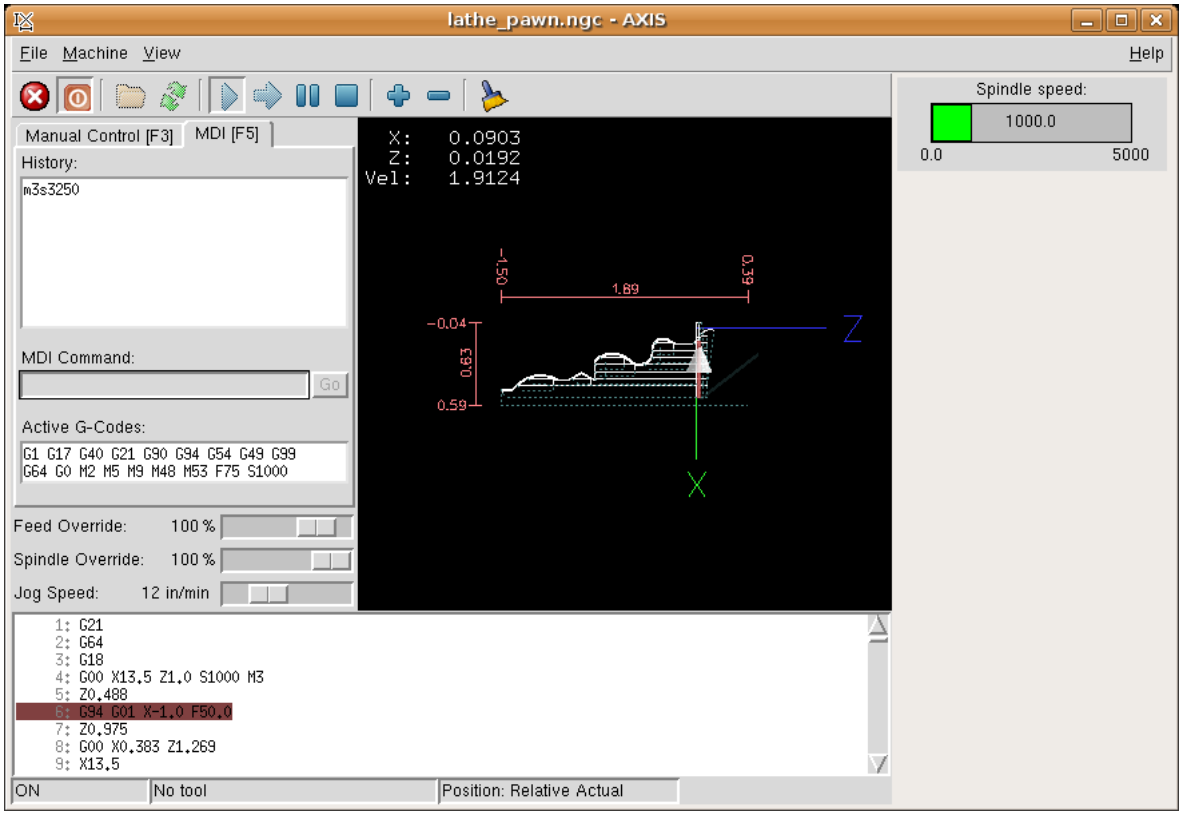
To make our widget actually display the spindle-speed it needs to be hooked up to the appropriate HAL signal. A .hal file that will be run once AXIS and pyVCP have started can be specified in the [HAL] section of the .ini file:

```
POSTGUI_HALFILE = spindle_to_pyvcp.hal
```

This change will run the HAL commands specified in "spindle_to_pyvcp.hal". In our example the contents could look like this:

```
linksp spindle-rpm-filtered  pyvcp.spindle-speed
```

assuming that a singlal called "spindle-rpm-filtered" already exists. Note that when running together with AXIS, all pyVCP widget HAL pins have names that start with "pyvcp.".



This is what the newly created pyVCP panel should look like in AXIS.

8.4 pyVCP Widget reference

HAL signals come in two variants, BIT and FLOAT. pyVCP can either display the value of the signal with an indicator widget, or modify the signal value with a control widget. Thus there are four classes of pyVCP widgets that you can connect to a HAL signal. A fifth class of helper widgets allow you to organize and label your panel.

1. Widgets for indicating BIT signals: LED
2. Widgets for controlling BIT signals: Button, Checkbutton, Radiobutton
3. Widgets for indicating FLOAT signals: Number, Bar, Meter
4. Widgets for controlling FLOAT signals: Spinbox, Scale, Jogwheel
5. Helper widgets: Hbox, Vbox, Label

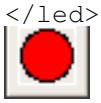
8.4.0.1 Syntax

Each widget is described briefly, followed by the markup used, and a screenshot. All tags inside the main widget tag are optional.

8.4.1 LED

A LED is used to indicate the status of a BIT signal. The LED color will be on_color when the BIT signal is true, and off_color otherwise.

```
<led>
    <halpin>"my-led"</halpin>
    <size>50</size>
    <on_color>"blue"</on_color>
    <off_color>"black"</off_color>
</led>
```



<halpin> sets the name of the pin, default is “led.n”, where n is an integer
<size> sets the size of the led, default is 20
<on_color> sets the color of the LED when the pin is true. default is “green”
<off_color> sets the color of the LED when the pin is false. default is “ref”

8.4.2 Button

A button is used to control a BIT pin. The pin will be set True when the button is pressed and held down, and will be set False when the button is released.

```
<button>
    <halpin>"my-button"</halpin>
    <text>"ON"</text>
</button>
```

8.4.3 Checkbutton

A checkbutton controls a BIT pin. The pin will be set True when the button is checked, and false when the button is unchecked.

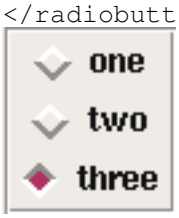
```
<checkbutton>
    <halpin>"my-checkbutton"</halpin>
</checkbutton>
```

An unchecked checkbutton: , and a checked one:

8.4.4 Radiobutton

A radiobutton will set one of a number of BIT pins true. The other pins are set false.

```
<radiobutton>
    <choices>["one","two","three"]</choices>
    <halpin>"my-radio"</halpin>
</radiobutton>
```

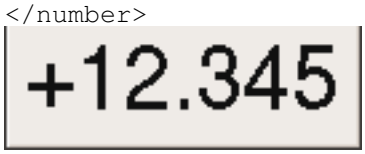


Note that the HAL pins in the example above will me named my-radio.one, my-radio.two, and my-radio.three

8.4.5 Number

The number widget displays the value of a FLOAT signal.

```
<number>
    <halpin>"my-number"</halpin>
    <font>('Helvetica',50)</font>
    <format>"+4.3f"</format>
</number>
```



<font> is a Tkinter font type and size specification. Note that on Ubuntu 6.06 'Helvetica' is not available in sizes above ca 40 or 50. One font that will show up to at least size 200 is 'courier 10 pitch', so for a really big Number widget you could specify:

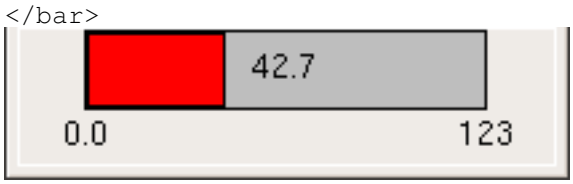
```
<font>('courier 10 pitch',100)</font>
```

<format> is a 'C-style' format specified that determines how the number is displayed.

8.4.6 Bar

A bar widget displays the value of a FLOAT signal both graphically using a bar display and numerically.

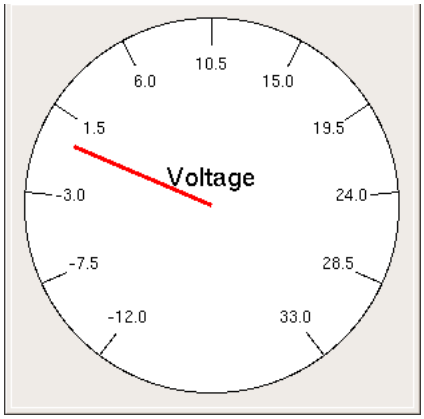
```
<bar>
    <halpin>"my-bar"</halpin>
    <min_>0</min_>
    <max_>123</max_>
    <bgcolor>"grey"</bgcolor>
    <fillcolor>"red"</fillcolor>
</bar>
```



8.4.7 Meter

Meter displays the value of a FLOAT signal using a traditional dial indicator.

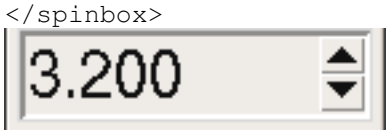
```
<meter>
    <halpin>"my-meter"</halpin>
    <text>"Voltage"</text>
    <size>300</size>
    <min_>-12</min_>
    <max_>33</max_>
</meter>
```



8.4.8 Spinbox

Spinbox controls a FLOAT pin. You increase or decrease the value of the pin by 'resolution' by either pressing on the arrows, or pointing at the spinbox and rolling your mouse-wheel.

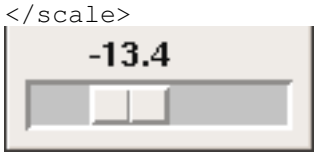
```
<spinbox>
    <halpin>"my-spibox"</halpin>
    <min_>-12</min_>
    <max_>33</max_>
    <resolution>0.1</resolution>
    <format>"2.3f"</format>
    <font>('Arial',30)</font>
</spinbox>
```



8.4.9 Scale

Scale controls a FLOAT pin. You increase or decrease the value of the pin be either dragging the slider, or pointing at the scale and rolling your mouse-wheel.

```
<scale>
    <halpin>"my-scale"</halpin>
    <resolution>0.1</resolution>
    <orient>HORIZONTAL</orient>
    <min_>-33</min_>
    <max_>26</max_>
</scale>
```

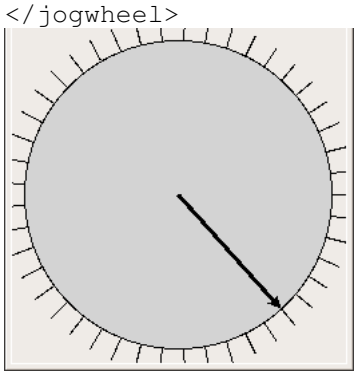


8.4.10 Jogwheel

Jogwheel mimics a real jogwheel by outputting a FLOAT pin which counts up or down as the wheel is turned, either by dragging in a cricular motion, or by rolling the mouse-wheel.

```
<jogwheel>
```

```
    <halpin>"my-wheel"</halpin>
    <cpr>45</cpr>
    <size>250</size>
</jogwheel>
```



8.4.11 Hbox

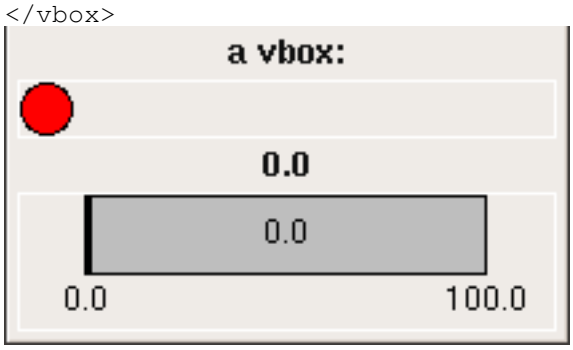
Use a Hbox when you want to stack widgets horizontally next to eachother.

```
<hbox>
    <label><text>"a vbox:"</text></label>
    <led></led>
    <number></number>
    <bar></bar>
</hbox>
```

8.4.12 Vbox

Use a Vbox when you want to stack widgets verticallyon top of eachother.

```
<vbox>
    <label><text>"a vbox:"</text></label>
    <led></led>
    <number></number>
    <bar></bar>
</vbox>
```



8.4.13 Label

A label is a piece of text on your panel.

```
<label>
    <text>"This is a Label:"</text>
    <font>('Helvetica',20)</font>
</label>
```

8.5 VCP: A small example

NOTE: VCP is deprecated, and will most likely not be getting any new development or additional widgets. We strongly recommend using pyVCP. Both pyVCP and VCP are included in EMC version 2.1, but it is likely that only pyVCP will be included in EMC 2.2. [1]

Place the following in the file `tiny.vcp`:

```
vcp {
  main-window {
    box {
      button {
        halpin = vcp.pushbutton
        label { text = "Push Me" }
      }
      LED {
        halpin = vcp.light
      }
    }
  }
}
```

The above file describes a tiny Virtual Control Panel, with one push button, and one light. To see what it looks like, we need to start HAL:

```
$halrun
halcmd:
```

Next we load halvcp, and give it the name of our .vcp file:

```
halcmd: loadusr halvcp tiny.vcp
halcmd:
```

There may be some text printed as halvcp parses the tiny.vcp file, but when it finishes, there should be a small window on your screen, with a button and an LED. It will look something like figure 8.1.

Figure 8.1: tiny.vcp on the screen

So, we have a button and an LED, but they aren't connected to anything, so nothing happens when you push the button. However, the LED and the button both have HAL pins associated with them:

[1] A .vcp to .xml translator that takes a vcp file and turns it into one that pyVCP can use is on my to-do list. That would enable VCP users to easily switch over to pyVCP. If such a translator is written, VCP may be removed from the version 2.2 release.

```
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 03    bit   IN      FALSE      vcp.light
 03    bit   OUT     FALSE      vcp.pushbutton
halcmd:
```

To make something happen, we can connect a HAL signal between the button and the light:

```
halcmd: newsig jumper bit
halcmd: linksp jumper vcp.pushbutton
halcmd: linksp jumper vcp.light
halcmd: show sig
Signals:
Type      Value      Name
bit       FALSE      jumper
                         ==> vcp.light
                         <== vcp.pushbutton
halcmd:
```

Now push the button, and the the LED should light up!

8.6 VCP: Another small example with EMC

Place the following in the file `estop.vcp`:

```
vcp {
   main-window {
      toggle { halpin = vcp.estop }
   }
}
```

In your .hal file, remove any existing signal linked to `iocontrol.0.emc-enable-in` and add the following lines:

```
loadusr -W halvcp estop.vcp
newsig estop bit
linkps vcp.estop => estop
linkps estop => iocontrol.0.emc-enable-in
```

Now, when running your machine, the ESTOP button in the GUI is disabled, and the ESTOP button in the VCP window is used instead.

8.7 VCP Syntax

8.7.1 Block

A block's format is:

```
tag { contents }
```

The contents can consist of attributes that describe the block, or other blocks that nest inside it.
A attribute's format is

```
name = value
```

The attribute names that are acceptable for each block depend on the block tag, and will be listed later.

Part III

HAL Programming

Chapter 9

comp: a tool for creating HAL modules

9.1 Introduction

Writing a HAL component can be a tedious process, most of it in setup calls to `rtapi_` and `hal_` functions and associated error checking. *comp* will write all this code for you, automatically.

Compiling a realtime HAL component is also much easier when using *comp*, whether the component is part of the emc2 source tree, or outside it.

For instance, the "ddt" portion of `blocks` is around 80 lines of code. The equivalent component is very short when written using the *comp* preprocessor:

```
component ddt "Compute the derivative of the input function";
pin in float in;
pin out float out;
option data float;
function _;
;;
MODULE_LICENSE("GPL");
FUNCTION(_) {
    float tmp = in;
    out = (tmp - data) / fperiod;
    data = tmp;
}
```

and it can be compiled and installed very easily: by simply placing `ddt.comp` in `src/hal/components` and running '`make`', or by placing it anywhere on the system and running `comp --install ddt.comp`

9.2 Definitions

component A component is a single real-time module, which is loaded with `halcmd loadrt`. One `.comp` file specifies one component.

instance A component can have zero or more instances. Each instance of a component is created equal (they all have the same pins, parameters, functions, and data) but behave independently when their pins, parameters, and data have different values.

singleton It is possible for a component to be a 'singleton', in which case exactly one instance is created. It seldom makes sense to write a `singleton` component, unless there can literally only be a single object of that kind in the system (for instance, a component whose purpose is to provide a pin with the current UNIX time, or a hardware driver for the internal PC speaker)

9.3 Instance creation

For a singleton, the one instance is created when the component is loaded.

For a non-singleton, the 'count' module parameter determines how many numbered instances are created.

9.4 Syntax

A .comp file consists of a number of declarations, followed by ;; on a line of its own, followed by C code implementing the module's realtime functions.

Declarations include:

```
component NAME [DOC];
pin PINDIRECTION TYPE NAME [DOC];
param PARAMDIRECTION TYPE NAME [= STARTVALUE] [DOC] ;
function NAME [fp | nofp] [DOC];
option NAME [VALUE];
```

Brackets indicate optional items. A vertical bar indicates alternatives. Words in *CAPITALS* indicate variable text, as follows:

NAME A C identifier.
When used to create a HAL identifier, any underscores are replaced with dashes, and any trailing dash is removed, so that "this_name_" will be turned into "this-name". If the name is "_", then a trailing period is removed as well, so that "function _" gives a HAL function name like component.<num> instead of component.<num>.
If present, the prefix hal_ is removed from the beginning of the component name when creating pins, parameters and functions.

DOC A string that documents the item. String can be a C-style "double quoted" string, like "Selects the desired edge: TRUE means falling, FALSE means rising" or a Python-style "triple quoted" string, which may include embedded newlines and quote characters, such as:

```
param rw bit zot=TRUE
"""The effect of this parameter, also known as "the orb of zot",
will require at least two paragraphs to explain.

Hopefully these paragraphs have allowed you to understand "zot"
better.""";
```

TYPE One of the HAL types: bit, s32, u32, or float.

PINDIRECTION One of the following: in, out, or io. A component sets a value for an out pin, it reads a value from an in pin, and it may read or set the value of an io pin.

PARAMDIRECTION One of the following: r or rw. A component sets a value for a r parameter, and it may read or set the value of a rw parameter.

STARTVALUE Specifies the initial value of a parameter. If it is not specified, then the default is 0 or FALSE, depending on the type of the parameter.

fp Indicates that the function performs floating-point calculations.

nofp Indicates that it only performs integer calculations. If neither is specified, fp is assumed. Neither comp nor gcc can detect the use of floating-point calculations in functions that are tagged nofp.

VALUE Depending on the option name, the valid VALUEs vary. The currently defined options are:

option singleton yes (default: no)
Do not create a `count` module parameter, and always create a single instance. With `singleton`, items are named `component-name.item-name` and without `singleton`, items for numbered instances are named `component-name.<num>.item-name`.

option default_count (default: 1)
Normally, the module parameter `count` defaults to 0. If specified, the `count` will default to this value instead.

option count_function yes (default: no)
Normally, the number of instances to create is specified in the module parameter `count`; if `count_function` is specified, the value returned by the function `int get_count(void)` is used instead, and the `count` module parameter is not defined.

option rtapi_app no (default: yes)
Normally, the functions `rtapi_app_main` and `rtapi_app_exit` are automatically defined. With `option rtapi_app no`, they are not, and must be provided in the C code.
When implementing your own `rtapi_app_main`, call the function `int export(char *prefix, long extra_arg)` to register the pins, parameters, and functions for `prefix`.

option data *type* (default: none)
If specified, each instance of the component will have an associated data block of *type* (which can be a simple type like `float` or the name of a type created with `typedef`).

option extra_setup yes (default: no)
If specified, call the function defined by `EXTRA_SETUP` for each instance. If using the automatically defined `rtapi_app_main`, `extra_arg` is the number of this instance.

option extra_cleanup yes (default: no)
If specified, call the function defined by `EXTRA_CLEANUP` from the automatically defined `rtapi_app_exit`, or if an error is detected in the automatically defined `rtapi_app_main`.

If an option's VALUE is not specified, then it is equivalent to specifying `option ... yes`. The result of assigning an inappropriate value to an option is undefined. The result of using any other option is undefined.

C++-style one-line comments (`// ...`) and C-style multi-line comments (`/* ... */`) are both supported in the declaration section.

9.5 Other restrictions on comp files

Though HAL permits a pin, a parameter, and a function to have the same name, comp does not.

9.6 Convenience Macros

Based on the items in the declaration section, *comp* creates a C structure called `struct state`. However, instead of referring to the members of this structure (e.g., `*(inst->name)`), they will generally be referred to using the macros below. The details of `struct state` and these macros may change from one version of *comp* to the next.

FUNCTION(name) Use this macro to begin the definition of a realtime function which was previously declared with '`function NAME`'. The function includes a parameter '`period`' which is the integer number of nanoseconds between calls to the function.

EXTRA_SETUP() Use this macro to begin the definition of the function called to perform extra setup of this instance. Return a negative Unix `errno` value to indicate failure (e.g., `return -EBUSY` on failure to reserve an I/O port), or 0 to indicate success.

EXTRA_CLEANUP() Use this macro to begin the definition of the function called to perform extra cleanup of the component. Note that this function must clean up all instances of the component, not just one. The 'pin_name', 'parameter_name', and 'data' macros may not be used here.

pin_name

parameter_name For each pin `pin_name` or param `parameter_name` there is a macro which allows the name to be used on its own to refer to the pin or parameter.

data If 'option data' is specified, this macro allows access to the instance data.

fperiod The floating-point number of seconds between calls to this function.

9.7 Compiling `.comp` files in the source tree

Place the `.comp` file in the source directory `emc2/src/hal/components` and re-run `make`. `Comp` files are automatically detected by the build system.

If a `.comp` file is a driver for hardware, it may be placed in `emc2/src/hal/components` and will be built except if emc2 is configured as a userspace simulator.

9.8 Compiling components outside the source tree

`comp` can process, compile, and install a component in a single step, placing `example.ko` in the emc2 realtime module directory:

```
comp --install example.comp
```

Or, it can process and compile in one step, leaving `example.ko` in the current directory:

```
comp --compile example.comp
```

Or it can simply process, leaving `example.c` in the current directory:

```
comp example.comp
```

`comp` can also compile and install a component written in C, using the `--install` and `--compile` options shown above:

```
comp --install example2.c
```

man-format documentation can also be created from the information in the declaration section:

```
comp --document example.comp
```

The resulting manpage, `example.9` can be viewed with

```
man ./example.9
```

or copied to a standard location for manual pages.

9.9 Examples

9.9.1 constant

This component functions like the one in 'blocks', including the default value of 1.0. The declaration "function _" creates functions named 'constant.0', etc.

```
component constant;
pin out float out;
param r float value = 1.0;
function _;
option extra_setup yes;
;;
FUNCTION(_) { out = value; }
```

9.9.2 sincos

This component computes the sine and cosine of an input angle in radians. It has different capabilities than the 'sine' and 'cosine' outputs of siggen, because the input is an angle, rather than running freely based on a 'frequency' parameter.

The pins are declared with the names `sin_` and `cos_` in the source code so that they do not interfere with the functions `sin()` and `cos()`. The HAL pins are still called `sincos.<num>.sin`.

```
component sincos;
pin out float sin_ out;
pin out float cos_ out;
pin in float theta in;
function _;
;;
#include <rtapi_math.h>
FUNCTION(_) { sin_ = sin(theta); cos_ = cos(theta); }
```

9.9.3 out8

This component is a driver for a *fictional* card called "out8", which has 8 pins of digital output which are treated as a single 8-bit value. There can be a varying number of such cards in the system, and they can be at various addresses. The pin is called `out_` because `out` is an identifier used in `<asm/io.h>`. It illustrates the use of `EXTRA_SETUP` and `EXTRA_CLEANUP` to request an I/O region and then free it in case of error or when the module is unloaded.

```
component out8;
pin out u32 out_ "Output value; only low 8 bits are used";
param r u32 ioaddr;

function _;

option count_function;
option extra_setup;
option extra_cleanup;
option constructable no;

;;
#include <asm/io.h>

#define MAX 8
```

```
int io[MAX] = {0,};
RTAPI_MP_ARRAY_INT(io, MAX, "I/O addresses of out8 boards");

int get_count(void) {
    int i = 0;
    for(i=0; i<MAX && io[i]; i++) { /* Nothing */ }
    return i;
}

EXTRA_SETUP() {
    if(!rtapi_request_region(io[extra_arg], 1, "out8")) {
// set this I/O port to 0 so that EXTRA_CLEANUP does not release the IO
// ports that were never requested.
        io[extra_arg] = 0;
        return -EBUSY;
    }
    ioaddr = io[extra_arg];
    return 0;
}

EXTRA_CLEANUP() {
    int i;
    for(i=0; i < MAX && io[i]; i++) {
        rtapi_release_region(io[i], 1);
    }
}

FUNCTION(_) { outb(out_, ioaddr); }
```

9.9.4 hal_loop

```
component hal_loop;
pin out float example;
```

This fragment of a component illustrates the use of the `hal_` prefix in a component name. `loop` is the name of a standard Linux kernel module, so a `loop` component might not successfully load if the Linux `loop` module was also present on the system.

When loaded, `halcmd show comp` will show a component called `hal_loop`. However, the pin shown by `halcmd show pin` will be `loop.0.example`, not `hal-loop.0.example`.

Chapter 10

Creating Userspace Python Components with the 'hal' module

10.1 Basic usage

A userspace component begins by creating its pins and parameters, then enters a loop which will periodically drive all the outputs from the inputs. The following component copies the value seen on its input pin (`passthrough.in`) to its output pin (`passthrough.out`) approximately once per second.

```
#!/usr/bin/python
import hal, time
h = hal.component("passthrough")
h.newpin("in", hal.HAL_FLOAT, hal.HAL_IN)
h.newpin("out", hal.HAL_FLOAT, hal.HAL_OUT)
h.ready()
try:
    while 1:
        time.sleep(1)
        h['out'] = h['in']
except KeyboardInterrupt:
    raise SystemExit
```

Copy the above listing into a file named "passthrough", make it executable (`chmod +x)`, and place it on your `$PATH`. Then try it out:

```
$ halrun
halcmd: loadusr passthrough
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 03    float IN              0  passthrough.in
 03    float OUT             0  passthrough.out
halcmd: setp passthrough.in 3.14
halcmd: show pin
Component Pins:
Owner  Type  Dir     Value      Name
 03    float IN           3.14  passthrough.in
 03    float OUT          3.14  passthrough.out
```

10.2 Userspace components and delays

If you typed "show pin" quickly, you may see that `passthrough.out` still had its old value of 0. This is because of the call to 'time.sleep(1)', which makes the assignment to the output pin occur at most once per second. Because this is a userspace component, the actual delay between assignments can be much longer–for instance, if the memory used by the passthrough component is swapped to disk, the assignment could be delayed until that memory is swapped back in.

Thus, userspace components are suitable for user-interactive elements such as control panels (delays in the range of milliseconds are not noticed, and longer delays are acceptable), but not for sending step pulses to a stepper driver board (delays must always be in the range of microseconds, no matter what).

10.3 Create pins and parameters

```
h = hal.component("passthrough")
```

The component itself is created by a call to the constructor '`hal.component`'. The arguments are the HAL component name and (optionally) the prefix used for pin and parameter names. If the prefix is not specified, the component name is used.

```
h.newpin("in", hal.HAL_FLOAT, hal.HAL_IN)
```

Then pins are created by calls to methods on the component object. The arguments are: pin name suffix, pin type, and pin direction. For parameters, the arguments are: parameter name suffix, parameter type, and parameter direction.

Table 10.1: HAL Option Names

Pin and Parameter Types:	HAL_BIT	HAL_FLOAT	HAL_S32	HAL_U32
Pin Directions:	HAL_IN	HAL_OUT	HAL_IO	
Parameter Directions:	HAL_RO	HAL_RW		

The full pin or parameter name is formed by joining the prefix and the suffix with a ".", so in the example the pin created is called `passthrough.in`.

```
h.ready()
```

Once all the pins and parameters have been created, call the `.ready()` method.

10.3.1 Changing the prefix

The prefix can be changed by calling the `.setprefix()` method. The current prefix can be retrieved by calling the `.getprefix()` method.

10.4 Reading and writing pins and parameters

For pins and parameters which are also proper Python identifiers, the value may be accessed or set using the attribute syntax:

```
h.out = h.in
```

For all pins, whether or not they are also proper Python identifiers, the value may be accessed or set using the subscript syntax:

```
h['out'] = h['in']
```

10.4.1 Driving output (HAL_OUT) pins

Periodically, usually in response to a timer, all HAL_OUT pins should be "driven" by assigning them a new value. This should be done whether or not the value is different than the last one assigned. When a pin is connected to a signal, its old output value is not copied into the signal, so the proper value will only appear on the signal once the component assigns a new value.

10.4.2 Driving bidirectional (HAL_IO) pins

The above rule does not apply to bidirectional pins. Instead, a bidirectional pin should only be driven by the component when the component wishes to change the value. For instance, in the canonical encoder interface, the encoder component only sets the **index-enable** pin to **FALSE** (when an index pulse is seen and the old value is **TRUE**), but never sets it to **TRUE**. Repeatedly driving the pin **FALSE** might cause the other connected component to act as though another index pulse had been seen.

10.5 Exiting

A "`halcmd unload`" request for the component is delivered as a `KeyboardInterrupt` exception. When an unload request arrives, the process should either exit in a short time, or call the `.exit()` method on the component if substantial work (such as reading or writing files) must be done to complete the shutdown process.

10.6 Project ideas

- Create an external control panel with buttons, switches, and indicators. Connect everything to a microcontroller, and connect the microcontroller to the PC using a serial interface. Python has a very capable serial interface module called pyserial http://pyserial.sourceforge.net/ (Ubuntu package name "python-serial", in the universe repository)
- Attach a LCDProc http://lcdproc.omnipotent.net/-compatible LCD module and use it to display a digital readout with information of your choice (Ubuntu package name "lcdproc", in the universe repository)
- Create a virtual control panel using any GUI library supported by Python (gtk, qt, wxwindows, etc)

Appendix A

Legal Section

Handbook Copyright Terms

Copyright (c) 2000 LinuxCNC.org

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and one Back-Cover Text: "This EMC Handbook is the product of several authors writing for linuxCNC.org. As you find it to be of value in your work, we invite you to contribute to its revision and growth." A copy of the license is included in the section entitled "GNU Free Documentation License". If you do not find the license you may order a copy from Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307

A.1 GNU Free Documentation License Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

A.1.1 GNU Free Documentation License Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five). C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and

replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

blocks, 12

classicladder, 12
CNC, 9

encoder, 12

freqgen, 12

HAL, 9
HAL Component, 11
HAL Function, 11
HAL Parameter, 11
HAL Physical-Pin, 11
HAL Pin, 11
HAL Signal, 11
HAL Thread, 11
HAL Type, 11
hal-ax5214h, 12
hal-m5i20, 12
hal-motenc, 12
hal-parport, 12
hal-ppmc, 12
hal-stg, 12
hal-vti, 12
halcmd, 13
halmeter, 13
halscope, 13
halui, 12

iocontrol, 12

motion, 12

Parport, 52
pid, 12

siggen, 12
stepgen, 12
supply, 12