HAL for Integrators

1st May 2006

Contents

Part I

Introduction and 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 complicated 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.

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.

Part Selection

The machine builder does not need to worry 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.

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 get connected to the servo

6

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.

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.

Testing

Very few machines work right the first time. While testing the builder may use a meter to see if 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 equivalent of a voltmeter, oscilloscope, signal generator, and other tools for testing and tuning a system. The same commands used to build the system can be used to make changes as needed.

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 behaviour, 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 8 types[1], 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.

- U8 - an 8 bit unsigned integer, legal values are 0 to +255

- S8 - an 8 bit signed integer, legal values are -128 to +127

- U16 - a 16 bit unsigned integer, legal values are 0 to +65535

- S16 - a 16 bit signed integer, legal values are -32768 to +32767

- U32 - a 32 bit unsigned integer, legal values are 0 to +4294967295

- S32 - a 32 bit signed integer, legal values are -2147483648 to +2147483647

---

[1]There has been some discussion about whether we really need all the integer types. Maybe they will be reduced or eliminated later. Most signals and pins will be either floats or bits.

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.

For now a quick example will help get the concept across. 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 behaviour, that can be installed and interconnected as needed. This section lists available components and a brief description of what they do. Complete details for each component are available later in this document.

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 (note: right now experimental), it is intended to work as a full User Interface using external knobs & switches

Internal Components

stepgen Software step pulse generator with position loop. See section 6.1

freqgen Software step pulse generator. See section 6.2

encoder Software based encoder counter. See section 6.3

pid Proportional/Integral/Derivative control loops. See section 6.4

siggen A sine/cosine/triangle/square wave generator for testing. See section 6.8

supply a simple source for testing

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

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 5.1

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

Utilities

halcmd Command line tool for configuration and tuning. See section 7.1

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

halmeter A handy multimeter for HAL signals. See section 7.3

halscope A full featured digital storage oscilloscope for HAL signals. See section 7.4

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.

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.

Erector Sets[2]

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.

---

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

> 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.

Tinkertoys[3]

> 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.

---

[3]Tinkertoyŏ is now a registered trademark of the Hasbro company.

A Lego Example[4]

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.

> 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 it's 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.

---

[4]The Lego name is a trademark of the Lego company.

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 .

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.

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.

Root Privilges

In the beginning days of HAL there was quite often the need to run things with root privileges. Most of those things were related to the fact that HAL uses kernel modules to do much of it's work, and because it also can access hardware directly. Knowing that it is usually safer to avoid doing day-to-day work as root, most of those things were reworked so that very limited root privileges are required. If you are running a version of EMC2 and HAL more recent than early 2006, you can pretty much ignore this section.

To get around the need for root, emc2 uses a small program called emc_module_helper, which during the build process gets setuid status (thus has root privileges). This small software module takes care of the insmod/rmmod commands that are needed to insert/delete modules from the HAL. You can use this program directly, but it's lots easier to use the **halcmd loadrt & unloadrt** commands 7.1

Here is an example of what happens when you don't have root privileges:

```
emc2$ bin/hal_parport 0278
PARPORT: ERROR: could not get I/O permission
emc2$
```

As an alternative to logging in as root, you can use the `sudo` command or the `su -c` command. The `sudo` command is very convenient to use, and does not require you to know the root password. However, it needs to be configured by someone who does know the root password. The configuration

determines who may use `sudo`, and what commands they can use it for. We will not discuss `sudo` configuration here, try `man sudo` and/or talk to your system administrator. If sudo is properly configured, here is what happens:

```
emc2$ sudo bin/hal_parport 0278
Password: <enter your password>
PARPORT: installed driver for 1 ports
emc2$
```

As an added convenience, `sudo` remembers your password for a short time, so if you enter another `sudo` command within the time limit (usually 5 minutes) you don't have to type your password again.

The `su -c` command does not require configuration, but does require you to know the root password, and to type it in for every command. You also must put quotes around the command you are trying to run:

```
emc2$ su -c "bin/hal_parport 0278"
Password: <enter root password>
PARPORT: installed driver for 1 ports
emc2$
```

To avoid cluttering up the examples, we will not show `sudo` or `su -c`. Instead, commands that require root privileges will be preceded by #, and other commands will be preceded by by $.

```
emc2$ ls bin
emc2# bin/hal_parport 0278
```

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 (needs root privileges):

```
emc2# scripts/realtime start
```

With the realtime OS and RTAPI loaded, we can move into the first example.

2.2 A Simple Example

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 6.8 of this document. It is a realtime component, implemented as a Linux kernel module and located in the directory `emc2/rtlib/`. To load `siggen` use the `halcmd loadrt` command:

```
emc2$ bin/halcmd loadrt siggen
emc2$
```

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 7.1 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:

```
emc2$ bin/halcmd show comp
Loaded HAL Components:
ID  Type  Name
02  User  halcmd21345
01  RT    siggen
emc2$
```

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:

```
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir    Value        Name
 02    float -W   0.00000e+00  siggen.0.cosine
 02    float -W   0.00000e+00  siggen.0.sawtooth
 02    float -W   0.00000e+00  siggen.0.sine
 02    float -W   0.00000e+00  siggen.0.square
 02    float -W   0.00000e+00  siggen.0.triangle
emc2$
```

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:

```
emc2$ bin/halcmd show param
Parameters:
Owner  Type  Dir    Value        Name
 02    float  -W   1.00000e+00  siggen.0.amplitude
 02    float  -W   1.00000e+00  siggen.0.frequency
 02    float  -W   0.00000e+00  siggen.0.offset
 02    s32    R-             0  siggen.0.update.time
 02    s32    RW             0  siggen.0.update.tmax
emc2$
```

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

---

[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.

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:

```
emc2$ bin/halcmd show funct
Exported Functions:
Owner CodeAddr   Arg      FP   Users   Name
 02   C48E31C4 C48D2054 YES    0      siggen.0.update
emc2$
```

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

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):

```
emc2$ bin/halcmd loadrt threads name1=test-thread period1=1000000
```

Let's see if that worked:

```
emc2$ bin/halcmd show thread
Realtime Threads:
   Period   FP    Name     (Time, Max-Time)
    999849 YES  test-thread    ( 0, 0 )
emc2$
```

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:

```
emc2$ bin/halcmd addf siggen.0.update test-thread
emc2$
```

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:

```
emc2$ bin/halcmd show thread
Realtime Threads:
   Period   FP    Name     (Time, Max-Time)
    999849 YES  test-thread    ( 0, 0 )
                   1 siggen.0.update
emc2$
```

---

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

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:

```
emc2$ bin/halcmd start
emc2$
```

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

```
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir     Value       Name
  02    float -W    5.61498e-01  siggen.0.cosine
  02    float -W   -6.89775e-01  siggen.0.sawtooth
  02    float -W    8.27478e-01  siggen.0.sine
  02    float -W   -1.00000e+00  siggen.0.square
  02    float -W    3.79549e-01  siggen.0.triangle
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir     Value       Name
  02    float -W    9.23063e-01  siggen.0.cosine
  02    float -W   -8.74322e-01  siggen.0.sawtooth
  02    float -W    3.84649e-01  siggen.0.sine
  02    float -W   -1.00000e+00  siggen.0.square
  02    float -W    7.48645e-01  siggen.0.triangle
emc2$
```

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, and it happened to be at -1.0 for both commands.

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:

```
emc2$ bin/halcmd setp siggen.0.amplitude 5
emc2$
```

Check the parameters and pins again:

```
emc2$ bin/halcmd show param
Parameters:
Owner  Type  Dir     Value       Name
  02    float  -W    5.00000e+00  siggen.0.amplitude
  02    float  -W    1.00000e+00  siggen.0.frequency
  02    float  -W    0.00000e+00  siggen.0.offset
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir     Value       Name
  02    float -W    -1.66602e+00  siggen.0.cosine
  02    float -W     1.95935e+00  siggen.0.sawtooth
  02    float -W    -4.71428e+00  siggen.0.sine
```

```
   02    float -W    5.00000e+00  siggen.0.square
   02    float -W   -1.08130e+00  siggen.0.triangle
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir     Value       Name
   02    float -W   -3.82623e+00  siggen.0.cosine
   02    float -W   -1.11309e+00  siggen.0.sawtooth
   02    float -W    3.21869e+00  siggen.0.sine
   02    float -W   -5.00000e+00  siggen.0.square
   02    float -W   -2.77382e+00  siggen.0.triangle
emc2$
```

Note that the value of parameter `siggen.0.amplitude` has changed to 5.000, and that the pins now have larger values. The square wave output now switches from +5.0 to -5.0, and we happened to catch it switching this time.

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:

```
emc2$ bin/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
emc2$
```

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:

```
emc2$ bin/halcmd save >saved.hal
emc2$
```

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

2.3 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$ scripts/realtime start
emc2$ loadrt siggen
emc2$ loadrt threads name1=test-thread period1=1000000
emc2$ bin/halcmd addf siggen.0.update test-thread
emc2$ bin/halcmd start
emc2$ bin/halcmd setp siggen.0.amplitude 5
emc2$
```

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. We can start halmeter in the background by following it's name with a '&':

```
emc2$ bin/halmeter &
[1] 22093
emc2$
```

Since we started halmeter in the background, Linux prints its process id `[1] 22093` and immediately returns to the shell prompt. At the same time, a halcmd window opens on your screen, looking something like figure 2.1. Note that you don't have to run halmeter in the background. If you omit '&', it will start and behave exactly the same, but you won't get your shell prompt back until you exit from halmeter.
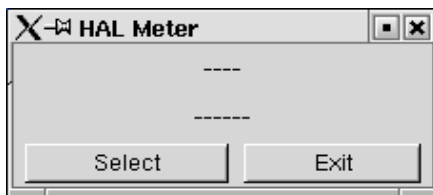
Figure 2.1: Halmeter at startup, nothing selected

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.
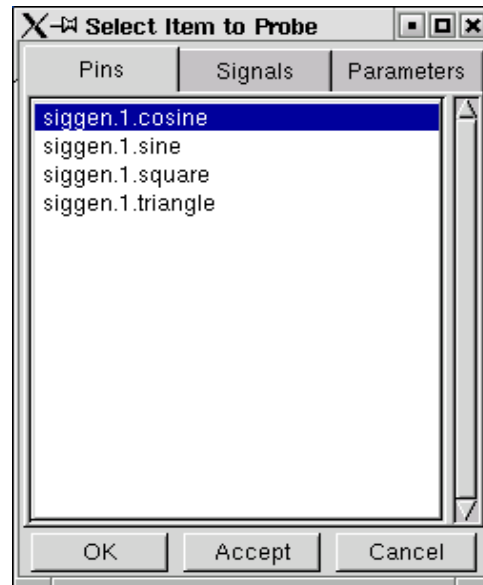
Figure 2.2: Halmeter source selection dialog
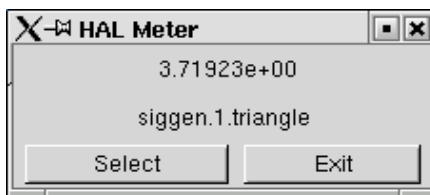
Figure 2.3: Halmeter displaying the value of a pin

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 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.

2.4   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:

```
emc2$ bin/halcmd unloadrt all
emc2$ scripts/realtime restart
emc2$
```

Installing the components

Now we are going to load the step pulse generator component. For a detailed description of this component refer to section 6.2. For now, we can skip the details, and just run the following commands:

```
emc2$ bin/halcmd loadrt freqgen step_type=0,0
emc2$ bin/halcmd loadrt siggen
emc2$ bin/halcmd loadrt threads name1=fast fp1=0 period1=50000 name2=slow period2=1000
emc2$
```

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:

```
emc2$ bin/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
emc2$ bin/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
02     u8     -W       1  (01)  freqgen.0.dirhold
02     u8     -W       1  (01)  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     u8     -W       1  (01)  freqgen.0.steplen
02     u8     -W       1  (01)  freqgen.0.stepspace
02     float  -W   1.00000e+00  freqgen.0.velocity-scale
02     u8     -W       1  (01)  freqgen.1.dirhold
02     u8     -W       1  (01)  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     u8     -W       1  (01)  freqgen.1.steplen
02     u8     -W       1  (01)  freqgen.1.stepspace
02     float  -W   1.00000e+00  freqgen.1.velocity-scale
emc2$
```

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:

```
emc2$ bin/halcmd newsig X_vel float
emc2$ bin/halcmd newsig Y_vel float
emc2$
```

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

```
emc2$ bin/halcmd show sig
Signals:
Type       Value        Name
float   0.00000e+00   X_vel
float   0.00000e+00   Y_vel
emc2$
```

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.

```
emc2$ bin/halcmd linksp X_vel siggen.0.cosine
emc2$
```

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

```
emc2$ bin/halcmd show sig
Signals:
Type      Value       Name
float    0.00000e+00  X_vel
                          <== siggen.0.cosine
float    0.00000e+00  Y_vel
emc2$
```

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:

```
emc2$ bin/halcmd linksp X_vel freqgen.0.velocity
emc2$
```

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:

```
emc2$ bin/halcmd linksp Y_vel siggen.0.sine
emc2$ bin/halcmd linksp Y_vel freqgen.1.velocity
emc2$
```

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

```
emc2$ bin/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
emc2$
```

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`.

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:

```
emc2$ bin/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
emc2$
```

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:

```
emc2$ bin/halcmd show thread
Realtime Threads:
   Period   FP    Name
   1005720 YES   slow     ( 0, 0 )
     50286 NO    fast     ( 0, 0 )
emc2$
```

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 `siggen.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:

```
emc2$ bin/halcmd addf siggen.0.update slow
emc2$ bin/halcmd addf freqgen.update_freq slow
emc2$ bin/halcmd addf freqgen.make_pulses fast
emc2$
```

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

```
emc2$ bin/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
emc2$
```

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

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:

```
emc2$ bin/halcmd setp freqgen.0.velocity-scale 10000
emc2$ bin/halcmd setp freqgen.1.velocity-scale 10000
emc2$
```

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.

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:

```
emc2$ bin/halcmd start
emc2$
```

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.

2.5 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.

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. Before starting the GUI you must load the realtime part:

```
emc2$ bin/halcmd loadrt scope_rt
emc2$
```

Once the realtime part is loaded, we can start the GUI. Like halmeter, you can follow it with & so it runs in the background and you get your shell prompt back immediately:

```
emc2$ bin/halscope &
[2] 3678
emc2$
```

The scope GUI window will open, immediately followed by a "Realtime function not linked" dialog that looks like figure 2.4[4].
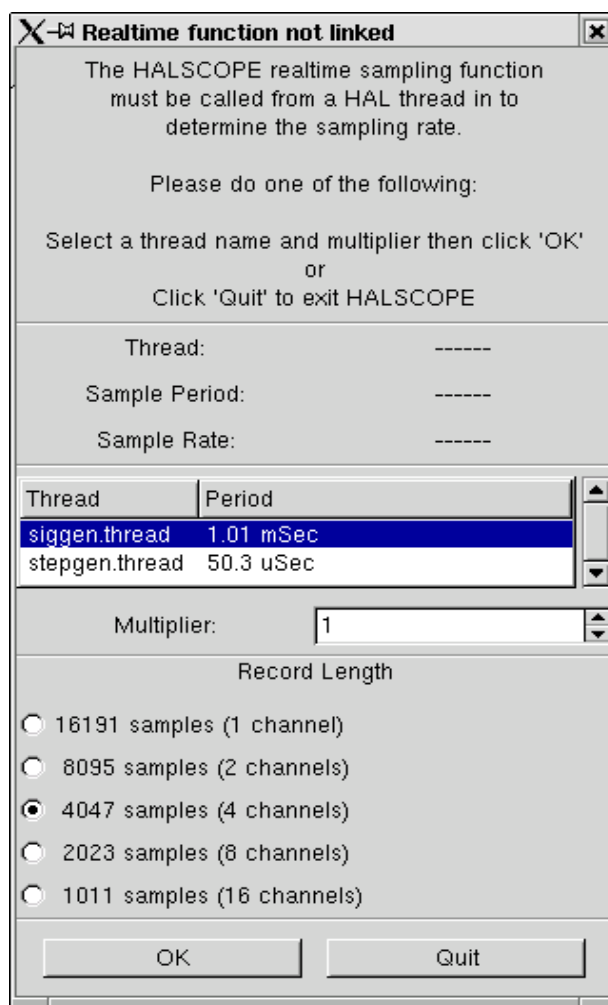
Figure 2.4: "Realtime function not linked" dialog

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.

---

[4]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 2.5: Initial scope window

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 "vi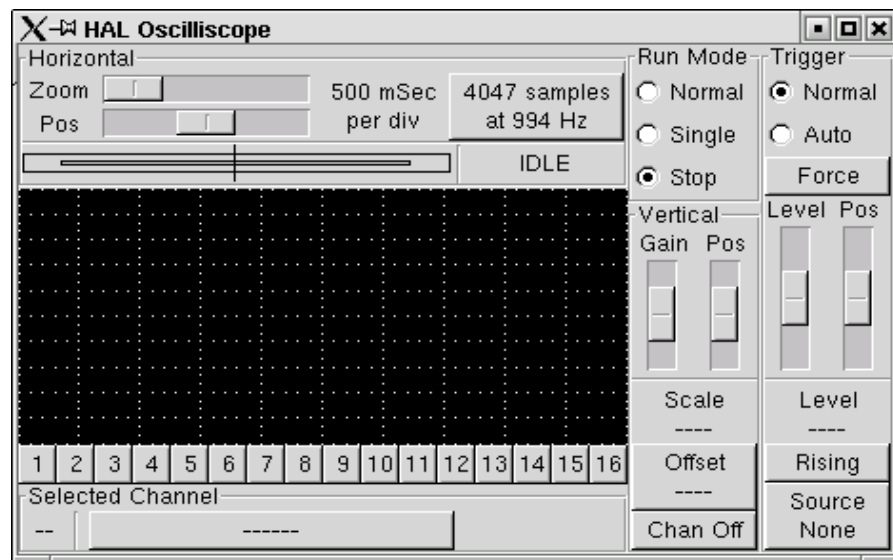rtual scope probes" to the HAL. Halscope has 16 channels, but the number you can use at any one time depends on the record length - more channels means shorter records, since the memory available for the record is fixed at approximately 16,000 samples.

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).
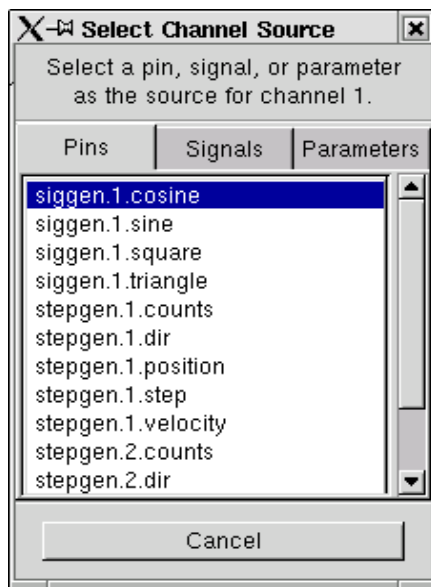
Figure 2.6: Select Channel Source dialog

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".

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.
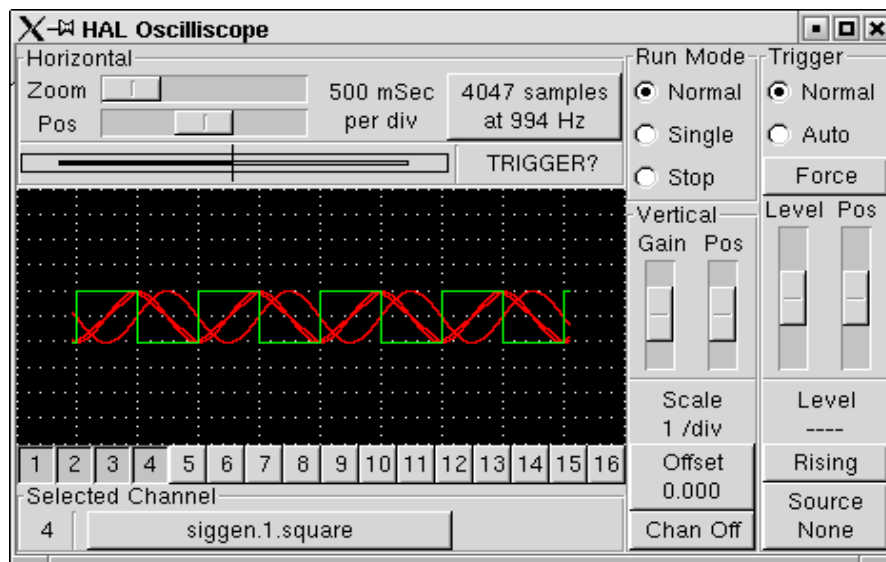

Figure 2.7: Captured Waveforms

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.

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 7.4 for details).

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.
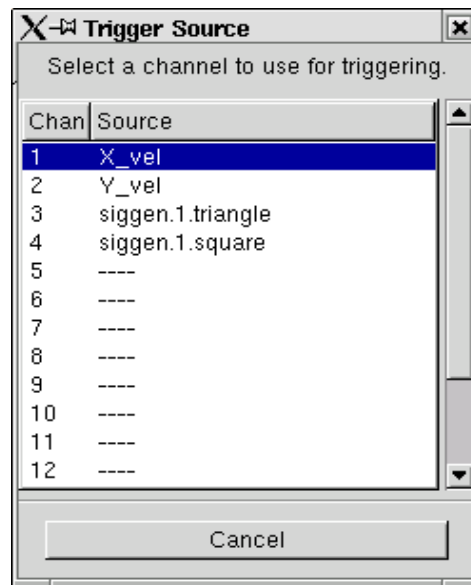
Figure 2.8: Trigger Source Dialog

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.
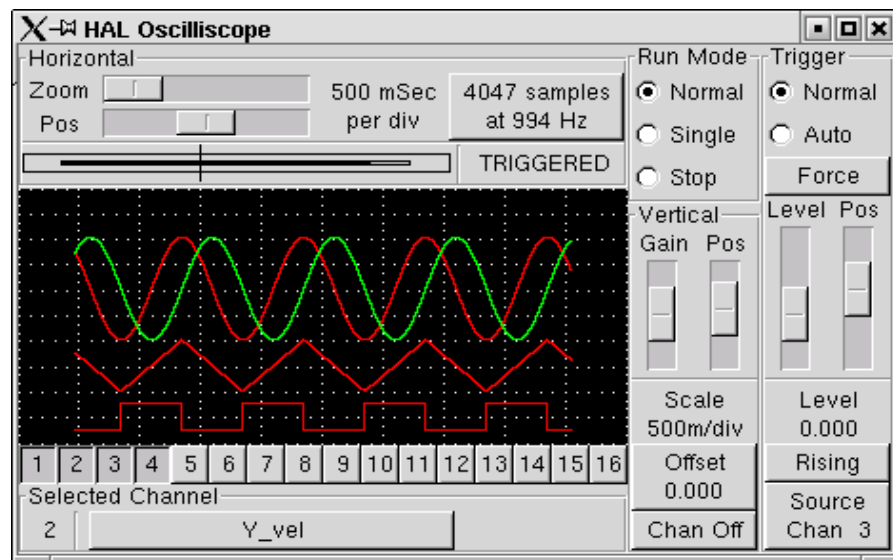
Figure 2.9: Waveforms with Triggering

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.
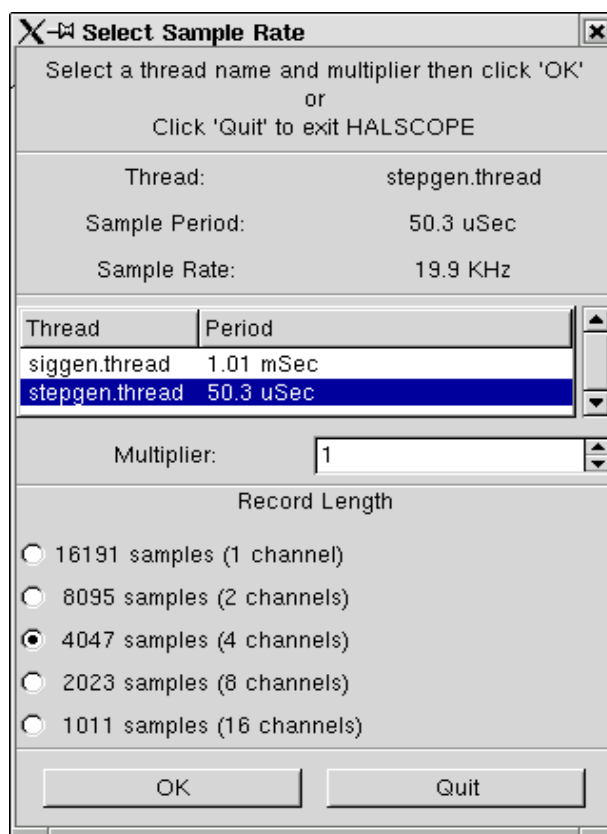
Figure 2.10: Sample Rate Dialog

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.
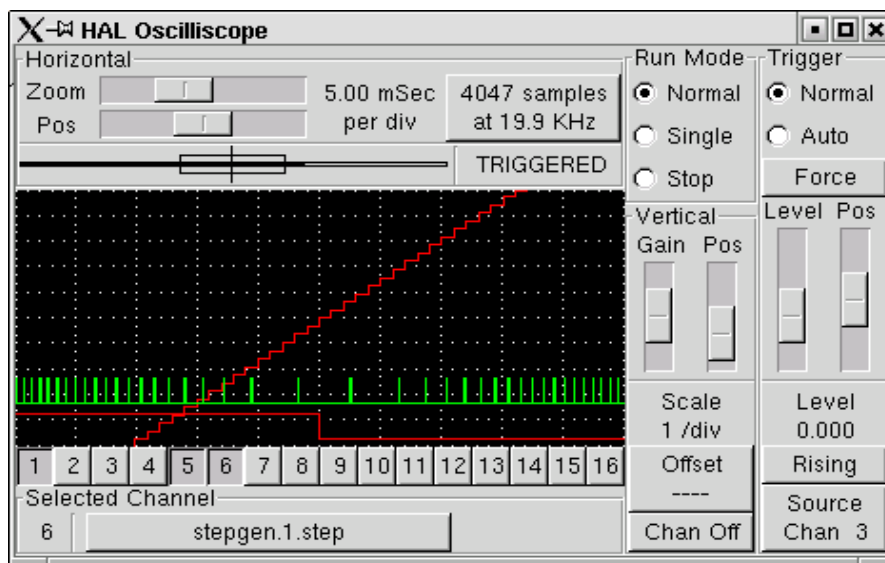
Figure 2.11: Looking at Step Pulses

Part II

HAL Reference

Chapter 3

General Reference Information

3.1 Notation

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.

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 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.[1]

- Hypens ("-") separate words or fields in the same level of the heirarchy.

- HAL modules should not use underscores or "MixedCase". [2]

- Use only lowercase letters and numbers in names.

3.3 Hardware Driver Naming Conventions[3]

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.[4]

`<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.

---

[1]You might ask " why not use a slash?" That is a very good question, and it is being considered for the next revision of HAL. Comments and opinions from users are wanted.

[2]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".

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

[4]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. I consider this a bug.

<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.[5] 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, 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.

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.

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.[6]

<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.

---

[5]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.

[6]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.

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

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.

Pins

- (BIT) **in** – State of the hardware input.

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

Parameters

- None

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.

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.

Parameters

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

Functions

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

4.3 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.)

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.

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.

Functions

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

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

Chapter 5

Hardware Drivers

5.1 Parport

Parport is a driver for the traditional PC parallel port. Each 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 output pins, and the status group is 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. No other combinations are supported, and a port cannot be changed from input to output once the driver is installed. Figure 5.1 shows two block diagrams, one showing the driver when the data group is configured for output, and one showing it configured for input.

There are actually two versions of the parport driver. One is a kernel module, and provides realtime control of the parallel port. The other is a user space process, and is not realtime. The non-realtime version is intended mainly for testing, and is not recommended for most applications. Using both the realtime and non-realtime versions at the same time is a bad idea.

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

Installing

Realtime version:

```
emc2$ bin/halcmd loadrt hal_parport cfg="<config-string>"
```

Non-realtime version:

```
emc2# bin/hal_parport <config-string> &
```

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

```
emc2# bin/hal_parport 278 378 in 20A0 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. However PCI ports may at nearly any address, and finding the address can be tricky[1]. There is no default address - if `<config-string>` does not contain at least one address, it is an error.

Removing

Realtime version:

>   emc2$ **bin/halcmd unloadrt hal_parport**

Non-realtime version:

>   Remove the non-realtime version by sending SIGINT or SIGTERM.

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.

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.

---

[1]Perhaps a future version of this driver will attempt to auto-identify PCI port addresses - however, it is very important that the user (or system integrator) makes sure the ports are configured correctly. Sending step and direction pulses to a LaserJet by accident simply wastes paper, but spooling a print job to stepper or servo motors could cause unexpected machine movement and possibly serious or fatal injuries.
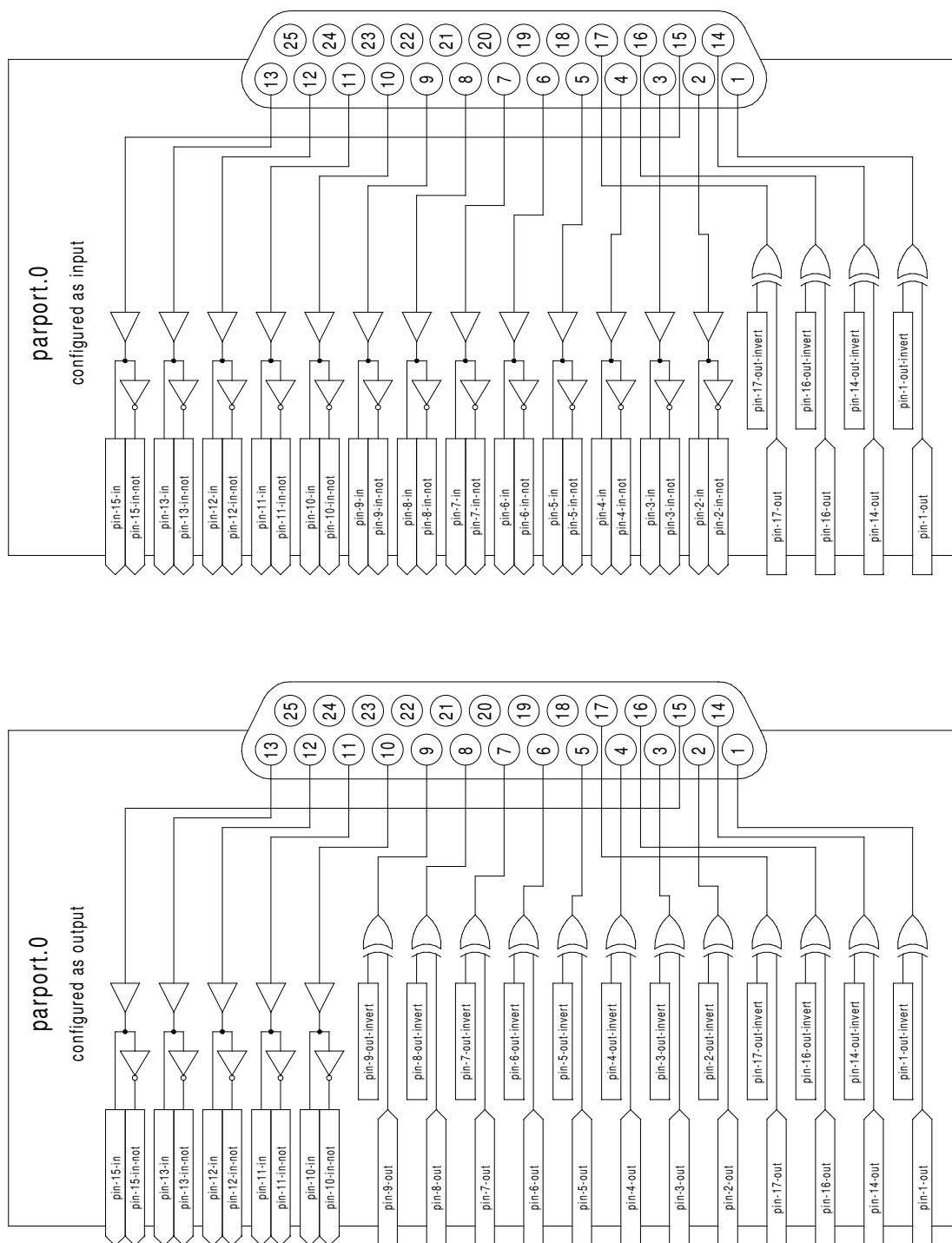
Figure 5.1: Parport Block Diagram

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.

The user space version of the driver cannot export functions, instead it exports parameters with the same names. Then the driver sits in a loop checking the parameters. If they are zero, it does nothing. If any parameter is greater than zero, the corresponding function runs once, then the parameter is reset to zero. If any parameter is less than zero, the corresponding function runs on every pass through the loop. The driver will loop forever, until it receives either SIGINT (ctrl-C) or SIGTERM, at which point it cleans up and exits.

5.2 Servo-To-Go

The Servo-To-Go is one of the first PC motion control cards[2] 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.

Installing:

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

The base address field is optional, in case 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[3]. For example:

```
emc2$ bin/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

---

[2]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.)

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

```
emc2$ bin/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.

Removing

```
emc2$ bin/halcmd unloadrt hal_stg
```

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.

- (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[4].

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.

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.

---

[4]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)

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.

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.

Chapter 6

Internal Components

6.1 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 6.1 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).

Installing

        emc2$ **bin/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# **/sbin/insmod rtlib/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.

Removing

        emc2$ **bin/halcmd unloadrt stepgen**

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).
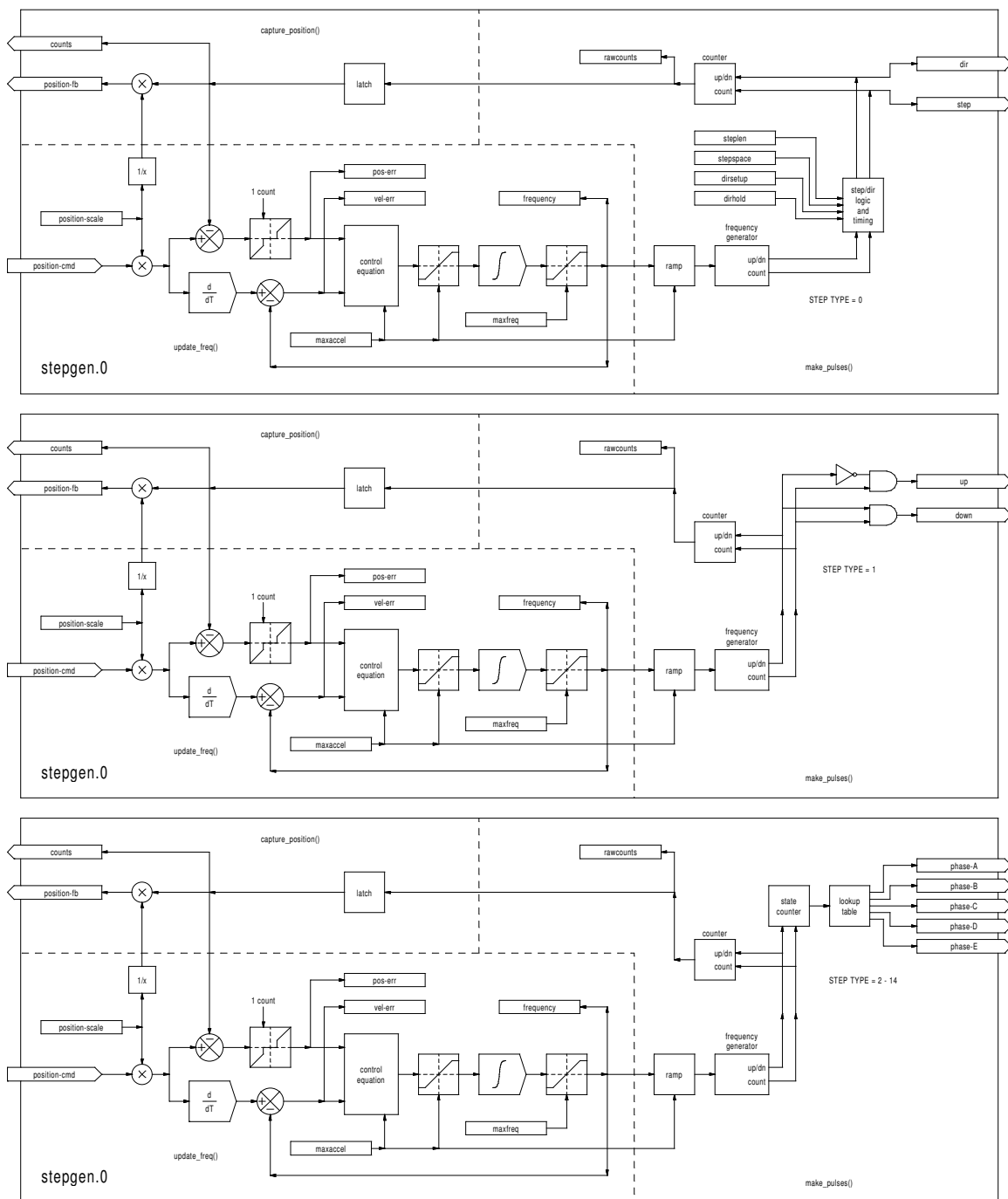
Figure 6.1: Step Pulse Generator Block Diagram

- (S32) `stepgen.<chan>.count` – Feedback position in counts, updated by `capture_position()`.

- (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).

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 value, 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.

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 6.2 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.
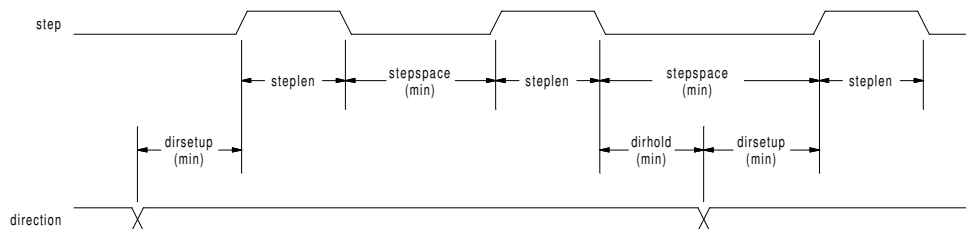
Figure 6.2: Step and Direction Timing

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 6.3, 6.4, and 6.5 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.

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.
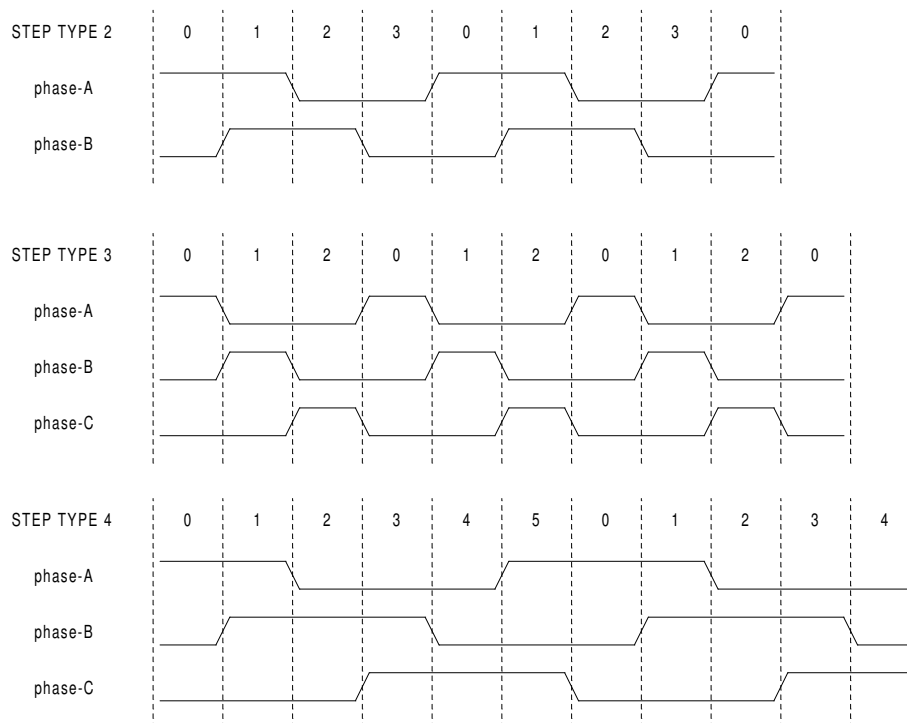
Figure 6.3: Quadrature and Three Phase Step Types

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

- (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 6.4: Four-Phase Step Types

Figure 6.5: Five-Phase Step Types

6.2 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 6.1 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 loo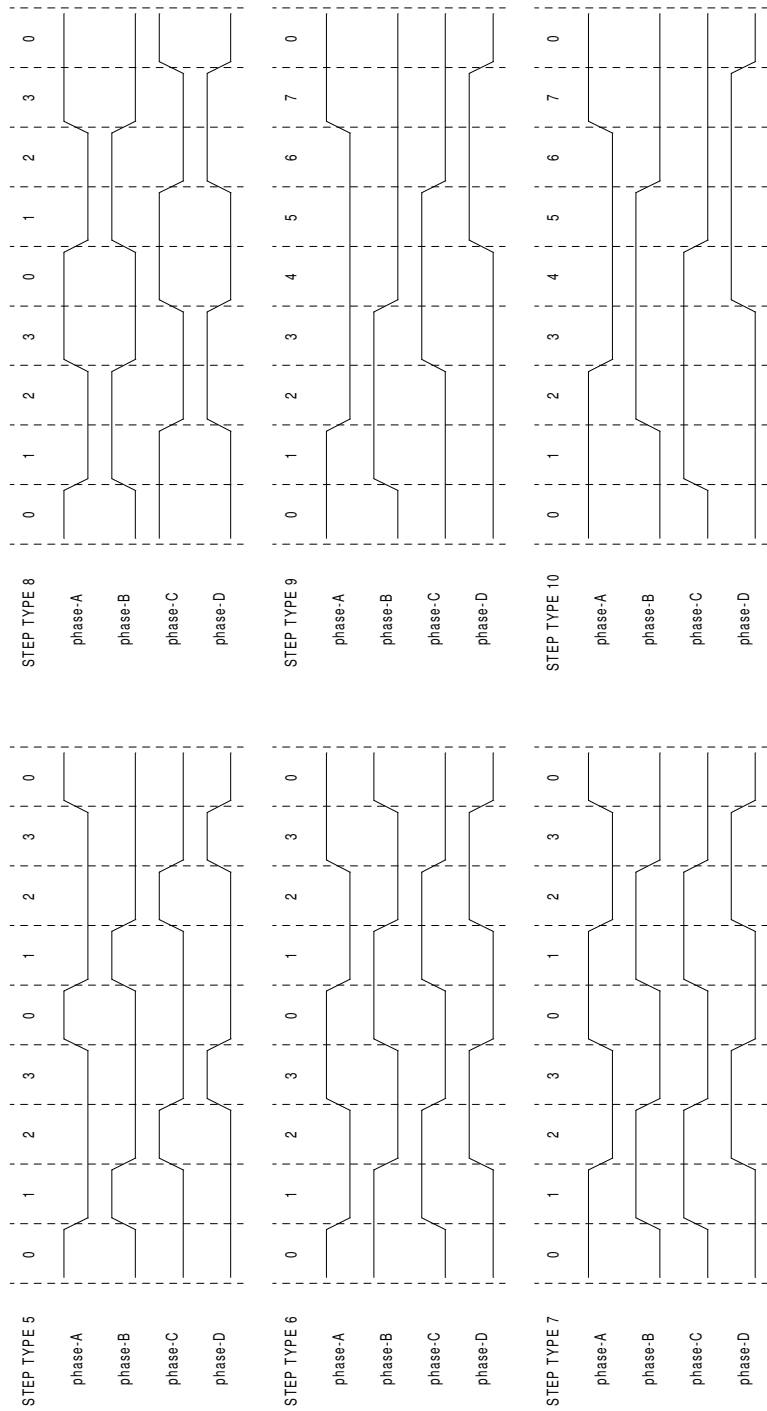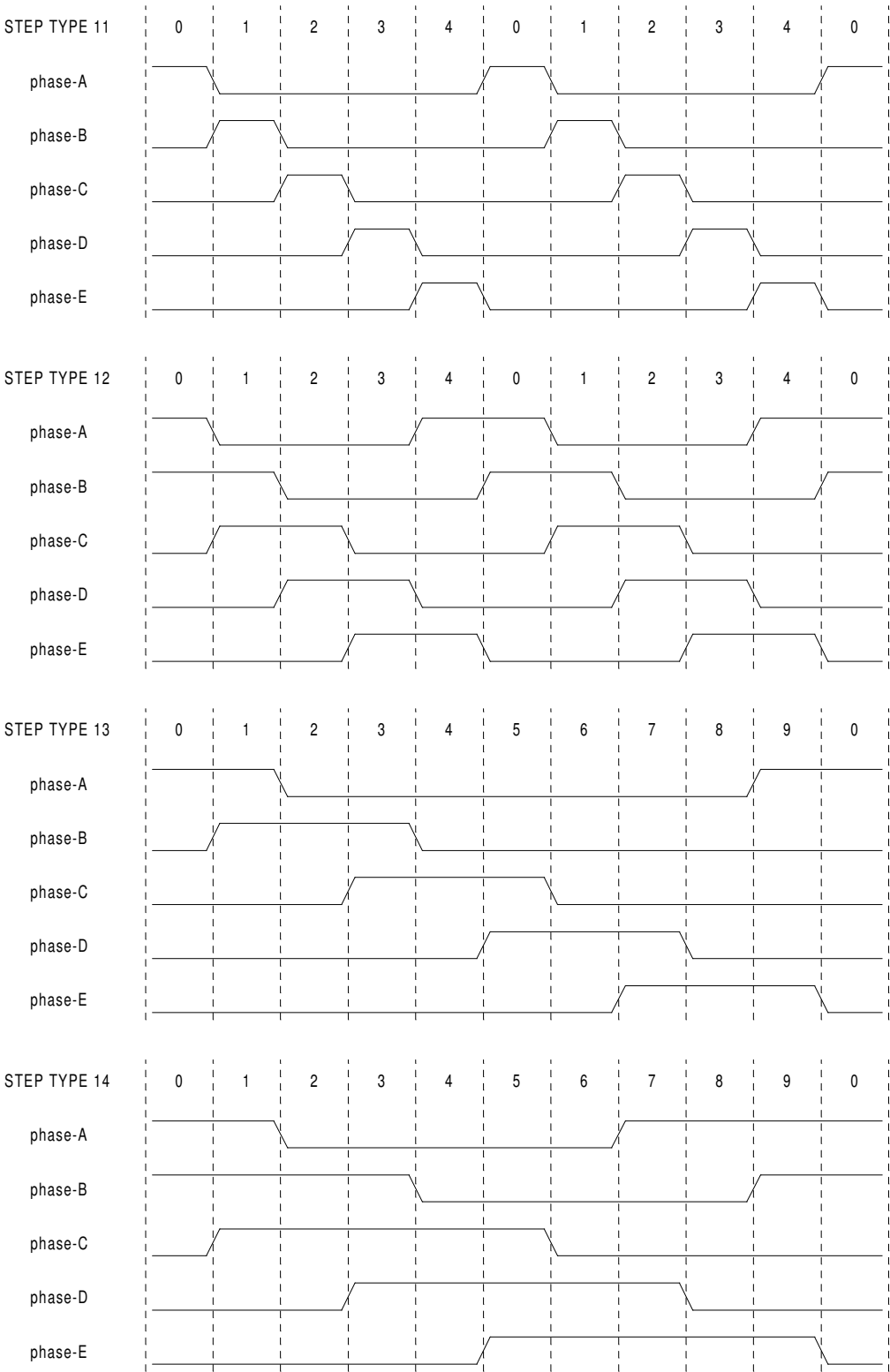p. the software encoder counter module, and a simple H-bridge, this has been used to run small DC servomotors. Freqgen uses the same pulse generator core as stepgen, however it has no position loop. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum step rates of 10kHz to perhaps 50kHz. Figure 6.6 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).

Installing

    emc2$ **bin/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$ **bin/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.

Removing

    emc2$ **bin/halcmd unloadrt freqgen**

Pins

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

- (BIT) 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).
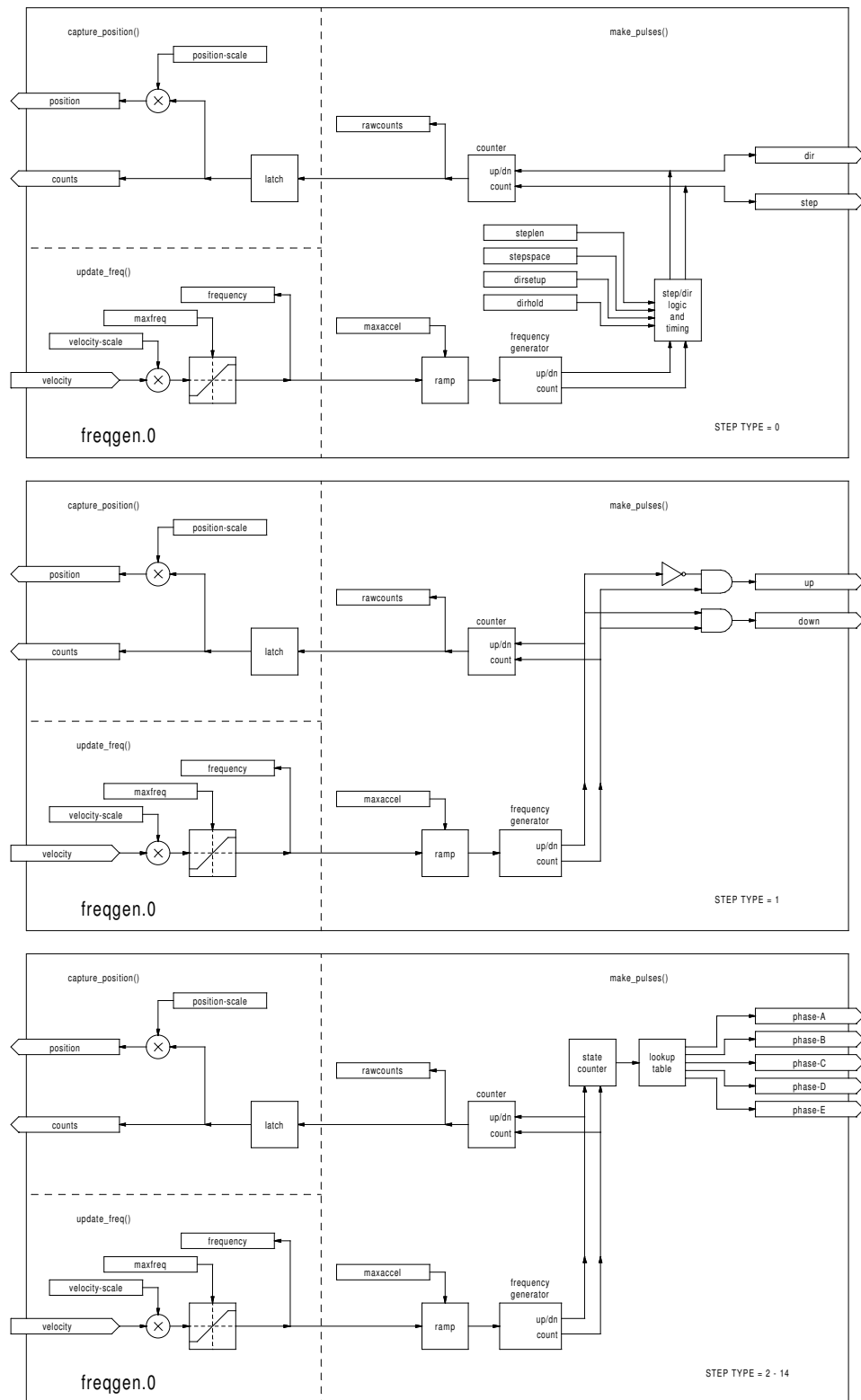- (BIT) freqgen.<chan>.phase-D – Phase D output (step types 5-14 only).

Figure 6.6: 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()`.

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 value, updated by `make_pulses()`.

- (`FLOAT`) `freqgen.<chan>.position-scale` – The scale factor used to convert from feedback counts to position units.

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 (section 6.1). Refer to that section for more information.

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. If you need a distinct pulse for each step, the frequency needs to be limited 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.

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.

6.3 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 6.7 is a block diagram of one channel of encoder counter.
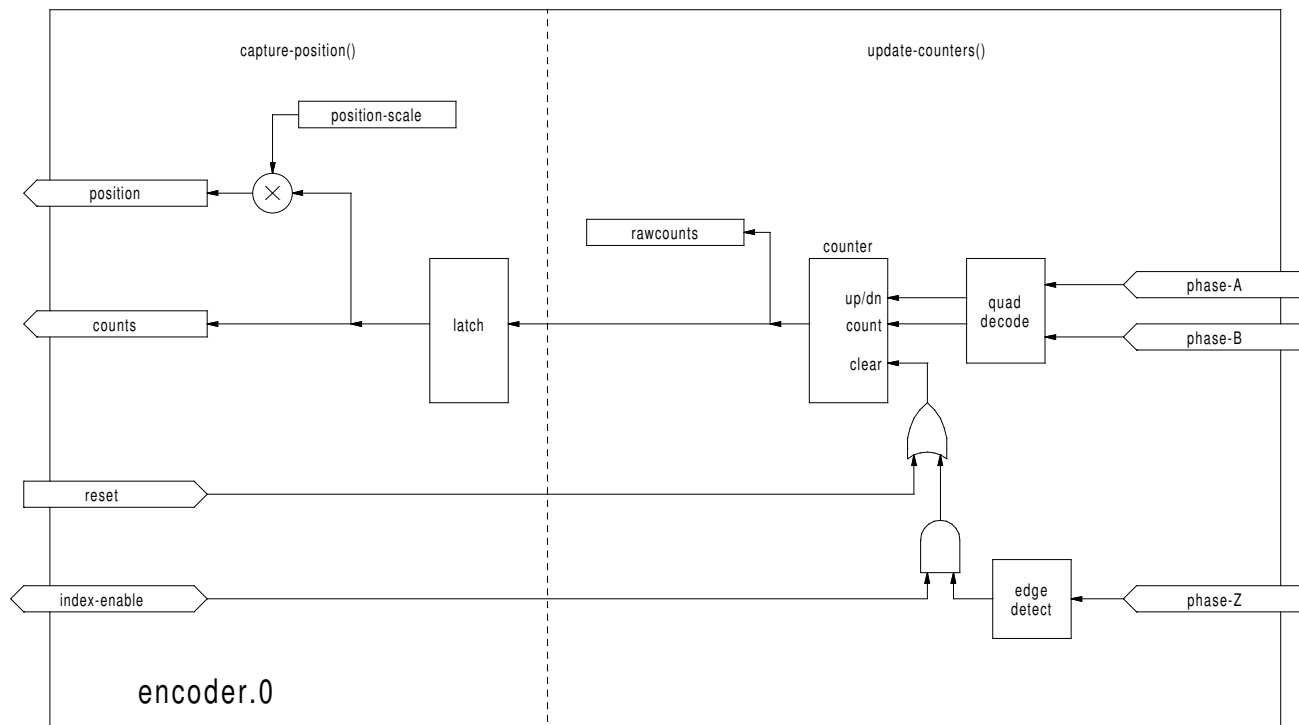
Figure 6.7: Encoder Counter Block Diagram

Installing

```
emc2$ bin/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.

Removing

```
emc2$ bin/halcmd unloadrt encoder
```

Pins

- (BIT) encoder.<chan>.phase-A – Phase A of the quadrature encoder signal, read by update-counters()
- (BIT) encoder.<chan>.phase-B – Phase B of the quadrature encoder signal, read by update-counters()
- (BIT) encoder.<chan>.phase-Z – Phase Z (index pulse) of the quadrature encoder signal, read by update-counters().
- (BIT) encoder.<chan>.reset – See canonical encoder interface, section 4.3.[1]
- (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.

Parameters

- (s32) encoder.<chan>.raw-count – The raw count value, updated by update-counters().
- (FLOAT) encoder.<chan>.position-scale – See canonical encoder interface, section 4.3.

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.

---

[1]The component doesn't currently implement the velocity pin, so it is not a complete implementation of the canonical interface.

6.4   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 6.8 is a block diagram of a single PID loop.

Installing

    emc2$ **bin/halcmd loadrt pid [num_chan=<loops>] [debug=1] [fp_period=<nsec>]**

`<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.

If `fp_period` is specified, the component will create a realtime thread, capable of running floating point functions. The period of the thread will be `<nsec>` nano-seconds. By default, no thread is created.

Removing

    emc2$ **bin/halcmd unloadrt pid**

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.
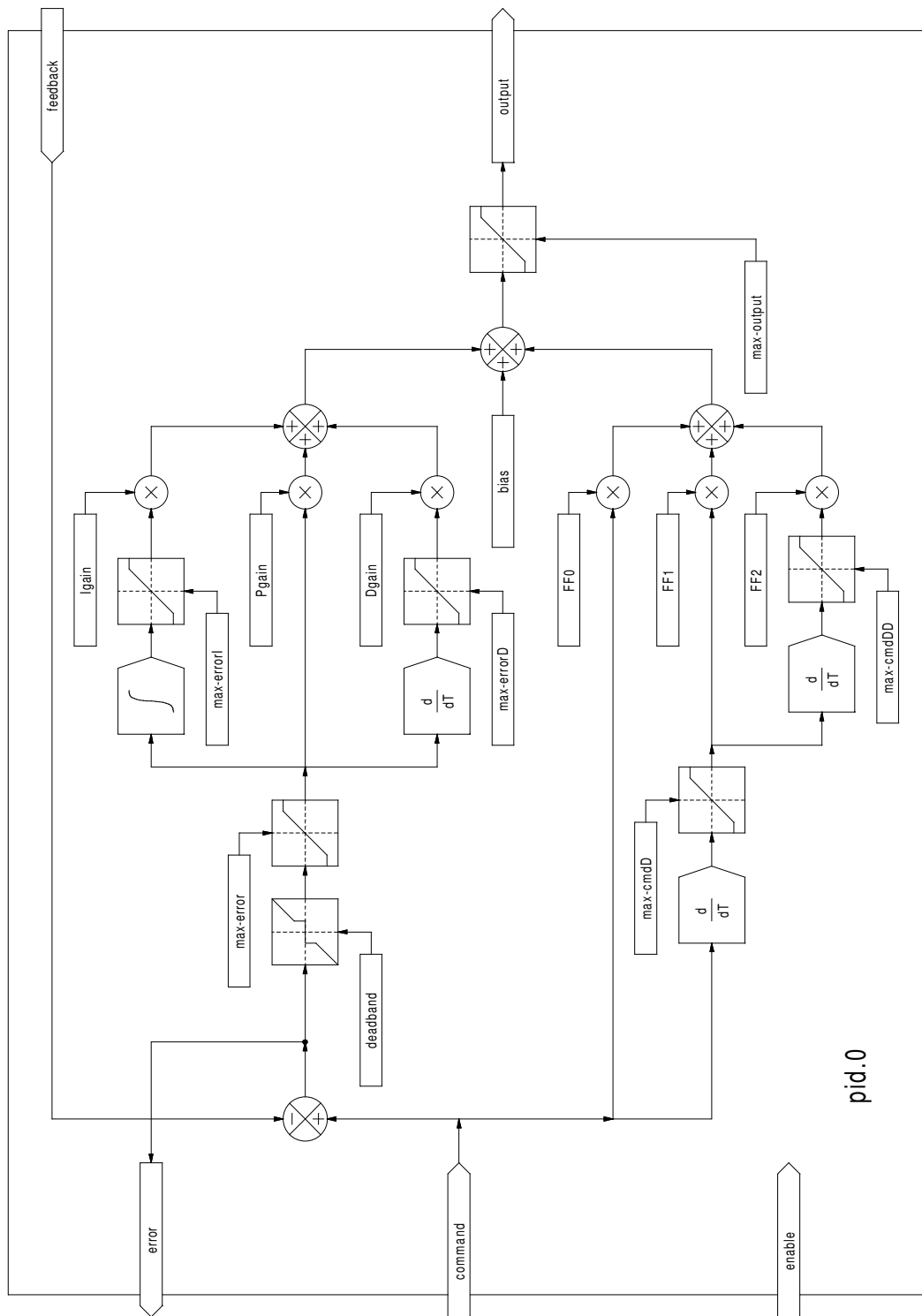
Figure 6.8: PID Loop Block Diagram

Parameters

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

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

- (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)[2].

- (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.

Functions

The component exports one function for each PID loop. This function performs all the calculations needed for the loop. Since each loop has it's 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 6.8, 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().

---

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

6.5   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.

Installing

    emc2$ **bin/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).

Removing

    emc2$ **bin/halcmd unloadrt sim-encoder**

Pins

- (FLOAT) sim-encoder.<chan-num>.speed – The speed command for the simulated shaft, in revs per second.

- (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.

Parameters

- (U32) sim-encoder.<chan-num>.ppr – Pulses Per Revolution.

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.

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**.

6.6   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.

Installing

    emc2$ **bin/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$ **bin/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.

Removing

    emc2$ **bin/halcmd unloadrt debounce**

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>`.

Parameters

Each group of filters has one parameter[3].

- (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 it's 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.

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>`.

6.7 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.

---

[3]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.

Installing

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

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

Removing

```
emc2$ bin/halcmd unloadrt blocks
```

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)

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.

Functions

Each individual block has it's 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`.

6.8 Siggen

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

Installing

```
emc2$ bin/halcmd loadrt siggen [num_chan=<chans>] [fp_period=<nsec>]
```

`<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).

If `fp_period` is specified, the component will create a realtime thread, capable of running floating point functions. The period of the thread will be `<nsec>` nano-seconds. By default, no thread is created.

Removing

```
emc2$ bin/halcmd unloadrt siggen
```

Pins

Each generator has four 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.

Parameters

Each generator is controlled by three parameters.

- (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.

Functions

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

6.9 Supply

Chapter 7

Utility Components

7.1 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, while this section is a detailed reference listing all of the available commands.

Usage

```
halcmd [<options>] [<command>[<args>]]
```

Options

```
-f [<filename>]
```

File mode: Ignores commands on command line, takes input from `<filename>` instead. If `<filename>` is not specified, takes input from `stdin`.

```
-k
```

Keep going after failed command(s). The default is to stop and return failure if any command fails.

```
-q
```

Quiet: Prints messages only when errors occur. This is the default.

```
-Q
```

Extra quiet: Prints nothing, executes commands silently. Note that some commands like `show` and `save` do nothing except print some information. Giving the `-Q` option to one of those commands prevents the information from being printed, making the command pointless.

```
-s
```

Script mode: Modified how `show` outputs information. Supresses headers, and prints most lists (e.g. thread functions, signal pins) on a single line. Suppresses the `halcmd:` prompt if `-f` is also specified (for use by other programs that issue commands via a pipe).

    -v

Verbose: Prints messages showing the results of each command.

    -V

Extra Verbose: Prints lots of debugging messages. Very messy, not normally used.

    -h

Help: Prints a help screen and exits.

Commands

Commands tell **halcmd** what to do. If invoked without the -f option, **halcmd** reads the remainder of the command line and treats it as a single command. If invoked with the `-f` option, halcmd reads the specified file and treats each line as a command. The -f option allows you to very quickly execute a whole series of commands, and is the most common way to configure the HAL.

    #

Commands starting with '#' are considered to be comments, and are ignored by halcmd.

    loadrt <compname> [<arg1>] [<arg2>]

Load realtime component: Loads the realtime HAL component `<compname>`, passing it configuration arguments `<arg1>`, `<arg2>`, etc. Halcmd uses the external program emc_module_helper to load the module. If `<compname>` is not found in the directory that emc_module_helper expects to contain the components, the command will fail.

    unloadrt <compname>

Unload realtime component: Unloads the realtime HAL component `<compname>`. Halcmd uses the external program emc_module_helper to unload the module. If `<compname>` is not loaded, the command will fail.

    newsig <signame> <type>

New signal: Creates a new HAL signal called `<signame>` that may later be used to connect two or more HAL pins. `<type>` is the data type of the new signal, and must be one of `bit`, `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, **or** `float`. If a signal of the same name already exists, the command will fail.

    delsig <signame>

Delete signal: Deletes a HAL signal called `<signame>`. Any pins currently linked to the signal will be unlinked. If a signal called `<signame>` does not exist, the command will fail.

```
linkps <pinname> <signame>
linkps <pinname> => <signame>
linkps <pinname> <= <signame>
linkps <pinname> <=> <signame>
linksp <signame> <pinname>
linksp <signame> => <pinname>
linksp <signame> <= <pinname>
linksp <signame> <=> <pinname>
```

Link pin to signal; Link signal to pin: Establishes a link between a HAL component pin `<pinname>` and a HAL signal `<signame>`. Any previous link to `<pinname>` will be broken. The "arrows", =>, <=, or <=>, are optional, and are ignored by halcmd. They should not be used on the command line, as the shell is likely to misinterpret them. However, when writing commands in a file, the arrows can be used to indicate which direction data flows through the link. The future confusion they prevent may be your own! The two forms `linksp` and `linkps` are provided for the same reason. Using the appropriate one for the situation can make a file easier to understand, however they both have the exact same effect. If either `<pinname>` or `<signame>` does not exist, or if their types don't match, the command will fail.

```
linkpp <pinname1> <pinname2>
linkpp <pinname1> => <pinname2>
linkpp <pinname1> <= <pinname2>
linkpp <pinname1> <=> <pinname2>
```

Link pin to pin: Establishes a link between two HAL component pins `<pinname1>` and `<pinname2>`. A signal is created with the same name and type as `pinname1`, and both the named pins are connected to the newly created signal. This is the equivalent of issuing the following three commands:

```
newsig pinname1_type pinname1
linksp pinname1 pinname1
linksp pinname1 pinname2
```

The "arrows", =>, <=, or <=>, are optional, and are ignored by halcmd. If either `<pinname1>` or `<pinname2>` does not exist, or if their types don't match, the command will fail.

```
unlinkp <pinname>
```

Unlink pin: Breaks any previous link to `<pinname>`. If `<pinname>` does not exist, or does not have a signal linked to it, the command will fail.

```
setp <paramname> <value>
<paramname> = <value>
```

Set parameter: Sets the value of parameter `<paramname>` to `<value>`. The second form of the command has exactly the same effect, but is provided for use in files where it may make the file more readable. The second form is not recommended on the command line, since the shell may misinterpret the equal sign. If `<paramname>` does not exist, or if it is not writable, the command will fail. It will also fail if `<value>` is not a legal value for `<paramname>`'s data type. For example, `300` is not a legal `u8` value, and `TRUE` is legal only for type `bit`. (Type `bit` will accept `0` and `1` as well as `TRUE` and `FALSE`).

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

Add function: Adds function `<functname>` to realtime thread `<threadname>`. `<position>` determines where in the thread the function is added, and thus in what order the thread's functions

will execute. If `<position>` is positive, the function will be inserted in the corresponding location relative to the beginning of the thread. Thus if `<position>` is 1, the function will be inserted at the beginning of the thread, and if it is 3 it will be inserted as the third function in the thread. If `<position>` is negative, the function will be inserted relative to the end of the thread. So -1 means the last function, and -2 is next to last, etc. Zero is illegal. If no position is specified, -1 is assumed, and the function is inserted at the end of the thread. If either `<functname>` or `<threadname>` does not exist, the command will fail. It will also fail if the function requires floating point and the thread does not support it, or if the function is non-reentrant and is already in a thread. It also fails if `<position>` specifies something impossible, for example asking for position 3 when the thread only has one function in it.

```
delf <functname> <threadname>
```

Delete function: Removes function `<functname>` from realtime thread `<threadname>`. The command will fail if either `<functname>` or `<threadname>` does not exist, or if `<functname>` is not currently part of `<threadname>`. If the function appears in the thread more than once, only the first instance is deleted.

```
newthread <threadname> <period>
```

New thread: Creates a new realtime thread that can be used to execute HAL functions at specific intervals. The thread is called `<threadname>`, and it executes every `<period>` nano-seconds. If a thread of the same name already exists, the command will fail. [1]

```
delthread <threadname>
```

Delete thread: Deletes a realtime thread called `<threadname>`. Any functions currently linked to the thread will be unlinked. If a thread called `<threadname>` does not exist, the command will fail. [2]

```
start
```

Start: Begins execution of realtime threads. When started, each thread runs at it's specified period. Each time the thread runs, it calls all of the functions that were added to it with the `addf` command. The functions are called in the order that was specified by the `<position>` argument of the `addf` command.

```
stop
```

Stop: Ends execution of realtime threads. The threads will no longer call their functions.

```
show [<item>]
```

Show info: Prints information about HAL items to `stdout` in human readable format. `<item>` can be `comp` (components), `pin`, `sig` (signals), `param` (parameters), `funct` (functions), or `thread`, or omitted. If `<item>` is omitted, `show` will print everything.

```
save [<item>]
```

---

[1]This command is NOT currently implemented. It requires some complex user/kernel calling procedures, and may not be working in time for the NAMES show and demonstration. At the present time, threads can only be created by kernel modules, not by a user space process like halcmd. A number of the existing kernel module HAL components have provisions for creating threads when they are insmod'ed.

[2]This command is NOT currently implemented. See note above.

Save info: Prints HAL items to `stdout` in the form of HAL commands. These commands can be redirected to a file and later executed using `halcmd -f` to restore the saved configuration. `<item>` can be one of the following: `sig` generates a `newsig` command for each signal, `link` and `linka` both generate `linkps` commands for each link. (`linka` includes arrows, while `link` does not.) `net` and `neta` both generate one `newsig` command for each signal, followed by `linksp` commands for each pin linked to that signal. (`neta` includes arrows.) `param` generates one `setp` command for each parameter. `thread` generates one `addf` command for each function in each realtime thread.[3] If `<item>` is omitted, **save** does the equivalent of `sig`, `link`, `param`, and `thread`. [4][5]

7.2 Halgui

Halgui is a program that doesn't exist yet. It will be a GUI version of halcmd. Halcmd is a rather thin wrapper over the core HAL api as described in `emc2/src/hal/hal.h`. If a GUI expert would like to work on a more sophisticated wrapper, please contact me (jmkasunich at att dot net). I would be happy to work with you on it. If nobody volunteers, I will do it eventually. However, GUI and user interface coding are not areas in which I am talented, so it will take a while.

7.3 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.)

Halmeter is due for a rewrite - the new version will have a nicer display, with autoranging, range hold, and an analog bar graph to supplement the digital display. However it's purpose will remain the same - a handy software equivalent to DMM for basic testing and troubleshooting.

---

[3]Once `newthread` is working, `thread` will be modified to generate a `newthread` command for each thread before generating the `addf` commands.

[4]Halcmd cannot load components. As a result, the save command does not generate commands that can reload the current set of components. It only generates commands to interconnect them once they are loaded. Eventually I would like to extend halcmd so that it CAN load components, and save the list of loaded components in the form of commands that can reload them later. Once challenge will be saving config info that was given to the components when they were insmod'ed. That info is not visible to the HAL api or to the save command. Instead it remains inside the individual components. Technically that is where it should be - since the config requirements of individual components can vary widely.

[5]Update Oct. 5, 2004: The loadrt and unloadrt commands now allow halcmd to load and unload realtime components. However it is still impossible for halcmd to know about any arguements that were passed to the components when they were loaded. Therefore it is still impossible to save the currently loaded components in a way that will allow them to be reloaded correctly. In the long run, I intend to revise the way components are configured, such that they require fewer (ideally no) configuration arguments at load time. At that time, it will probably be practical to save the entire state of the HAL, including the list of loaded components.

7.4   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 2.5 in the tutorial chapter, which explains the basics.