HAL for Integrators

19th June 2004

Contents

1 Introduction **5**

1.1 What is HAL? 5

HAL is based on traditional system design techniques 5

Part Selection 5

Interconnection Design 6

Implementation 6

Testing 6

Summary 6

1.2 HAL Concepts 7

1.3 HAL components 8

External programs with HAL hooks 8

Internal Components 9

Hardware drivers 9

Utilities 9

1.4 Tinkertoys, Erector Sets, Legos and the HAL 9

Tower 9

Erector Sets 10

Tinkertoys 10

A Lego Example 11

1.5 Timing Issues In HAL 12

1.6 Dynamic Linking and Configuration 13

2 HAL Configuration **14**

2.1 Before we start 14

Notation 14

Root Privilges 14

The RTAPI environment 15

2.2 A Simple Example 15

Loading a realtime component 15

Examining the HAL 16

Making realtime code run 17

Changing parameters . . . 18
Saving the HAL configuration . . . 19
Restoring the HAL configuration . . . 19
2.3 Looking at the HAL with halmeter . . . 20
Starting halmeter . . . 20
Using halmeter . . . 20
2.4 A slightly more complex example. . . . 23
Installing the components . . . 23
Connecting pins with signals . . . 24
Setting up realtime execution - threads and functions . . . 26
Setting parameters . . . 27
Run it! . . . 27
2.5 Taking a closer look with halscope. . . . 28
Starting Halscope . . . 28
Hooking up the “scope probes” . . . 31
Capturing our first waveforms . . . 32
Vertical Adjustments . . . 33
Triggering . . . 33
Horizontal Adjustments . . . 35
More Channels . . . 36

3 Detailed Description of Internal Components 37
3.1 General Information . . . 37
Notation . . . 37
Names . . . 37
3.2 Stepgen . . . 38
Installing . . . 38
Removing . . . 38
Pins . . . 38
Parameters . . . 40
Step Types . . . 41
Functions . . . 41
3.3 Freqgen . . . 46
Installing . . . 46
Removing . . . 46
Pins . . . 46
Parameters . . . 48
Step Types . . . 48
Functions . . . 49
3.4 Encoder . . . 50
Installing . . . 50

Removing 51
Pins 51
Parameters 51
Functions 51
3.5 PID 52
Installing 52
Removing 52
Pins 52
Parameters 54
Functions 55
3.6 Debounce 56
Installing 56
Removing 56
Pins 56
Parameters 56
Functions 57
3.7 Blocks 57
Installing 57
Removing 57
Pins 58
Parameters 58
Functions 59
3.8 Siggen 59
Installing 59
Removing 59
Pins 59
Parameters 60
Functions 60
3.9 Supply 60

4 Detailed Description of Hardware Drivers 61
4.1 Parport 61
Installing 61
Removing 62
Pins 62
Parameters 62
Functions 64

5 Detailed Description of Utility Components 65
5.1 Halcmd 65
Usage 65
Options 65

5.2 Halgui 68
5.3 Halmeter 69
5.4 Halscope 69

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

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 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 (section 1.2). 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 an 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 oscilliscope 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, oscilliscope, 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 is 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

---

[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.

- 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

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 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 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 has at least two functions. One reads physical input pins and updates the HAL pins, the other takes data from the HAL pins and writes it to the physical output pins. But 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 component 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

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

classicladder A PLC using HAL for all I/O

Internal Components

stepgen Software step pulse generator. See section 3.2

encoder Software based encoder counter. See section 3.4

pid Proportional/Integral/Derivative control loops. See section 3.5

siggen A sine/cosine/triangle/square wave generator for testing. See section 3.8

supply a simple source for testing

Hardware drivers

hal_parport PC parallel port. See section 4.1

hal_stg Servo To Go card (not implemented yet)

hal_usc Pico Systems Universal Stepper Controller (not implemented yet)

Utilities

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

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

halmeter A handy multimeter for HAL signals. See section 5.3

halscope A full featured digital storage oscilliscope for HAL signals. See section 5.4

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.

> 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 humaine 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, catepillar, or earwig. A backbone of hubs, HAL components, strung together with rods, HAL signals. Each component takes in it own parameters and input

---

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

[3] Tinkertoyő is a registered trademark of the Playscool company.

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.

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 to 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.

---

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

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

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

1.5 Timing Issues In HAL

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

True relay logic consists of relays connected together, and when a contact opens or closes, current flows (or stops) immediately. Other coils may change state, etc, and it all just "happens". But in PLC style ladder logic, it doesn't work that way. Usually in a single pass thru 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 was 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 Configuration

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

Because HAL uses kernel modules to do much of it’s work, and because it also can access hardware directly, many commands will require root privileges. Note that it is usually safer to avoid doing day-to-day work as root. 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 3.8 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 `insmod` command:

```
emc2# /sbin/insmod rtlib/siggen.o fp_period=1000000
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 5.1 of this document . The first halcmd feature is the show command. This command displays information about the current state of the HAL. To show all installed components:

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

Since `halcmd` itself is a HAL component, it will always show up on the list. The list also shows the `siggen` component that we installed in the previous step. The “RT” under 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
```

The show param command shows all the parameters in the HAL. Right now each parameter has the default value it was given when the component was loaded. Note the column labeled `Dir`. The parameters labeled `-W` are writeable ones that are never changed by the component itself, instead they are meant to be changed by the user to control the component. We will see how to do this later. Parameters labeled `R-` are read only parameters. They can be changed only by the component. Finally, parameter labeled `RW` are read-write parameters. That means that thay are changed by the component, but can also be changed by the user.

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[1].

Making realtime code run

To actually run the code contained in the function `siggen.0.update`, we need a realtime thread. When we loaded the component we used the `fp_period` parameter to request a thread with a period of 1mS (1000000nS). Let's see if that worked:

```
emc2$ bin/halcmd show thread
Realtime Threads:
   Period   FP   Name
     999849 YES  siggen.thread
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 siggen.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.1.update` to the thread `siggen.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
     999849 YES  siggen.thread
                  1 siggen.0.update
emc2$
```

---

[1] 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  -8.21251e-01  siggen.0.cosine
 02    float -W   5.65397e-01  siggen.0.sine
 02    float -W  -1.00000e+00  siggen.0.square
 02    float -W  -6.17446e-01  siggen.0.triangle
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir    Value      Name
 02    float -W  -6.94009e-01  siggen.0.cosine
 02    float -W   7.19966e-01  siggen.0.sine
 02    float -W  -1.00000e+00  siggen.0.square
 02    float -W  -4.88315e-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, 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  -2.52580e+00  siggen.0.cosine
 02    float -W   4.31513e+00  siggen.0.sine
 02    float -W  -5.00000e+00  siggen.0.square
 02    float -W  -1.68567e+00  siggen.0.triangle
emc2$ bin/halcmd show pin
Component Pins:
Owner  Type  Dir    Value      Name
 02    float -W  -4.94738e+00  siggen.0.cosine
 02    float -W  -7.54523e-01  siggen.0.sine
 02    float -W   5.00000e+00  siggen.0.square
 02    float -W  -4.45782e+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.

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
# 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 siggen.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

Tobin/halcmd save >saved.hal

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# /sbin/insmod rtlib/siggen.o fp_period=1000000
emc2$ bin/halcmd addf siggen.0.update siggen.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, we can start it 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.

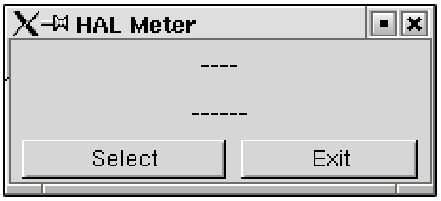
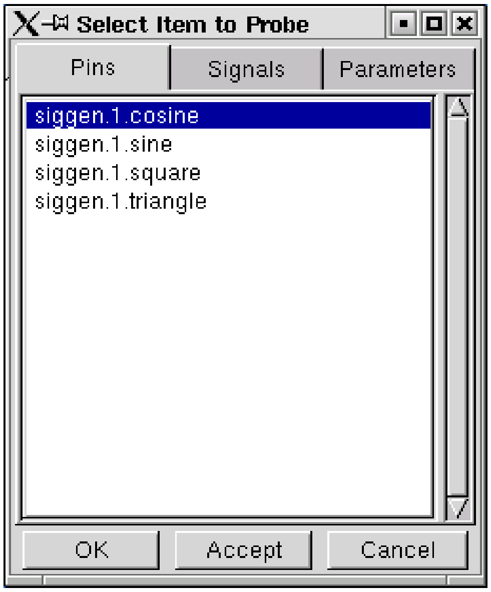


Figure 2.1: Halmeter at startup, nothing selected

Using halmeter

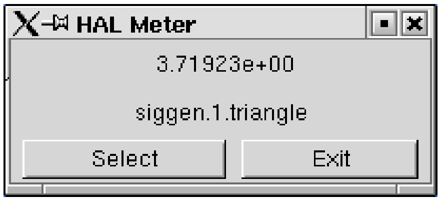
The meter in figure 2.1 isn't very useful, because it isn't displaying anything. To change that, click on the 'Select' button, which will open the probe selection dialog (figure 2.2).

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.



siggen.1.cosine
siggen.1.sine
siggen.1.square
siggen.1.triangle

Figure 2.2: Halmeter source selection dialog



3.71923e+00
siggen.1.triangle

Figure 2.3: Halmeter displaying the value of a pin

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

If you want to quickly look at a number of pins, you can use the 'Accept' button in the source selection dialog. Click on 'Select' to open the dialog again. This time, click on another pin, like siggen.1.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. [2]

---

[2] Halmeter is due for a rewrite, probably right after NAMES is over. 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 siggen component and reload the RTAPI and HAL libraries:

```
emc2# /sbin/rmmod siggen
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 3.3. For now, we can skip the details, and just run the following command:

```
emc2# /sbin/insmod rtlib/freqgen.o cfg="0 0" period=50000
emc2$
```

This command loads two step generators, both configured to generate stepping type 0. It also creates a 50 micro-second thread. Next we load our old friend siggen, the signal generator component, and create a 1 millisecond thread:

```
emc2# /sbin/insmod rtlib/siggen.o fp_period=1000000
emc2$
```

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.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 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.

Tthe 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  siggen.thread
      50286 NO   freqgen.thread
emc2$
```

There are two threads (which were created when we `insmod`'ed the components). The first one, `siggen.thread`, 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 `freqgen.thread`, 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 siggen.thread
emc2$ bin/halcmd addf freqgen.update_freq siggen.thread
emc2$ bin/halcmd addf freqgen.make_pulses freqgen.thread
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
    1005720 YES  siggen.thread
                  1 siggen.0.update
                  2 freqgen.update_freq
      50286 NO   freqgen.thread
                  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 oscilliscope. 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 supplied the GUI and display. Before starting the GUI you must load the realtime part:

```
emc2# /sbin/insmod rtlib/scope_rt.o
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:

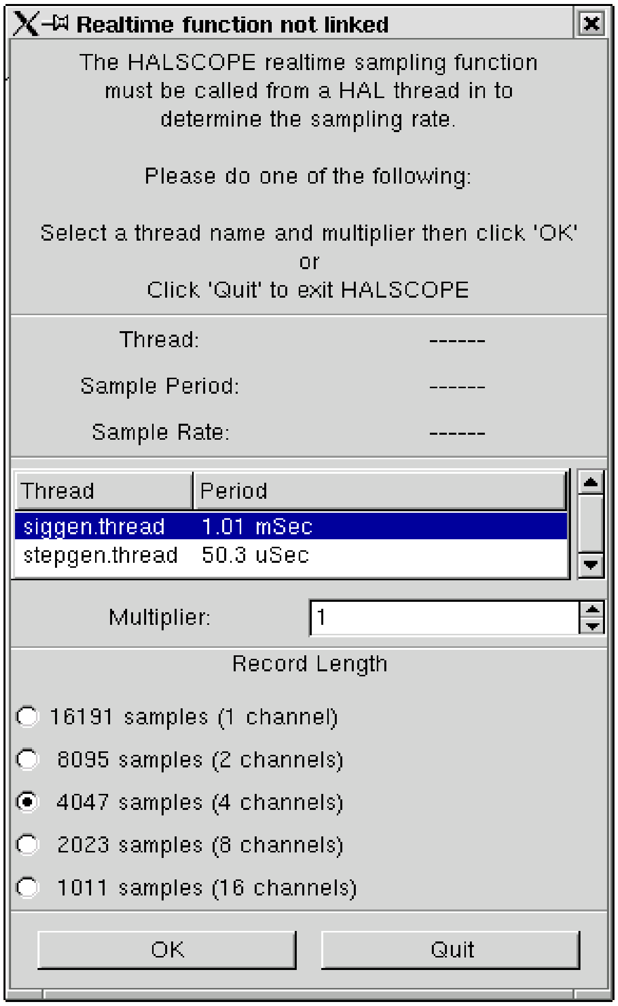
```
emc2$ bin/halscope &
[2] 3678
emc2$
```

The scope GUI window will open, immediately followed by a "Realtime function not linked" dialog that looks like figure 2.4[3].

This dialog is where you set the sampling rate for the oscilliscope. For now we want to once per millisecond, so click on the 1.03mS thread "siggen.thread", 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.

---

[3] Several of these screen captures refer to 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.



Realtime function not linked

The HALSCOPE realtime sampling function must be called from a HAL thread in to determine the sampling rate.

Please do one of the following:

Select a thread name and multiplier then click 'OK'
or
Click 'Quit' to exit HALSCOPE

Thread: ------

Sample Period: ------

Sample Rate: ------

Thread	Period
siggen.thread	1.01 mSec
stepgen.thread	50.3 uSec

Multiplier: 1

Record Length

- ○ 16191 samples (1 channel)
- ○ 8095 samples (2 channels)
- ● 4047 samples (4 channels)
- ○ 2023 samples (8 channels)
- ○ 1011 samples (16 channels)

Figure 2.4: “Realtime function not linked” dialog

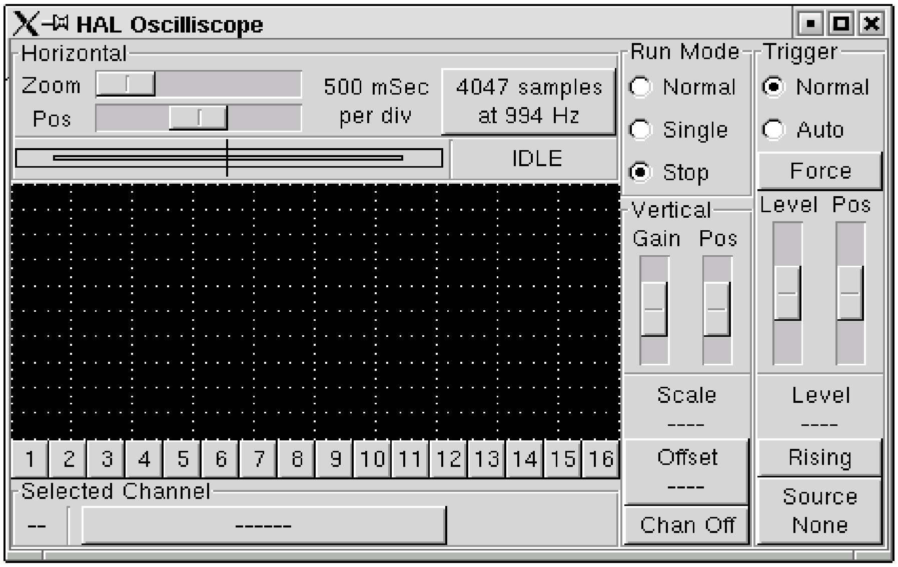
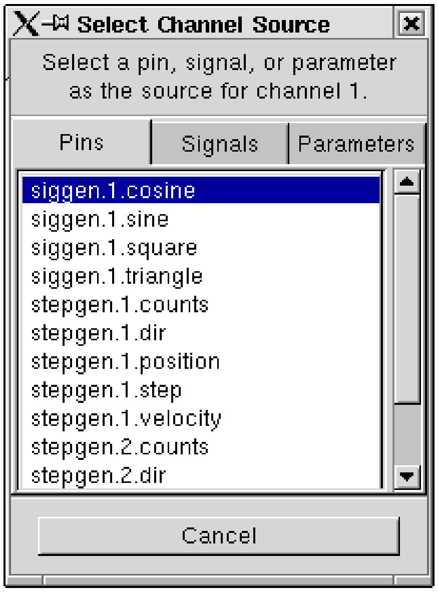


Figure 2.5: Initial scope window

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

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



Select Channel Source

Select a pin, signal, or parameter
as the source for channel 1.

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 on 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.1.triangle" and for channel 4, select "siggen.1.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 for 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.

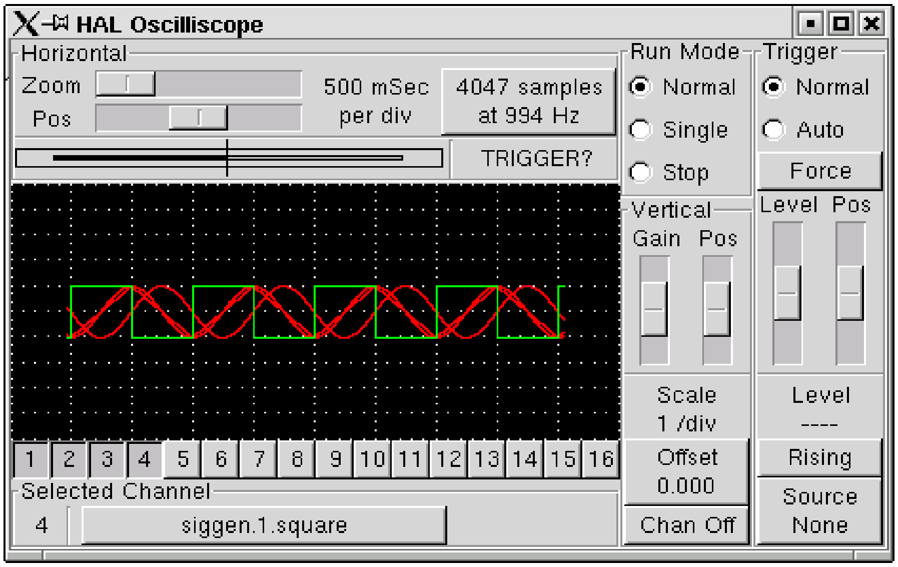


Figure 2.7: Captured Waveforms

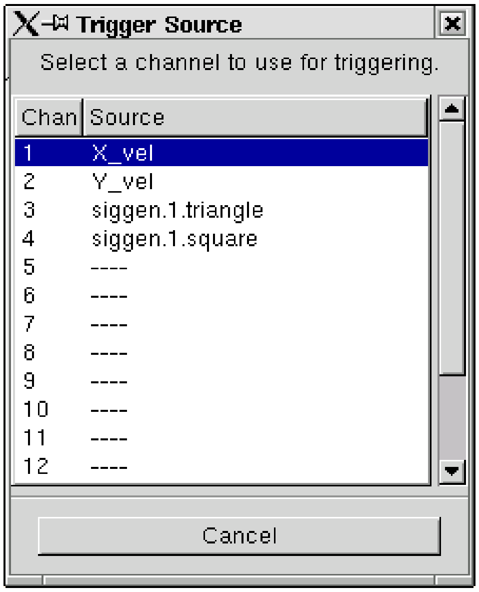
The "Selected Channel box at the bottom tells you that the green trace is the currently selected one, channel 4, which is displaying the value of the pin "siggen.1.square". Try clicking channel buttons 1 thru 3 to highlight the other three traces.

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 5.4 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.



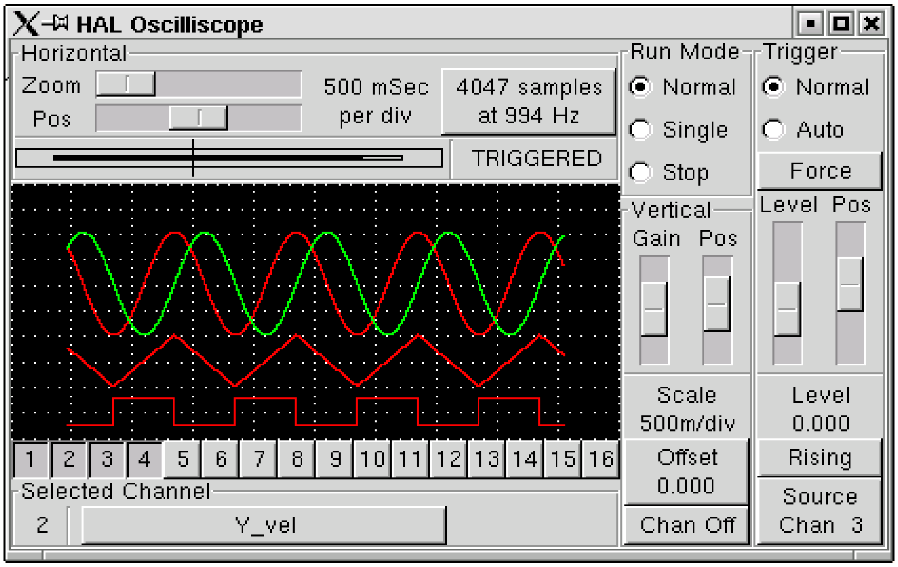
Select a channel to use for triggering.

Chan	Source
1	X_vel
2	Y_vel
3	siggen.1.triangle
4	siggen.1.square
5	----
6	----
7	----
8	----
9	----
10	----
11	----
12	----

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.



500 mSec per div

4047 samples at 994 Hz

TRIGGERED

Scale 500m/div

Offset 0.000

Level 0.000

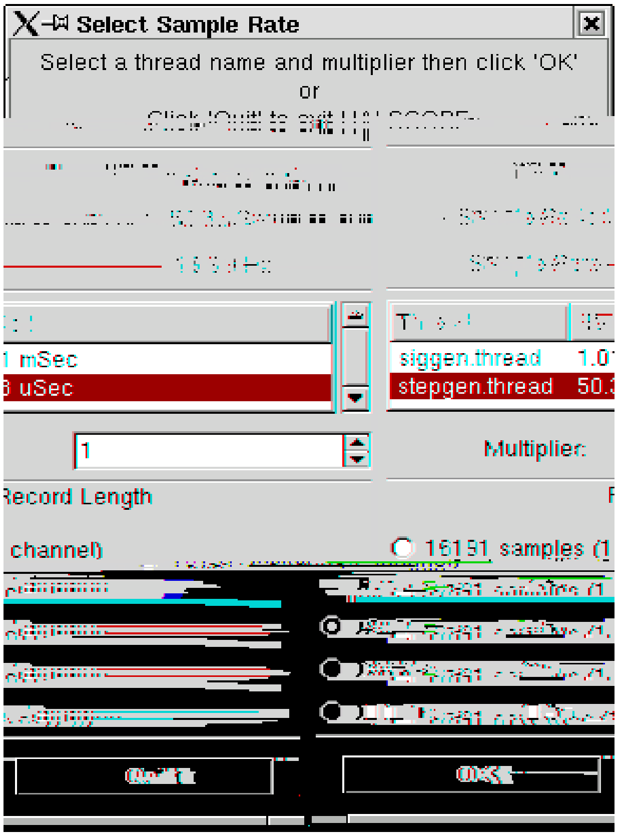
Source Chan 3

Selected Channel 2 Y_vel

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, "freqgen.thread", 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.

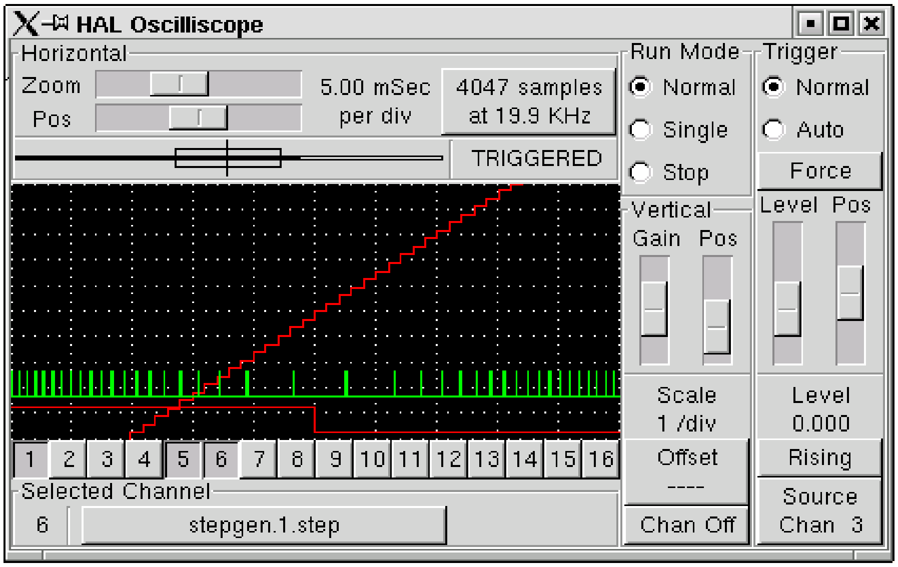


Select a thread name and multiplier then click 'OK'
or

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 "freqgen.1.dir", then channel 6, and select "freqgen.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.



5.00 mSec per div

4047 samples at 19.9 KHz

TRIGGERED

Run Mode: ● Normal ○ Single ○ Stop

Trigger: ● Normal ○ Auto

Scale 1 /div

Offset ----

Level 0.000

Rising

Source Chan 3

Selected Channel: 6 stepgen.1.step

Figure 2.11: Looking at Step Pulses

Chapter 3

Detailed Description of Internal Components

3.1 General Information

Each HAL component is described here in detail. This detail includes the input pins used and output pins generated by the component, the parameters accepted and generated by the component, and the functions available when the component is installed.

Notation

Command line examples are presented in **`bold typewriter`** font. Responses from the computer will be in `typewriter` font. Commands that require root privileges will be preceded by #, others by $. 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.

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 31 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.

For example, a general description of a pin name might be `parport.<portnum>.pin-<pinnum)-in`, where `<portnum>` and `<pinnum>` are specific numbers. So real names corresponding to this general description might be `parport.0.pin-10-in`, and `parport.1.pin-14-in`. Fields like `<portnum>` and `<pinnum>` will be explained in the adjacent text. When pins, signals, or parameters are described for the first time, their names will be preceeded by their type 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 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 3.1 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 thru 14 (various stepping patterns).

Installing

```
emc2# /sbin/insmod rtlib/stepgen.o cfg="<config-
string>" [period=<nsec>] [fp_period=<nsec>]
```

`<config-string>` is a series of space 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 cfg="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-string>` is "`0 0 0`" which will install three type '0' (step/dir) generators. The maximum number ofstep 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.

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

Removing

```
emc2# /sbin/rmmod 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).
- (`s32`) `stepgen.<chan>.count` – Feedback position in counts, updated by `capture_position()`.

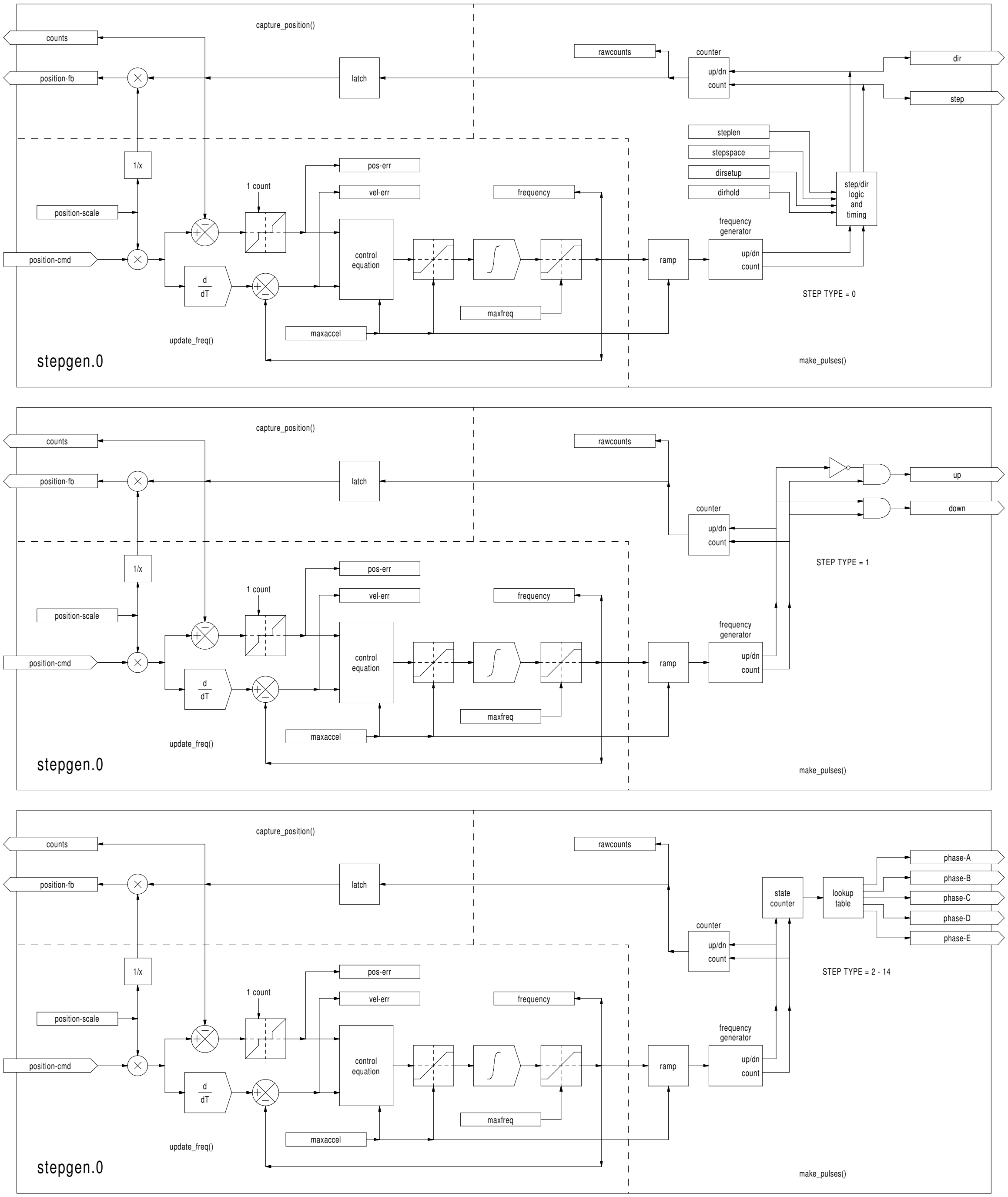


Figure 3.1: Step Pulse Generator Block Diagram

- `(FLOAT) stepgen.<chan>.position-fb` – Feedback position in position units, updated by `capture_position()`[1].
- `(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.

---

[1] There will eventually be additional pins or parameters to preset or reset `position-fb`, for homing purposes.

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 3.2 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.

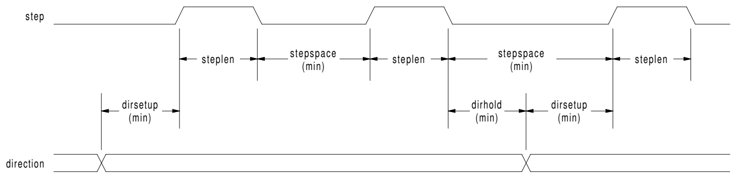


Figure 3.2: Step and Direction Timing

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 thru 14 are state based, and have from two to five outputs. On each step, a state counter is incremented or decremented. Figures 3.3, 3.4, and 3.5 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.

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

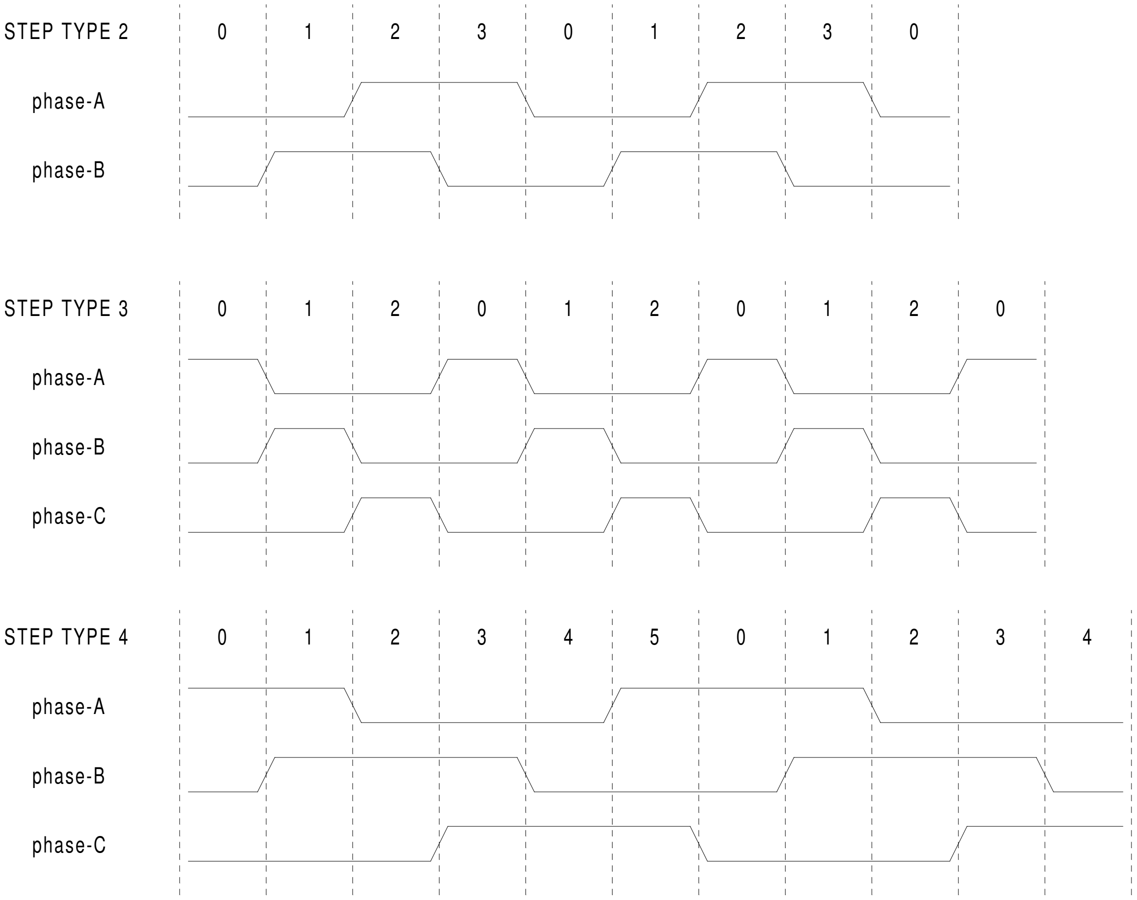


Figure 3.3: Quadrature and Three Phase Step Types

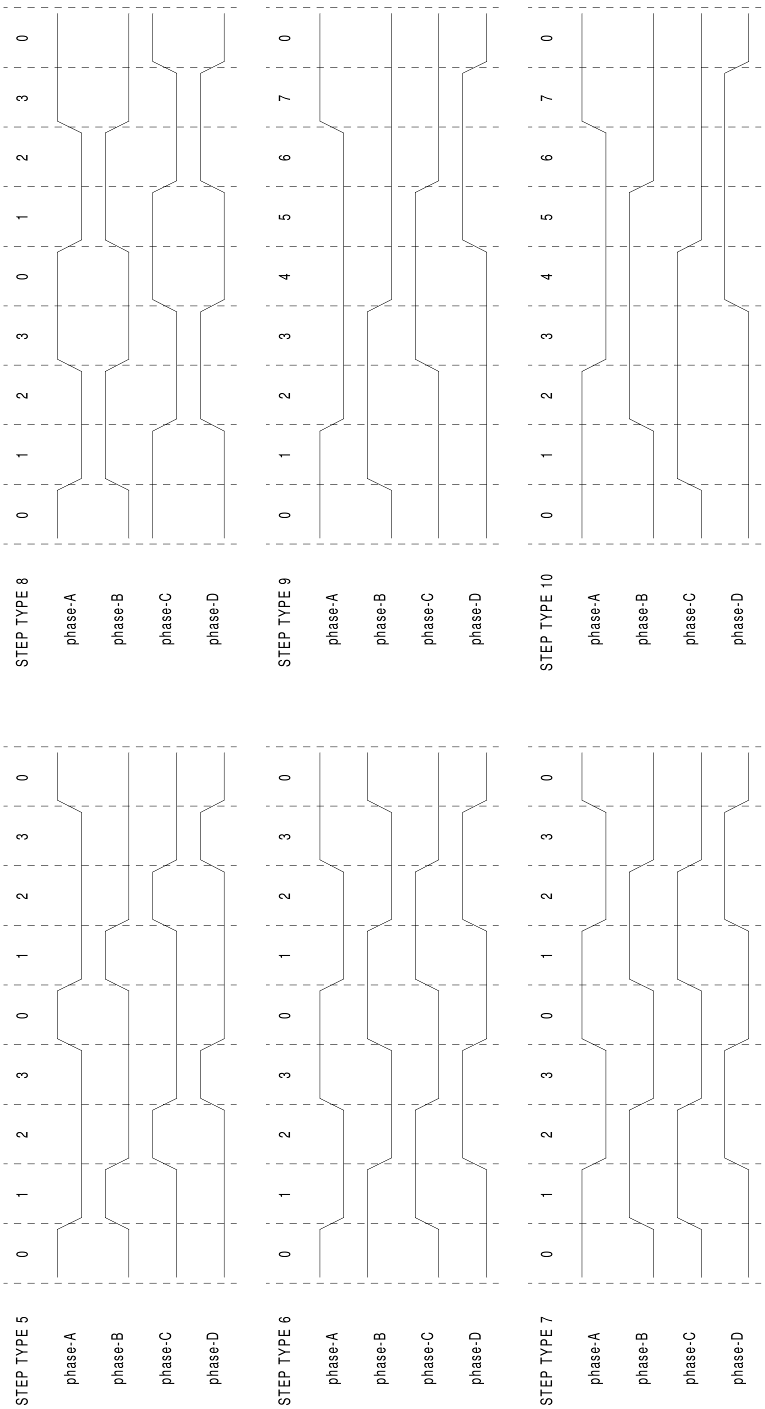


Figure 3.4: Four-Phase Step Types

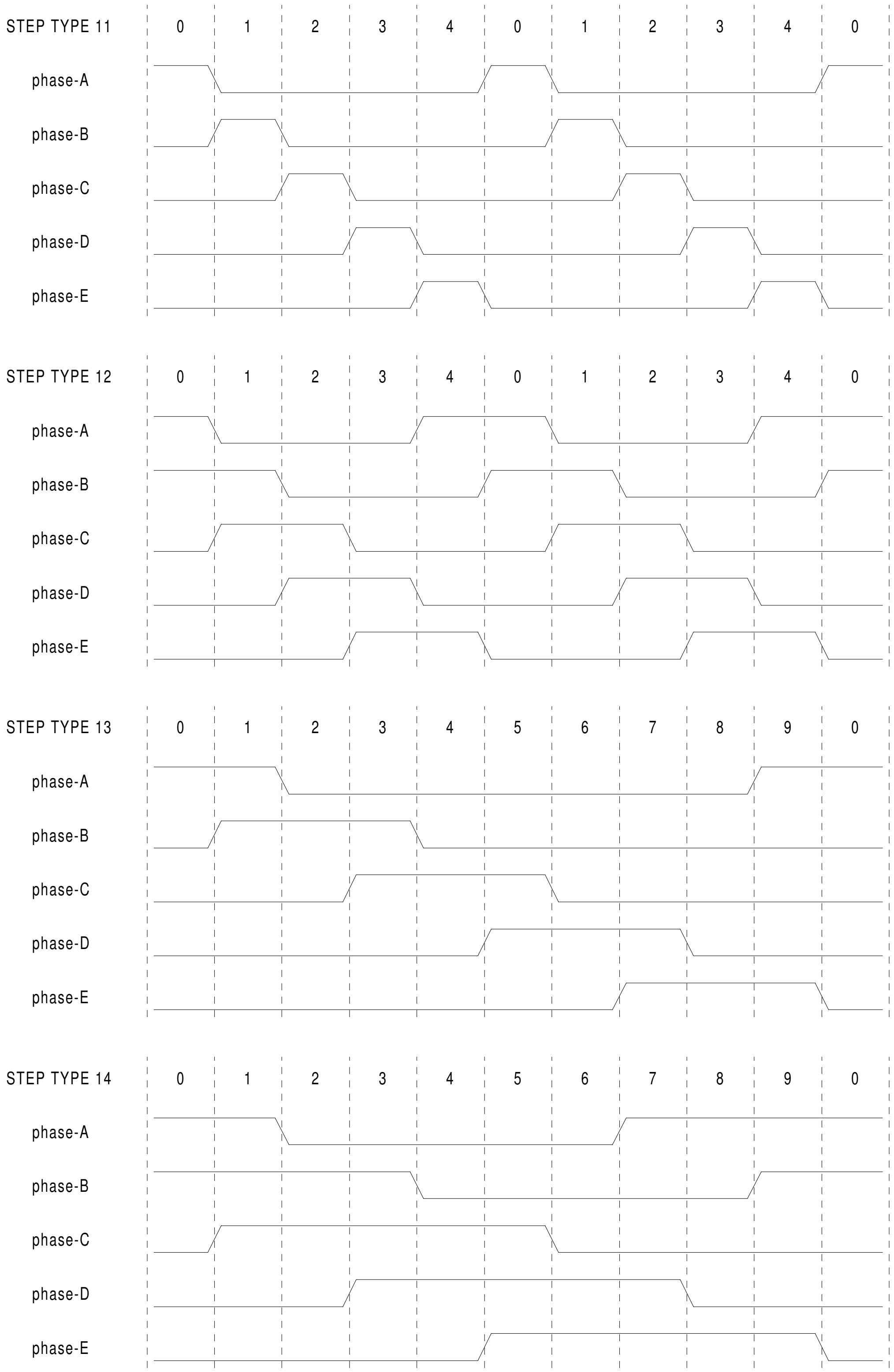


Figure 3.5: Five-Phase Step Types

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

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

3.3 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 3.2 is more appropriate. However, there may be applications where velocity based pulses are needed. 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 3.6 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 thru 14 (various stepping patterns).

Installing

```
emc2# /sbin/insmod rtlib/freqgen.o cfg="<config-
string>" [period=<nsec>] [fp_period=<nsec>]
```

`<config-string>` is a series of space 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# /sbin/insmod rtlib/freqgen.o cfg="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-string>` 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.

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

Removing

```
emc2# /sbin/rmmod 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).

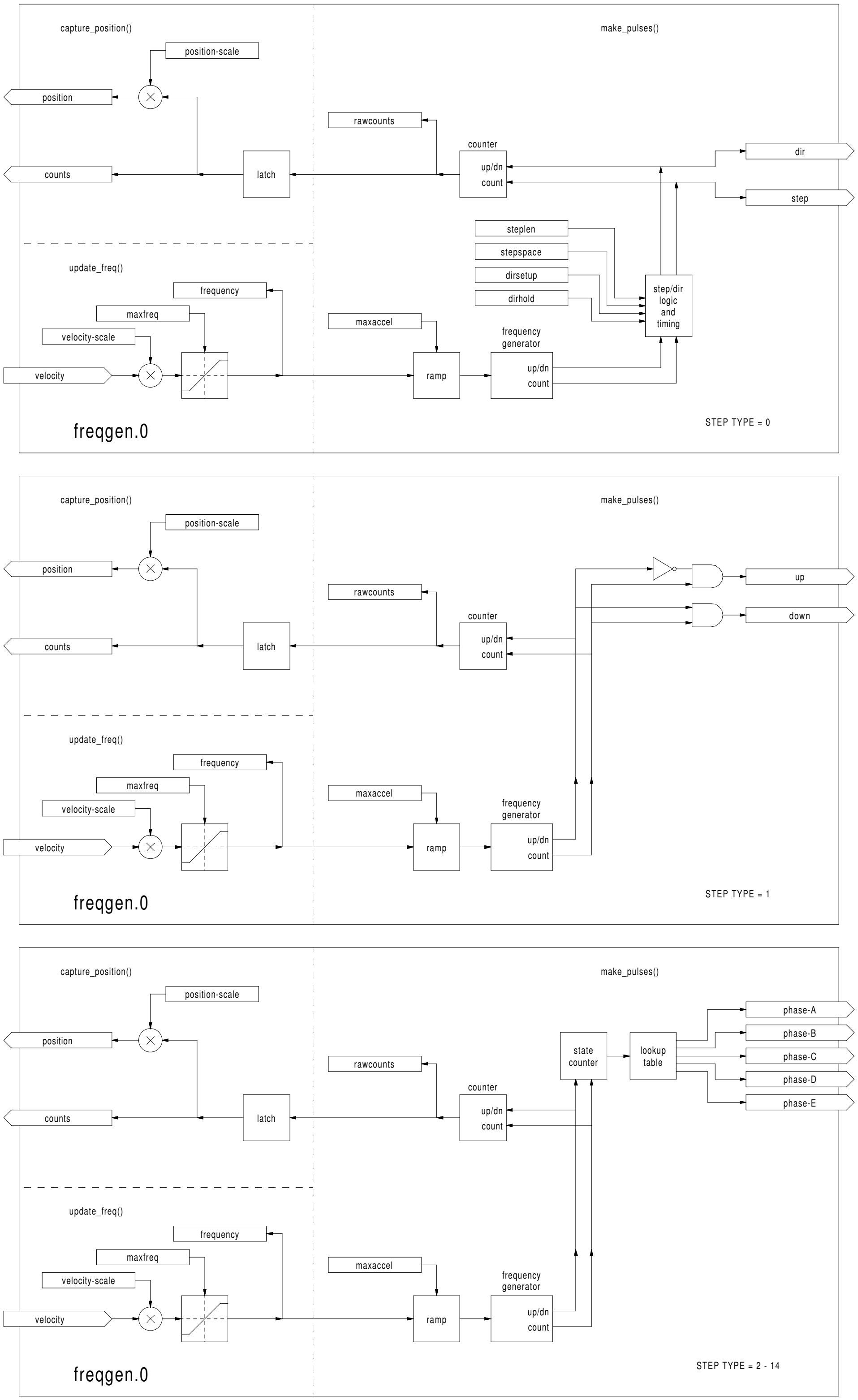


Figure 3.6: Step Pulse Generator Block Diagram

- (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).
- (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 step generator supports 15 different “step types”. Except for stepping type 1, they are identical those generated by the stepgen component (section 3.2). 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.

3.4 Encoder

This component provides software based counting of signals from quadrature encoders. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum count rates of 10kHz to perhaps 50kHz. Figure 3.7 is a block diagram of one channel of encoder counter.

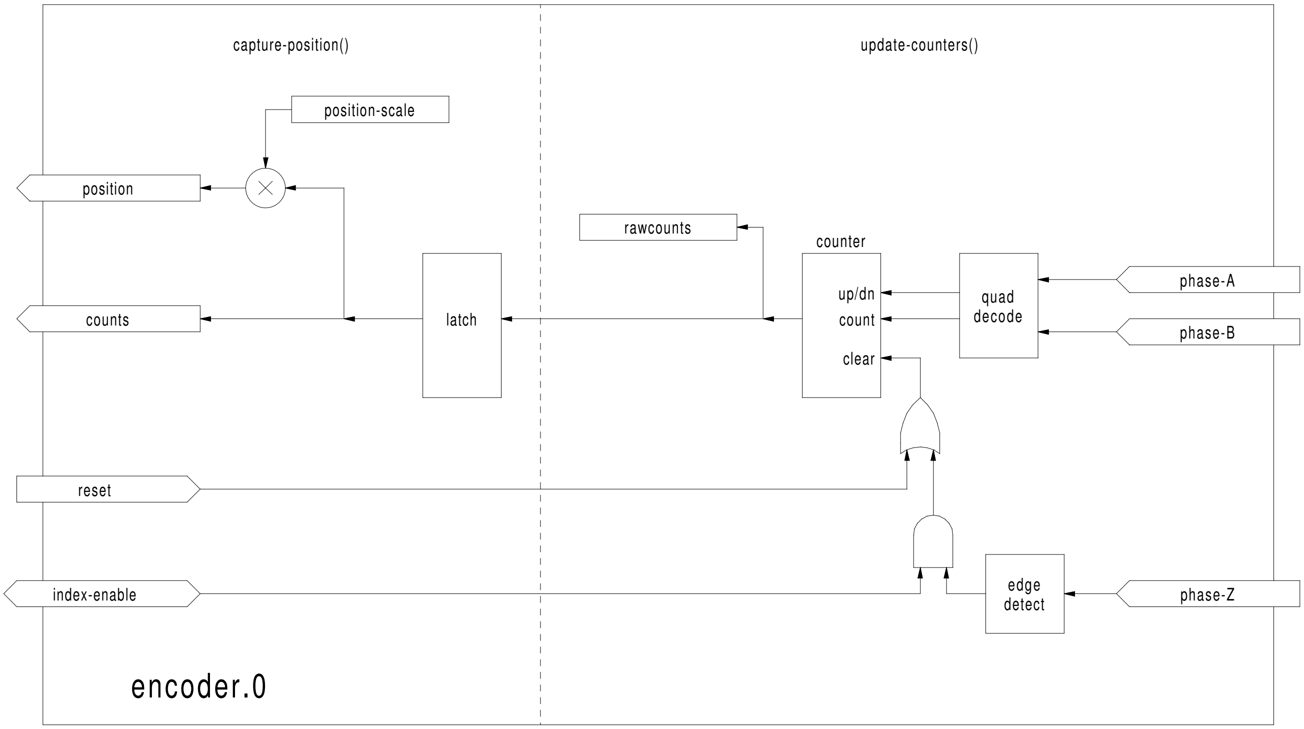


Figure 3.7: Encoder Counter Block Diagram

Installing

```
emc2# /sbin/insmod rtlib/encoder.o [num_chan=<counters>] [period=<nsec>]
```

`<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.

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

Removing

```
emc2# /sbin/rmmod encoder
```

Pins

- (`BIT`) `encoder.<chan>.phase-A` – Phase A of the quadrature encoder signal.
- (`BIT`) `encoder.<chan>.phase-B` – Phase B of the quadrature encoder signal.
- (`BIT`) `encoder.<chan>.phase-Z` – Phase Z (index pulse) of the quadrature encoder signal.
- (`BIT`) `encoder.<chan>.reset` – When TRUE, forces the counter to zero.
- (`BIT`) `encoder.<chan>.index-enable` – Enables reset to zero on phase Z rising edge.
- (`s32`) `encoder.<chan>.count` – Encoder value in counts, updated by `capture-position()`.
- (`FLOAT`) `encoder.<chan>.position` – Encoder value in position units, updated by `capture-position()`.

The `index-enable` pin is bi-directional. When set TRUE by another component, the next rising edge on `phase-Z` will reset the counter to zero. That rising edge will also reset `index-enable` to FALSE, so that the counter will only be reset once.[2]

Parameters

- (`s32`) `encoder.<chan>.raw-count` – The raw count value, updated by `count-pulses()`.
- (`FLOAT`) `encoder.<chan>.position-scale` – The scale factor used to convert counts to position units.

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.

---

[2] This behavior (and that of the reset input) may be changed based on discussions at NAMES.

3.5 PID

This component provides Proportional/Integeral/Derivative control loops. It is a realtime component only. For simplicity, this discussion assumes that we are talking about position loops, however this component can be used to implement other feedback loops such as speed, torch height, temperature, etc. Figure 3.8 is a block diagram of a single PID loop.

Installing

```
emc2# /sbin/insmod rtlib/pid.o [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# /sbin/rmmod 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, metres, or whatever is relavent. Likewise, for a 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`.

Figure 3.8: PID Loop Block Diagram

- (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.

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)[3].
- (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.

---

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

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 3.8, 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()`.

3.6 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# /sbin/insmod rtlib/debounce.o 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# /sbin/insmod rtlib/debounce.o 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# /sbin/rmmod 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[4].

- `(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.

---

[4] 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.

Functions

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

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

3.7 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.

Installing

```
emc2# /sbin/insmod rtlib/blocks.o <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# /sbin/indmod rtlib/blocks.o 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 hystersis
- 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
- 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

Removing

```
emc2# /sbin/rmmod siggen
```

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`) `comp.<num>.in0` – Inverting input of 2-input comparator.
- (`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.

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`.

3.8 Siggen

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

Installing

```
emc2# /sbin/insmod rtlib/siggen.o [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# /sbin/rmmod siggen
```

Pins

Each generator has four output pins.

- (FLOAT) `siggen.<chan>.sine` – Sine wave output.
- (FLOAT) `siggen.<chan>.cosine` – Cosine output.

- (FLOAT) `siggen.<chan>.triangle` – Triangle wave output.
- (FLOAT) `siggen.<chan>.square` – Square wave output.

All four 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 four outputs.

3.9 Supply

Chapter 4

Detailed Description of Hardware Drivers

4.1 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

```
emc2# bin/hal_parport 278 378 in 20A0 out
```

This example installs drivers for one port at 0x0378, with pins 2-9 as inputs, and two ports at 0x0278 and 0x20A0, with pins 2-9 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# /sbin/rmmod 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-nor` – 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 thru 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 thru 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 polarity 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.

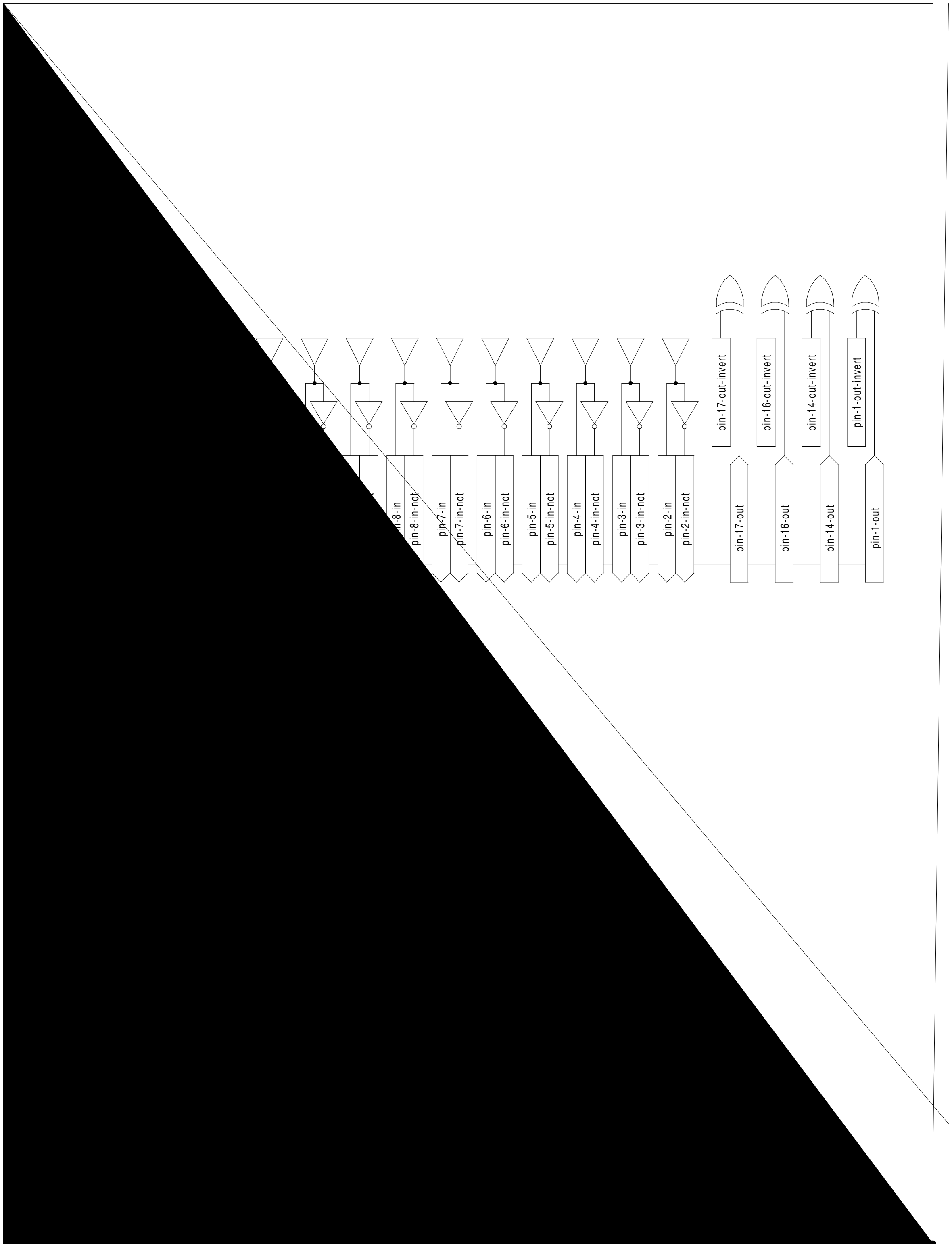


Figure 4.1: Parport Block Diagram

Chapter 5

Detailed Description of Utility Components

5.1 Halcmd

Halcmd is a command line tool for manipulating the HAL. Eventually this short paragraph will be expanded into a complete tutorial on how to use it. Unfortunately that will take more time than I have right now. However, in the meantime there is a rather complete man page for halcmd. If you run “make install” for emc2, it should be installed on your system. Even if it isn’t installed in your manpath, it is still accessible at as part of the emc2 CVS tree at “emc2/docs/man/man1/halcmd.1”. 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`.

```
-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.

```
-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 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 commands, and are ignored by halcmd.

```
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 mis-interpret 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.

```
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>
```

---

[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.

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>]
```

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]

5.2 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.

---

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

[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.

5.3 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.

5.4 Halscope