General Information on the Perl-based Electronics Interface

Steve Eichblatt Steve Furlanetto

August 11, 1998

1 Introduction

This document gives an overview of the Perl-based electronics control interface for the CDMS detector electronics. It contains a brief description of the interface program and its setup file, each of the software packages used in the interface (all of which are invisible to the casual user), and the manner in which commands flow between these various software packages. These descriptions are designed to introduce the principles behind the architecture and to aid in future upgrading and debugging of the software.

One of the strengths of the perl-based design is that it can function through the internet as described in ref [1]. This document will not explicitly describe the reasoning for such functionality, but it will (hopefully) illustrate how that design has been implemented in this architecture.

This document will also not describe the specific commands available for each type of electronics card. For information on those commands, see the relevant documentation (for example, in ref [2], ref [3], or ref [4]).

This documentcan be found at
`http://fnphyx-www.fnal.gov/experiments/cdms/software/GPIB/PerlInterface.ps`

2 Overall Architecture

In designing the interface system, we have taken advantage of Perl's capability for object-oriented programming. The idea behind object-oriented programming is quite simple: to have the software reflect the physical organization of the hardware. In general, one creates a single "package" or "module" for each physical or conceptual object in the system. Each package contains all the data necessary for control of the particular object it represents, and several routines, or "methods," for manipulating the object. The great advantage to such an object-oriented design is that it becomes very easy to upgrade one part of the system without affecting other parts, because the relevant code is cleanly separated from all other code.

Figure 1 is a diagram of the various packages used in this interface.

Because of the modularity in this software, we have made extensive use of *inheritance*, implemented through *'is-a' relationships.* In many cases, low-level objects share a certain amount of functionality (for example, `RTFcard.pm`, `BLIPcard.pm`, and `ZIPcard.pm` each represent 9U electronics modules; as such, they have some routines in common). Therefore, routines shared by several packages are included in a higher-level package, and the lower-level packages inherit these routines from the high-level packages. This relationship is shown in Figure 1 as a solid line, with the solid dot labeling to the sub-class.

The other important type of relationship between modules is the *"has-a" relationship'*, shown in Figure 1 with a dotted line. This relationship reflects the fact that some objects contain others (for example, and electronics card will contain several digital objects such as DACs and control registers). Rather than build the smaller objects into the larger objects several times, the larger object simply calls the package representing the smaller object, which performs all of the operations on that object. The end of the dotted line in Figure 1 pointing to the contained object has an integer giving the number of contained objects of that type.

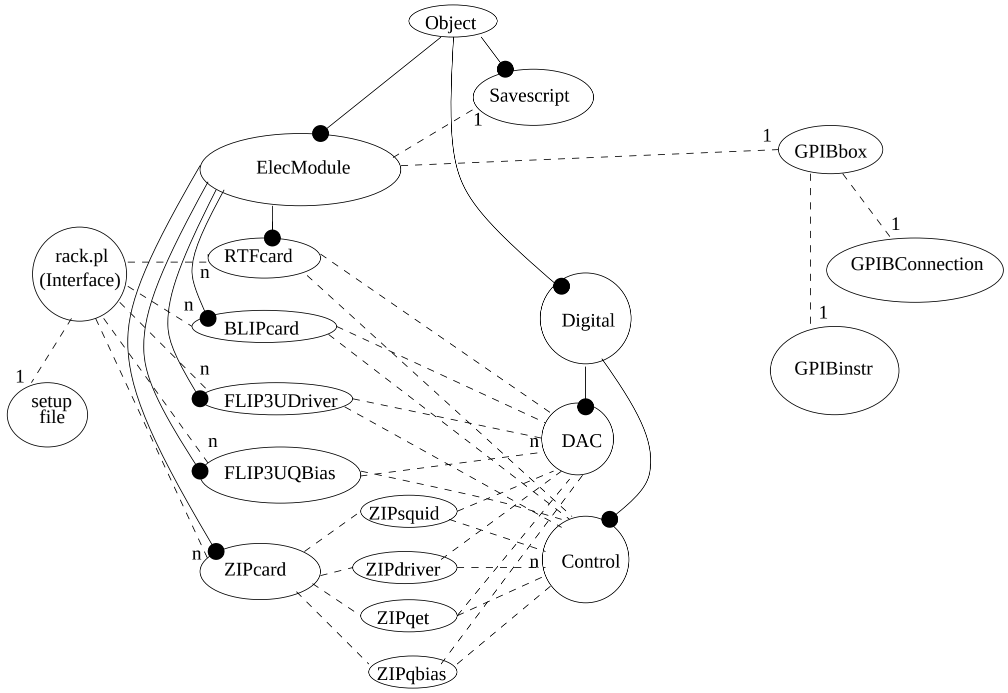

Figure 1: *A diagram of objects in the Perl interface. Solid lines represent 'is-a' relationships, and dotted lines represent 'has-a' relationships. Each object is a separate file, with the extension* `.pm`*. The only exceptions to this naming convention are the* `rack.pl` *file and the setup file.*

Two caution are in order at this point. First, the `rack.pl` user interface program is not actually an object, and does not create any objects save those existing from other packages. It is, therefore, not technically object-oriented. This is important only to the programmer debugging or upgrading `rack.pl`, because it should be transparent to the user (the interface "feels like" it is object-oriented). Second, it is not quite true that each electronics module contains its own `GPIBbox`. In fact, all of the modules on a subrack share a single `GPIBbox`.

3 Command Flow

Because of the extensive modularity of the software architecture, the flow of input commands is somewhat complicated. It is illustrated in Figure 2.

The path begins when the user types a command into the interactive terminal session, created with the `rack.pl` program. With a few exceptions, this program does not process any commands, it simply pipes them through to the package representing the currently selected electronics card. This package contains a set of routines which point the command to the correct digital object (or objects) which the command affects. It then sends the command to the package representing those digital objects, which translates the command into the hexadecimal code to be sent to the GPIB box.

Once this translation takes place, the command is sent to the `ElecModule.pm` package, which contains routines for sending information to the GPIB box. Once the `GPIBbox.pm` module receives the command, it places it into a `GPIBinstr.pm` slot (short for GPIB instruction), which fills out the syntax requirements for communication with a real GPIB box. The completed command is then piped to the internet through the `GPIBconnection.pm` module and the TCP server until it reaches the physical GPIB box. This box then distributes the message to the appropriate electronics card.

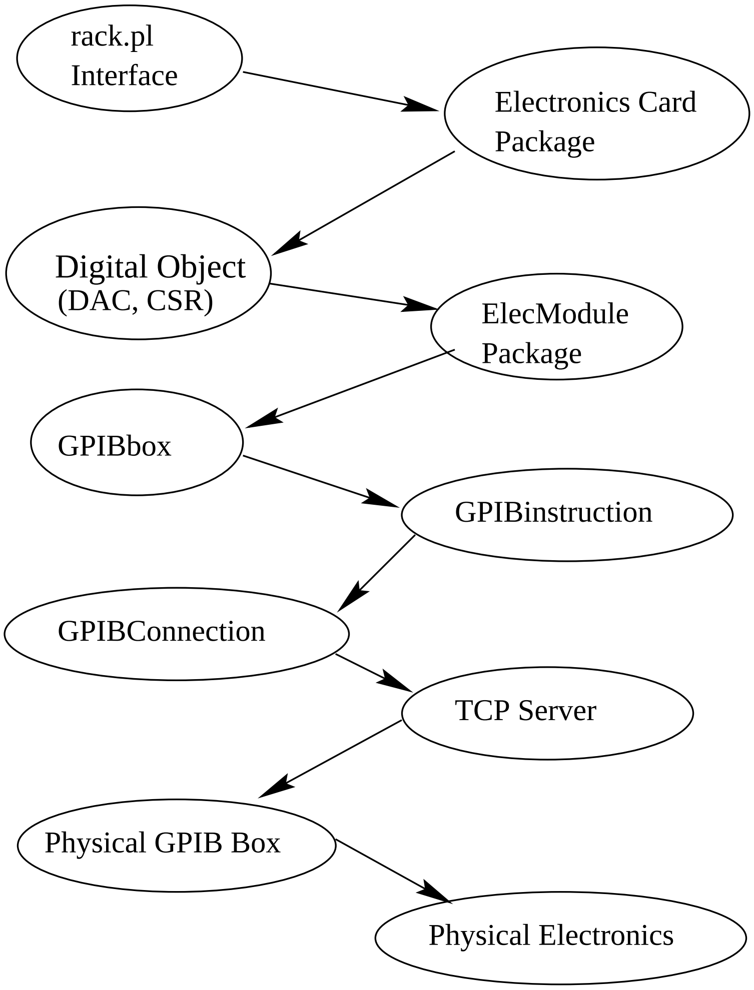

Figure 2: *The flow of commands between software packages and physical objects.*

4 The `rack.pl` Terminal Interface

The `rack.pl` interface is the executable program which allows interactive user control of the electronics. It is essentially the "master" program, which creates all of the objects to be controlled. To begin a session, log on to your account (on any terminal), move to the appropriate directory, and type

`<prompt> rack.pl -i setup.rack`

Here, `setup.rack` is the name of a file containing a rack configuration (see below). In addition, there are several optional arguments that can be entered on the command line. These optional arguments and the setup file name is processed through a command line processor contained in `rack.pl`. The syntax for command line options is `-(character)␣argument`. Possible command line arguments are:

-h **String** `hostname = current host`
Sets location of TCP internet server to `hostname`.

-i **String** `filename`
This item is required!. Sets name of rack configuration file.

-m **int** `module=8`

Sets initially selected module to `module`.

-p **int** `port=2345`
Sets port on TCP internet server to `port`.

-r **int** `subrack=3`
Sets initially selected subrack to `subrack`.

-s **void**
Disables communication to server. Designed to be used for scripting processes.

4.1 The Configuration File

The first thing that `rack.pl` does is to create a "rack" of electronics cards. It does this according to the configuration input through the `-i` switch. The configuration file is a file containing two lines for each subrack to be created. The first line gives the number of the subrack, and the second line lists the cards contained in that subrack. Up to 8 subracks, numbered through 7, may be included in a single rack. The format is as follows:

```
subrack #\n
CARDTYPE,CARDTYPE,CARDTYPE,...\n
subrack #\n
CARDTYPE,CARDTYPE,CARDTYPE,...\n
end
```

Note that the list of card types must be on a single line. The allowed card type abbreviations are:

B BLIP card

F3D FLIP3UDriver card

F3QB FLIP3UQBias card

R RTF card

Z ZIP card

The `end` statement is optional and may be omitted.

For those familiar with Perl syntax, the rack is created as a 2-dimensional array named `@boards`. The first index refers to the card's subrack, and the second to its module position.

4.2 The `rack.pl` Command Environment

Once `rack.pl` has created the rack configuration, it loads an interactive Perl interpreter. Once this is complete, the user should see something like the following:

```
> rack.pl -i test.rack
Host is ppdm06.fnal.gov:2345.
Using subrack 3, set to module 5.
subrack 1
Z, B, R, Z, B,R
subrack 3
R,Z,B, R,Z,B
got autohistory
zip r3,m5: _
```

Here, the final line is the prompt followed by the cursor. The prompt indicates the selected card, according to the following syntax:

`CARDTYPE rSUBRACK,mMODULE:`

4.2.1 Changing the Module and Subrack

All commands are issued directly to this card (with the exception of macro commands, see below). The currently selected module can be changed with the command `module` **int**, and the currently selected subrack can be changed with the command `subrack` **int**. The prompt will be automatically changed when these commands are executed (if the `CARDTYPE` field is empty, then there is no card in the currently selected module).

4.2.2 The `mode` Command

The only way to send commands to cards other than the currently selected one is through the `mode` **String** command. The argument of this command is a card type (`rtf`, `blip`, `zip`, etc.). When a particular mode is selected, commands sent to one card of the type given by **String**, it is also sent to all other cards of that type. This feature can be turned off by entering `mode normal`.

4.2.3 Command execution

Once a command is entered, `rack.pl` executes it according to the following set of rules.

1. If the input ends with `.pl`, then the interpreter assumes the command is a Perl script and executes it.
2. If the input begins with `module` or `subrack`, it changes the selected module or subrack.
3. If the input begins with `mode`, it sets the mode.
4. If the input contains a space, it is executed as a standard Perl command.
5. If the input matches a macro command, the command is executed for every card in the rack (see below).
6. Otherwise, the interpreter prepends `$boards[SUBRACK][MODULE]->` to the command and executes it (this sends the command to the currently selected card).

4.2.4 Macro Execution

The user is likely to want to apply a certain subset of card commands to all of the cards in the rack. To accomodate this, `rack.pl` contains a list of possible macro commands. When one of these commands is entered, `rack.pl` will execute the command for all of the cards. Currently enabled macros are: saveScript, endScript, enableBox, disableBox, accumulate, unaccumulate, and execute. Macros can be added simply by modifying the `@macros` array in `rack.pl`.

5 The Software Packages

This section contains a brief description of the functions contained in each of the Perl modules (those with `.pm` file extensions). Each item lists the modules from which that package inherits, and those modules which that package contains.

object *Inherits from:* None
Contains: None
Contains basic contructor, help routines, and an AUTOLOAD function for smooth running.

Savescript *Inherits from:* object
Contains: None
Includes 3 routines, all of which relate to creating a hex script: the first opens a file, the second adds a single line to the open script containing the formatted hex code, and the third routine closes the script.

ElecModule *Inherits from:* object
Contains: Savescript, GPIBbox
ElecModule contains routines common to all electronics cards. These fall into four general categories: routines for sending messages to (and receiving messages from) the GPIB box, commands for saving scripts and enabling or disabling communication with the server, routines for retrieving basic information from the card (such as the module type or serial number), and routines for card persistence (for a description of how this feature works, see the next section).

BLIPcard *Inherits from:* ElecModule
Contains: Control, DAC
Contains commands for setting and reading back the digital DACs and controls on the BLIP card. Also contains routines for printing the card settings.

FLIP3UDriver *Inherits from:* ElecModule
Contains: Control, DAC
Contains commands for setting and reading back the digital DACs and controls on the FLIP3U Driver card. Also contains routines for printing the card settings.

FLIP3UQBias *Inherits from:* ElecModule
Contains: Control, DAC
Contains commands for setting and reading back the digital DACs and controls on the FLIP3U QBias card. Also contains routines for printing the card settings.

RTFcard *Inherits from:* ElecModule
Contains: Control, DAC
Contains commands for setting and reading back the digital DACs and controls on the RTF card. Also contains routines for printing the card settings.

ZIPcard *Inherits from:* ElecModule
Contains: ZIPdriver, ZIPqbias, ZIPqet, ZIPsquid
Contains commands for setting and reading back the digital DACs and controls on the ZIP card. Also contains routines for printing the card settings.

ZIPdriver *Inherits from:* ElecModule
Contains: Control, DAC
This object simply contains the controls and DACs for the driver section of the ZIP card. Other than the constructor and a print routine, it does not contain any methods. (It could possibly used for combining the 3U and 9U versions of the ZIP card in the software).

ZIPqbias *Inherits from:* ElecModule
Contains: Control, DAC
This object simply contains the controls and DACs for the QBias section of the ZIP card. Other than the constructor and a print routine, it does not contain any methods. (It could possibly used for combining the 3U and 9U versions of the ZIP card in the software).

ZIPqet *Inherits from:* ElecModule
Contains: Control, DAC
This object simply contains the controls and DACs for the QET section of the ZIP card. Other than the constructor and a print routine, it does not contain any methods. (It could possibly used for combining the 3U and 9U versions of the ZIP card in the software).

ZIPsquid *Inherits from:* ElecModule
Contains: Control, DAC
This object simply contains the controls and DACs for a SQUID section of the ZIP card. Other than the constructor and a print routine, it does not contain any methods. (It could possibly used for combining the 3U and 9U versions of the ZIP card in the software).

Digital *Inherits from:* object
Contains: None
Contains routines common to all digital objects on an electronic card. All of these routines handle the persistence mechanism for each digital object (see the next section for a description of the persistence feature).

Control *Inherits from:* Digital
Contains: None
This package contains routines for initializing a control register and for translating the user's command into the appropriate hex code for the CSR.

DAC *Inherits from:* Digital
Contains: None
This package contains routines for initializing a DAC and for translating the user's command into the appropriate hex code for the DAC.

GPIBbox *Inherits from:* None
Contains: GPIBconnection, GPIBinstr
This represents the GPIB box for a particular subrack. It contains initialization and help routines in addition to routines for setting up the GPIBinstr and GPIBconnection to send the hex code through the server to the physical GPIB box.

GPIBinstr *Inherits from:* None
Contains: None
This represents an instruction slot for the GPIB box, and contains routines for concatenating the data string with the appropriate header and footer elements, with the correct format.

GPIBconnection *Inherits from:* None
Contains: None
This package handles all of the communication between the software and the internet connection. It contains routines for opening and closing the socket connection, sending datastrings to the server (in two different ways: one at a time, or by creating a queue of instructions and sending all of the commands at once), and for reading datastrings from the server.

6 Persistence

The persistence feature is designed to guarantee that the computer memory retains the correct card settings between sessions. This is necessary because if the computer or internetconnection crashes, it is necessary to reload the software. When the user reloads the software, he or she would like to continue with the same command series, because the actual configuration of the board will not have changed. Therefore, the interface creates a so-called "persistence" file for each board. When the software is loaded, it reads each of these persistence files and sets the data memory of each DAC and CSR appropriately. Thus, a command series can continue between sessions.

The first time a command is issued to a particular card, the software creates a persistence file for that card. This file is named `/tmp/HOST-CARDTYPE-SUBRACK-MODULE.out`. It contains one line for each DAC or control register on the card, in the following format:

```
SUBRACK ADDRESS DATAWORD 0000 % CARDTYPE
```

Each of the first three entries is a 4-digit hexadecimal word. (The first two digits of the address field give the module number, and the last two digits give the location on the card). The `0000` and `CARDTYPE` fields are included for compatibility with scripting.

Several cautions are in order when using persistence:

1. Cards should not be controlled through two different host servers; they will not share persistence files.
2. If communication with the GPIB box is disabled, the software does not interact with the "normal" persistence file. Instead, it creates a new persistence file named `/tmp/script-CARDTYPE-SUBRACK-MODULE.out`. This is because, if communication with the GPIB box is disabled, the actual card is unaffected by commands. However, if one wishes to create scripts in this mode, the continuity of commands allowed by persistence is necessary. Normally, this feature will be transparent to the user.
3. Similarly, if the "accumulate" mode is turned on (for a description of this mode, see, for example, ref [4]), the software creates a new persistence file `/tmp/accum-CARDTYPE-SUBRACK-MODULE.out`. When commands are executed, this persistence file is copied over the "normal" persistence file. Thus, this feature keeps track of both the command-persistence and the actual card persistence, so it should be transparent to the user.
4. Persistence will also function smoothly with the `readscript.pl` program described in the next section provided only that persistence files exist for each card called during the script. There is a slight danger in this process; see below for details.

7 Scripting

Another useful feature of the Perl software interface is its ability to create scripts for easy and fast execution of common operations. Scripting can be enabled by executing a `saveScript` command from within the interactive session, and it is stopped with an `endScript` command (see ref [4] for a description of these commands). When scripting is enabled, every time a command is executed a line is appended to the script file, with the following format:

```
SUBRACK ADDRESS DATAWORD READ/WRITE COMMENT}
```

The first four fields are all four digit hexadecimal words. The `SUBRACK` lists the card's subrack, the first two digits of the `ADDRESS` list the module number, the second two digits of the `ADDRESS` list the card address of the digital object to be set, the `DATAWORD` field contains the data to be sent to that address, and the `READ/WRITE` field contains a `0001` for a read command and a `0000` for a write command. The `%` indicates the beginning of the comment, and the `COMMENT` describes the operation to be performed. It is free format, except that the first word of the comment must label the card's type (options are currently BLIP, FLIP3UDriver, FLIP3UQBias, RTF, and ZIP). (Persistence within the Perl script-reading program requires this comment form.)

Once created, scripts can be read and executed using the `readscript.pl` executable program. This program is called using the following syntax:

```
<prompt> readscript.pl SCRIPTNAME ECHO
```

Both fields are optional, as the user will be prompted for them if they are omitted. The `SCRIPTNAME` field should contain the name of the script to be executed. The `ECHO` field should contain a `y` if the user wishes the executed operations to be printed on the screen, or an `n` otherwise.

`readscript.pl` cycles through each line of the script, adds commands to an instruction queue, and executes that queue at the end of the script (or whenever it encounters a "read" command: in that case, the program executes its queue, performs the read command, and then begins refilling the queue). Each time a command is read, the program also updates the "normal" persistence file. Note, however, that the command is not actually executed until later in the execution. Therefore, if the execution is interrupted, persistence may not function properly. We recommend re-executing the script if this does occur; that will guarantee that the persistence remains correct.

Note that because the format of persistence files matches the format of scripts, persistence files can also be used as scripts (This is essentially what the `powerUp` command does).

References

[1] Steve Eichblatt, Sae Woo Nam CDMS note 9712002. *The CDMS Net Protocol*

[2] Steve Eichblatt, Steve Furlanetto *The BLIP card Driver*

[3] Steve Eichblatt, Steve Furlanetto *The RTF card Driver*

[4] Steve Furlanetto *The ZIP card Driver*