# High voltage control software for Miniball (INTERNALS)

This document only makes sense in conjunction with the HV control source code.

You should only need to refer to this documentation if you want to modify the HV control software or understand how its innards work.

Normal users should refer to the users' manual instead.

[1] Nigel Warr March 2007

# Contents

# 1 Introduction

The software is divided up into two main programs (hv_server and hv_control) and a few small programs and scripts.

The server runs in background, logging to the system log and writing periodically to a status file. It constantly monitors the master configuration file (which tells it which mainframe to connect to) and the limits configuration file, which sets the names and limits for each channel, and re-reads them if the files change on disk. It also reads commands from a pipe.

It is possible to send commands directly to the pipe from a shell, script or program. Any user can write to the pipe. The server will then decide whether to obey those commands based on the policy. The policy limits the maximum voltage to that set in the limits file (which should be world readable, but should not be writable by normal users). This file also determines the ramp up and ramp down rates, as well as the conditions for a channel to trip. Furthermore, there is a hardcoded limit to the maximum increase in voltage in a single step, which is dependent on the measured voltage.

It is intended for all the security to be implemented in the server, so that normal users cannot make unsafe changes to the voltage.

The control program is a GUI, which monitors the status file produced by the server and sends commands to the command pipe, whenever the user clicks or slides.

The scripts send commands directly to the command pipe, making it possible to make changes directly from the command line, which is useful for slow connections (e.g. when the server is in CERN).

# 2 The server hv_server

The server consists of the following files:

- hv_server.c - main entry point
- master.c - code to read master configuration file
- mainframe.c - code to interface with HV mainframe
- limits.c - code to read limits configuration file
- channel.c - code to configure channels
- logmessage.c - code for logging to system log file
- parse_cmd.c - code to parse command lines

It is designed to run in background mode, for example, started at system startup. It does not normally write any output to the screen, but instead uses the system log file.

These are normally linked to libhscaenet.so and libcaenhvwrapper.so provided by CAEN, or for the fake server (which simulates a high voltage mainframe) fake_libcaenet.c is used instead.

The server continually watches the master and limits configuration files. If the modification date of one of these files changes, the file is read again and the modifications are applied at once.

The server also reads from a named command pipe, which can be use to send commands to the server. It is, in principle, possible to send commands directly to the server from the command pipe, without using the graphical user interface.

The server continually writes the current values to a file.

If a channel gets a warning, a script is called. If a channel trips, a different script is called. These scripts may be used to perform any actions the user deems appropriate.

## 2.1 hv_server.c

This file the main entry point to the program. It has just two functions:

- int main(int argc, char *argv);
- void signal_handler(int signum);
- void exit_handler();
- int write_empty_data_file();

### 2.1.1 main

This is the main entry point. First we log that we are starting to the system logger. Then we zero the structure containing the channel data. Next we set up the signal handler, so that it is called for all signals except SIGKILL (which cannot be trapped) and SIGHUP, SIGWINCH and SIGCHLD, which are ignored. Then we set up the exit handler, which is called when the program exits and enter the main loop. The main loop is broken out of, if a signal arrives which causes the signal handler to be called. This calls *exit* which in turn causes the exit handler to be called and exits the program.

The main loop does the following operations over and over, with a small sleep to prevent overuse of processor time:

- Check the master configuration file for changes by calling *master_check_for_changes*.
- Check if we need to connect to the mainframe by calling *mainframe_connect*.
- Get the values from the mainframe by calling *mainframe_get_values*
- Check the limits configuration file for changes by calling *limits_check_for_changes*
- Check for commands on the command pipe by calling *channel_check_for_command*

#### 2.1.2 signal_handler

This function is the signal handler which is called for any signal except SIGHUP, SIGWINCH, SIGCHLD and of course, SIGKILL (which can't be trapped).

It logs the event to the system log, and exits, which causes the exit handler to be called. The exit handler disconnects from the mainframe.

#### 2.1.3 exit_handler

This function is called whenever the program exits via the *exit* function. Since the signal handler calls exit, this means it is called for any signal except SIGHUP, SIGWINCH, SIGCHLD and of course, SIGKILL (which can't be trapped).

It logs the event and disconnects from the mainframe. It also writes an empty data file by calling *write_empty_data_file*.

#### 2.1.4 write_empty_data_file

This function is called when we start and when we stop. It writes a data file with just a timestamp and no other data, overwriting whatever was there before. Note that this empty data file is overwritten as soon as we read genuine values from the HV mainframe.

### 2.2 master.c

The code in this file is used to monitor and read the master configuration file. There are two main functions:

- int master_check_for_changes();
- int master_read_file();

#### 2.2.1 master_check_for_changes

Each time this function is called, it checks the modification time of the master configuration file. If this has changed since the last time, the function was called, it calls *master_read_file* to read it. N.B. the very first time the function is called, after the program is started, it always does this. It is called on each iteration of the main loop.

#### 2.2.2 master_read_file

This function opens the master configuration file and reads it line by line, using the *parse_cmd* functions to parse it.

Then there are five callback functions, which are called when *master_read_file* finds the corresponding entry in the master configuration file:

- master_parse_sysname
- master_parse_ip
- master_parse_username
- master_parse_password
- master_parse_loglevel

#### 2.2.3 master_parse_sysname

This function is called from *master_read_file* via *parse_cmd_line* if a SYSNAME entry is found in the master configuration file. It stores the new system name and disconnects from the mainframe. On the next iteration of the main loop, *mainframe_connect* will reconnect using the new name.

#### 2.2.4 master_parse_ip

This function is called from *master_read_file* via *parse_cmd_line* if an IP entry is found in the master configuration file. It stores the IP address and disconnects from the mainframe. On the next iteration of the main loop, *mainframe_connect* will reconnect using the new IP address.

#### 2.2.5 master_parse_username

This function is called from *master_read_file* via *parse_cmd_line* if a USERNAME entry is found in the master configuration file. It stores the username and disconnects from the mainframe. On the next iteration of the main loop, *mainframe_connect* will reconnect using the new username.

#### 2.2.6 master_parse_password

This function is called from *master_read_file* via *parse_cmd_line* if a PASSWORD entry is found in the master configuration file. It stores the password and disconnects from the mainframe. On the next iteration of the main loop, *mainframe_connect* will reconnect using the new password.

#### 2.2.7 master_parse_level

This function is called from *master_read_file* via *parse_cmd_line* if a LEVEL entry is found in the master configuration file. It calls *logsetlevel* to set the new logging level.

### 2.3 mainframe.c

This file contains the code to interface to the high voltage mainframe. The functions are:

- int mainframe_connect(char *ip, char *username, char *password);
- int mainframe_disconnect();
- int mainframe_set_voltage(int channel, float request_voltage);
- int mainframe_enable_channel(int channel);
- int mainframe_disable_channel(int channel);
- int mainframe_get_values();
- int mainframe_set_channel(int channel);
- int mainframe_set_values();
- int mainframe_set_limits();
- float mainframe_calculate_max_safe_voltage(float measured_voltage, float max_voltage);
- int mainframe_check_values(hv_channel *hv, int n);
- int mainframe_analyse_status(int n, int old_status, int new_status);
- int mainframe_call_trip_script(int n, int old_status, int new_status);
- int mainframe_call_warn_script(int n, int old_status, int new_status);

#### 2.3.1 mainframe_connect

If the program is already connected to the mainframe, this function does nothing, otherwise it attempts to establish a new connection. It is called on each iteration of the main loop.

#### 2.3.2 mainframe_disconnect

If the program is not connected to the mainframe, this function does nothing, otherwise it attempts to break the connection.

#### 2.3.3 mainframe_set_voltage

Set the request voltage for a given channel. It sets a value and calls *mainframe_set_values* to do the work.

#### 2.3.4 mainframe_enable_channel

Enable a given channel. It sets a flag and calls *mainframe_set_values* to do the work.

#### 2.3.5 mainframe_disable_channel

Disable a given channel. It sets a flag and calls *mainframe_set_values* to do the work.

#### 2.3.6 mainframe_get_values

This function reads all the parameters from the mainframe and then writes them to a file, so that the humans and programs alike can read the current high voltage status.

#### 2.3.7 mainframe_set_channel

This function sets the values for a given channel.

#### 2.3.8 mainframe_set_values

This function checks that the values for each channel are acceptable and then sets them for all channels on the system.

#### 2.3.9 mainframe_set_limits

This function checks that the limits for each channel are acceptable and then sets them for all channels on the system.

#### 2.3.10 mainframe_calculate_max_safe_voltage

This function calculates the maximum voltage it is safe to ramp to in a single step. We consider two regimes, allowing greater steps at lower voltages than higher ones.

#### 2.3.11 mainframe_check_values

This function checks that the values requested by the user are acceptable. It is here where we enforce the policies of ramp up and down rates, maximum step at a given voltage etc.

#### 2.3.12 mainframe_analyse_status

This function analyses the status flag and if warning bits are set, it calls the warning script and if error bits are set it calls the trip script. It also logs to the system logger.

#### 2.3.13 mainframe_call_trip_script

This function calls the trip script passing the channel number, name of channel, demand voltage, measured voltage, measured current, current limit and current time on the command line.

#### 2.3.14 mainframe_call_warn_script

This function calls the warning script passing the channel number, name of channel, demand voltage, measured voltage, measured current, current limit and current time on the command line.

### 2.4 limits.c

This file contains the code for watching the limits configuration file and reading it if it changes. It has the functions:

- int limits_check_for_changes();
- int limits_read_file();
- int limits_parse_channel(int argc, char **argv);

#### 2.4.1 limits_check_for_changes

This function is called from the main loop on each iteration and it checks to see if the modification time of the limits configuration file has changed since the last time. If so (and always on the first iteration), it calls *limits_read_file* to read the file.

### 2.4.2 limits_read_file

This function reads the limits configuration file and uses *parse_cmd_line* to parse each line. It only understands one command "CHANNEL" and if that is found, it calls *limits_parse_channel*.

### 2.4.3 limits_parse_channel

This is called for each channel line found in the limits file. It stores the values for that channel and marks it as being configured.

## 2.5 channel.c

The code in this file reads commands from the command pipe and parses them, calling the appropriate callbacks.

- int channel_check_for_command();
- int channel_read_line(FILE *fp);
- int channel_kill(int argc, char **argv);
- int channel_enable(int argc, char **argv);
- int channel_disable(int argc, char **argv);
- int channel_voltage(int argc, char **argv);
- int channel_ramp_up(int argc, char **argv);
- int channel_update_mainframe();

### 2.5.1 channel_check_for_command

This function checks the named pipe for a command, and if there is one, it calls *channel_read_line* to read it.

### 2.5.2 channel_read_line

This function reads a single line from the pipe and uses *parse_cmd_line* to parse it. It accepts five commands, which are handled by the appropriate functions.

### 2.5.3 channel_kill

This function is called by *parse_cmd_line* from *channel_read_line* if the kill command is sent to the command pipe. It causes all channels to be ramped down safely. Any arguments on the command line are logged as the reason for the kill, but do not influence the way the command works.

### 2.5.4 channel_enable

This function is called by *parse_cmd_line* from *channel_read_line* if the enable command is sent to the command pipe. It turns on a given channel.

### 2.5.5 channel_disable

This function is called by *parse_cmd_line* from *channel_read_line* if the disable command is sent to the command pipe. It turns off a given channel.

### 2.5.6 channel_voltage

This function is called by *parse_cmd_line* from *channel_read_line* if the voltage command is sent to the command pipe. It sets the voltage of a given channel.

### 2.5.7 channel_ramp_up

This function is called by *parse_cmd_line* from *channel_read_line* if the ramp_up command is sent to the command pipe. It tells a given channel to ramp up to the highest voltage which is allowed by the policy. Or, if the argument is "-a", it ramps all enabled channels up.

### 2.5.8 channel_update_mainframe

This function does nothing, if parsing a command has changed nothing, but if it has, it uses *mainframe_set_values* to set the values on the mainframe.

## 2.6 logmessage.c

This code is used to log messages to the system logger. There are two functions:

- int logmessage(int level, char *fmt, ...);
- int logsetlevel(int level);

The symbolic values for log levels, as well as the prototypes for the functions are defined in *logmessage.h*.

### 2.6.1 logmessage

This function sends the text of the message to the system logger (syslog) if the level is lower than the current level. The second and remaining parameters work like *printf* style functions with a format and variable parameters. E.g.

```
logmessage(LOG_WARN, "Some text with floating point value f=%f", myfloat);
```

### 2.6.2 logsetlevel

This function sets the current level of logging. The higher it is, the less logging. The default is *LOGLEV_WARN* which means only warnings and fatal errors are logged. Informational and debugging messages are not logged.

## 2.7 parse_cmd.c

This code is used to parse command lines and call the appropriate functions to handle those commands. It is quite generic and is reused from several other programs.

The file *parse_cmd.h* defines a type parse_command, which consists of a key string and a pointer to a function. The most important function is *parse_cmd_line* which takes two arguments. The first is a string containing the command line, to be parsed and the second is an array of parse_command items, the last one having NULL pointers for both key and function.

This function breaks up the command line into arguments (like argc, argv in C) finds the command by comparing each key in the array with argv[0] and if it finds a match, it calls the corresponding function.

The functions all have the same form as the *main* function in C. i.e.

```
int mycommand(int argc, char **argv);
```

# 3 The control program hv_control

This program is a graphical user interface to the server. It sends commands to the pipe which is read by the server and reads from the status file, which is periodically written by the server.

The files for this program are:

- hv_control.c
- channel_widget.c
- parse_cmd.c

Note that parse_cmd.c is the same command line parsing code used by the server (see section 2.7).

## 3.1 hv_control.c

This is the main part of the GUI. It consists of the following functions:

- int main(int argc, char **argv):
- void create_main();
- gboolean idle_handler(gpointer data);
- gint read_channel_data(gchar *filename);
- int parse_data(int argc, char **argv);
- int parse_time(int argc, char **argv);
- void ramp_up_all(gpointer user_data);
- void ramp_down_all_confirmed(GtkWidget *dialog, gint response, gpointer user_data);
- void ramp_down_all_query(gpointer user_data);
- gboolean delete_callback(GtkWidget *widget, GdkEvent *event, gpointer user_data);

### 3.1.1 main

This function is the main entry point to the program. It performs the following tasks:

- initialise gtk (*gtk_init*)
- initialise internationalisation (*setlocale, bindtextdomain, textdomain*)
- call *create_main* to create the main window
- setup the idle handler
- enter the gtk main loop (*gtk_main*).

### 3.1.2 create_main

This function creates the main window. This is a top level window, with a scrolled window in it. Inside that is a vertical box with a viewport. Within that is a table, which contains a grid of channel widgets and a horizontal box containing three buttons: ramp up, ramp down and quit.

Pressing the ramp up button causes the *ramp_up_all* function to be called. Pressing the ramp down button causes the *ramp_down_all_query* function to be called, while pressing quit causes the *delete_callback* function to be called.

Three signals of the channel widgets are connected to functions:

- void server_change_voltage(gpointer user_data);
- void server_enable(gpointer user_data);
- void server_disable(gpointer user_data);

#### 3.1.3 server_change_voltage

This is called if the user changes the voltage on a channel widget, by moving the slider. It sends the appropriate command to the server.

#### 3.1.4 server_enable

This is called if the user turns on a channel on a channel widget, by clicking on "ON". It sends the appropriate command to the server.

#### 3.1.5 server_disable

This is called if the user turns off a channel on a channel widget, by clicking on "OFF". It sends the appropriate command to the server.

#### 3.1.6 ramp_up_all

This function is called when the ramp up button is clicked. It calls *channel_set_to_max_voltage* for each channel, to ramp each enabled channel up as much as is safe to do.

#### 3.1.7 ramp_down_all_query

This function creates a dialog box asking the user if he/she really wants to ramp down all channels. Clicking OK causes *ramp_down_all_confirmed* to be called.

#### 3.1.8 ramp_down_all_confirmed

Destroy the dialog box, that was created by *ramp_down_all_query*, for which this function is a callback. Then send the KILL command to the server.

#### 3.1.9 delete_callback

This function is called if the user clicks on the quit button, or deletes the window. It simply exits the program.

#### 3.1.10 idle_handler

This function is called whenever the application has nothing better to do. It checks the status file produced by the server and if it has changed, reads it again by calling *read_channel_data*.

#### 3.1.11 read_channel_data

This function uses *parse_cmd_line* to parse the channel data file generated by the server. There are two commands accepted: DATA and TIME. For the former, it calls *parse_data* and the for the latter *parse_time*.

#### 3.1.12 parse_data

This function is called by *parse_cmd_line* from *read_channel_data* when a DATA line is found. It obtains all the various parameters for the channel and calls *channel_widget_set_parameters* to store them in the channel widget for that channel.

#### 3.1.13 parse_time

This function doesn't do anything at all. It is called by *parse_cmd_line* from *read_channel_data* when a TIME line is found. This merely gives the timestamp, when the server wrote the data. In principle, we could check if the timestamp is recent or not, but we don't.

### 3.2 channel_widget.c

This file contains the code for the widget for a single channel. It is implemented as a gtk class. It contains four standard gtk class functions:

- GType channel_widget_get_type(void);
- void channel_widget_class_init(ChannelWidgetClass *myclass);
- GtkWidget *channel_widget_new(int number);

- void channel_widget_destroy(GtkObject *object);

and the following channel widget specific functions:

- void channel_change_demand_voltage(GtkVScale *vscale, gpointer user_data);
- void channel_enable(GtkButton *button, gpointer user_data);
- void channel_disable_query(GtkButton *button, gpointer user_data);
- void channel_disable_confirmed(GtkWidget *dialog, gint response, gpointer user_data);
- void channel_widget_set_name(GtkWidget *widget, gchar *name);
- void channel_widget_set_parameters(GtkWidget *widget, gint channel_enabled, gfloat demand_voltage, gfloat measured_voltage, gfloat max_voltage, gfloat ramp_up_rate, gfloat ramp_down_rate, gfloat measured_current, gfloat max_current, gfloat current_time, gint channel_status);
- void channel_widget_set_colour(GtkWidget *widget);
- void channel_set_to_max_voltage(GtkWidget *widget); float channel_calculate_max_safe_voltage(float measured_voltage, float max_voltage);

The widget also creates three new signals:

- changed - when the voltage is changed
- enabled - when the channel is turned on
- disabled - when the channel is turned off

#### 3.2.1 channel_widget_get_type

Returns the type of the widget.

#### 3.2.2 channel_widget_class_init

Class initialisation for channel widget class.

#### 3.2.3 channel_widget_widget_new

Create a new instance of a channel widget.

#### 3.2.4 channel_widget_destroy

Destroy an instance of a channel widget.

#### 3.2.5 channel_change_demand_voltage

This function sets the demand voltage. It is called when the slider is moved. It emits the signal “changed”.

#### 3.2.6 channel_enable

This function turns on a channel. It is called when the ON button is clicked. It emits the signal “enabled”.

#### 3.2.7 channel_disable_query

This function turns off a channel. It is called when the OFF button is clicked. If the voltage is below 50 Volts or the channel has tripped, it calls *channel_disable_confirmed* to turn off the channel, but otherwise, it pops up a dialog box, asking the user to confirm. Clicking on OK in that box, calls *channel_disable_confirmed*.

#### 3.2.8 channel_disable_confirmed

This function is called from *channel_disable_query* either when the voltage is below 50 Volts, the channel has tripped, or after the user has confirmed. It emits the signal “disabled”.

#### 3.2.9 channel_widget_set_name

This function sets the name of the channel.

#### 3.2.10 channel_widget_set_parameters

This function sets the parameters for the channel. It does all the updating for the GUI for that channel. It also calls *channel_widget_set_colour* to set the appropriate background colour according to the status (on, off, trip).

The first time the function is called for a channel, it performs some initialisation tasks, creating the pixmaps for arrows etc.

It calls *channel_widget_calculate_max_safe_voltage* to calculate the maximum safe voltage, based on the maximum voltage allowed for the channel and the present voltage. It sets the range of the the voltage slider, so that it is not possible to exceed this value.

It sets the labels for the measured voltage and currents and calls *channel_widget_set_colour* to set the appropriate colour for the background based on status. It also activates or deactivates the on and off buttons as appropriate.

It draws bars to indicate the measured voltage and current and a horizontal line for the demand voltage.

If we are ramping up or down, a arrow is shown to indicate this. Finally, if the channel is in warning or trip status, a red cross is drawn over everything.

#### 3.2.11 channel_widget_set_colour

This function sets the background colour according to the channel status. Yellow means off, green means on, pale orange means that the channel has tripped.

#### 3.2.12 channel_widget_set_to_max_voltage

This function sets the voltage for that channel to its maximum allowed value, based on the policy. It emits the "changed" signal.

#### 3.2.13 channel_widget_calculate_max_safe_voltage

This function calculates the maximum voltage it is presently safe to ramp up to for a given channel.

# 4 The hv_generate_html program

This program consists of two files:

- hv_generate_html.c
- parse_cmd.c

Note that parse_cmd.c is the same command line parsing code used by the server (see section 2.7).

## 4.1 hv_generate_html.c

This file consists of the following functions:

- int main(int argc, char **argv);
- int do_header();
- int do_body();
- int do_trailer();
- int parse_data(int argc, char **argv);
- int parse_time(int argc, char **argv);
- int treat_channel_status(int status);
- int treat_board_status(int status);

### 4.1.1 main

This is the main entry point. It just calls *do_header, do_body* and then *do_trailer*.

### 4.1.2 do_header

Generate the html header. This writes a standard header to standard output.

### 4.1.3 do_body

Generate the html body. This reads the status file created by the server and uses *parse_cmd_line* to parse it. If it finds a DATA line, it calls *parse_data* and if it finds a TIME line it calls *parse_time*.

### 4.1.4 do_header

Generate the html trailer. This is just writes:

```
</table>
</body>
</html>
```

to standard output.

### 4.1.5 parse_data

This function generates the html for a single channel, calling *treat_channel_status* and *treat_board_status* to generate text versions of the numerical status value.

### 4.1.6 treat_channel_status

Convert the channel status value into a text string, which is written to standard output.

### 4.1.7 treat_board_status

Convert the board status value into a text string, which is written to standard output.

### 4.1.8 parse_time

This writes the timestamp to standard output in html. Note that it also compares the timestamp with the current time and adds a warning if it is more than 10 seconds old.

# 5 The hv_kill script

This is a trivial script which sends the word “KILL” to the server’s command pipe. This causes the server to ramp down all channels. Any arguments given on the command line are passed as a reason, which is logged, but does not influence the way the command works.

# 6 The hv_unkill script

This is a trivial script which sends the word “UNKILL” to the server’s command pipe. This causes the server to enable any channel, for which the measured voltage is over 100 Volts, and to set the demand voltage to that measured voltage. In other words, it cancels a kill, but doesn’t ramp back up.

# 7 The hv_enable script

This is a trivial script which sends the word "ENABLE channel_number" to the server's command pipe. This causes the server to turn that channel on.

# 8 The hv_disable script

This is a trivial script which sends the word "DISABLE channel_number" to the server's command pipe. This causes the server to turn that channel off.

# 9 The hv_set_voltage script

This is a trivial script which sends the word "VOLTAGE channel_number voltage" to the server's command pipe. This causes the server to set the specific channel at the given voltage.

# 10 The hv_ramp_up script

This is a trivial script which sends the word "RAMP_UP" to the server's command pipe. It takes a single argument, which is either the channel to ramp up, or "-a" for all enabled channels. This causes the server to ramp up that channel (or all enabled channels if "-a" was specified) to the maximum safe voltage.